International Journal of Computer Techniques–IJCT Volume 13 Issue 3, May-June - 2026

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

A CodeBERT-Driven Framework for Multi-Class Software Defect Prediction Using Defect Categorization

1st Baidehi Jena
Department of CSE-AIML
GITA Autonomous College
Bhubaneswar, India
baidehicse@gmail.com

2nd Debasish Pradhan
Department for AIML
GITA Autonomous College
Bhubaneswar, India
debashish2.cse@gmail.com

3rd Dolagovinda Mahanta
Department of CSE-AIML
GITA Autonomous College
Bhubaneswar, India
dolagovindacse@gmail.com

***Abstract*—Ensuring software reliability through early defect detection and prevention is increasingly important as software systems grow more complex. Automated testing has emerged as an effective solution for producing reliable and efficient code, while machine learning-based approaches, particularly those leveraging natural language models, have gained significant attention for building advanced defect prediction systems. In this paper, we propose a novel framework for automated software defect prediction that targets eight specific defect types: SIGFPE, NZEC, LOGICAL, SYNTAX, SIGSEGV, SIGABRT, SEMANTIC, and LINKER. The study utilizes a custom dataset comprising nine classes, including eight categories of programming errors and one representing error-free code, with the aim of improving defect detection in code snippets and enhancing software development processes. The proposed method employs a CodeBERT-based model with optimized hyperparameters to achieve superior predictive performance. Comparative evaluation against established models such as RoBERTa, Microsoft CodeBERT, and GPT-2 demonstrates that the proposed approach achieves notable improvements, with accuracy gains of up to 20% in binary classification and 7% in multi-class classification. Experimental findings further validate the effectiveness of neural language models, particularly CodeBERT, in software defect prediction, highlighting their potential to significantly enhance software testing practices and overall code quality within the software development lifecycle.**

***Index Terms*—Software Defect Prediction, CodeBERT, Machine Learning, Deep Learning, Natural Language Processing, Automated Testing, Multi-Class and Binary Classification, Software Reliability, Defect Classification, Code Analysis, Neural Language Models**

I. Introduction

The rapid evolution of software systems has significantly increased their complexity, making software reliability a critical concern in modern software engineering. As applications grow in size and functionality, the likelihood of defects during development also increases, leading to higher maintenance costs and reduced system performance. Software Defect Prediction (SDP) has therefore become an essential research area, aiming to identify defect-prone modules at early stages of development to minimize testing effort and improve overall software quality.

Traditional SDP approaches primarily rely on statistical techniques and classical machine learning models that utilize handcrafted features extracted from source code. These methods typically perform binary classification, categorizing code segments as either defective or non-defective. Although such approaches provide a baseline for defect detection, they lack the ability to identify the specific nature of defects, thereby limiting their effectiveness in real-world debugging and maintenance processes.

Recent advancements in deep learning, particularly transformer-based architectures, have significantly enhanced the effectiveness of SDP. Models such as CodeBERT are capable of capturing both syntactic and semantic relationships within source code. By leveraging large-scale pretraining, these models enable improved understanding of programming structures and context, leading to more accurate defect prediction. However, most existing approaches still operate at a coarse-grained level, focusing primarily on binary or limited multi-class classification.

To address this limitation, multi-class defect prediction techniques have been introduced, allowing defects to be categorized into broader classes such as syntax errors, logical errors, and runtime faults. While this improves interpretability, it still lacks the granularity required for precise debugging. Each defect category consists of multiple sub-types, each with distinct characteristics and requiring different corrective actions. For example, logical errors may arise from incorrect conditions, infinite loops, or flawed algorithm design, all of which require specific debugging strategies.

In this context, the proposed work introduces an enhanced hierarchical software defect prediction model that incorporates sub-class classification into the existing multi-class framework. The model operates in two stages: first, a main class classifier identifies the general category of the defect, and second, a sub-class classifier refines the prediction by identifying the specific type of defect. This hierarchical structure enables fine-grained defect analysis and improves the usability of prediction results.

The architecture of the proposed model is based on a transformer-driven pipeline, where input source code is to-

kenized and converted into high-dimensional embeddings. These embeddings capture both structural and contextual information of the code. The processed representations are then passed through a main class classifier to predict one of the predefined defect categories. Subsequently, a sub-class classifier, conditioned on the main class output, performs detailed classification to determine the exact defect type. This approach closely aligns with practical debugging workflows followed by developers.

Furthermore, the model leverages advanced learning strategies such as transfer learning and multi-task learning. Transfer learning enables the use of pre-trained models to improve performance even with limited data, while multi-task learning allows the model to simultaneously optimize both main class and sub-class predictions. These techniques enhance the efficiency, scalability, and generalization capability of the proposed system.

The introduction of sub-class classification also requires enhancements in dataset design and preprocessing. The dataset must include hierarchical labels, where each code snippet is annotated with both a main defect class and a corresponding sub-class. Ensuring a balanced distribution across classes is essential to avoid bias in model predictions. Techniques such as data augmentation and resampling can be applied to improve dataset quality and robustness.

The proposed hierarchical framework offers significant practical benefits in software engineering applications. By providing fine-grained defect predictions, it enables more efficient debugging, targeted testing, and automated error correction. The model can be integrated into development environments and continuous integration pipelines to provide real-time feedback, thereby improving productivity and software quality.

In conclusion, the integration of sub-class defect classification into the software defect prediction framework represents a significant advancement over traditional approaches. By combining transformer-based models with hierarchical classification techniques, the proposed model provides more precise, interpretable, and actionable predictions. This approach enhances software quality assurance and opens new opportunities for intelligent and automated software maintenance systems.

A. Problem Domain

The current state of research in software defect prediction predominantly relies on a binary classification paradigm, wherein software modules are categorized as either defective or non-defective. Although widely adopted, this approach fails to capture the inherent complexity and diversity of defect types present within software systems. Most existing studies primarily focus on distinguishing between clean and faulty code, without considering the detailed classification of specific defect categories embedded in the codebase.

This limitation reveals a significant research gap, as the finer granularity of defect identification remains largely unexplored. The absence of multi-class classification restricts the practical applicability of defect prediction models, particularly in scenarios requiring precise debugging and fault localization.

SDP utilizes historical software project data, software metrics, and defect information to build predictive models that classify modules as defective or non-defective [1]. These models help software engineers allocate testing resources efficiently and improve software quality assurance processes. By focusing on defect-prone modules, organizations can reduce testing time, minimize debugging effort, and enhance the reliability of software products. Early prediction of defects plays a crucial role in preventing failures and improving the success rate of software projects [2].

The objectives of this research paper are given below:

- The primary objective is to enhance the accuracy of software defect prediction by leveraging transformer-based models such as CodeBERT [3]which capture both syntactic and semantic features of source code.
- The model aims to overcome limitations of traditional approaches by moving from:
  Binary classification (defective / non-defective) Basic multi-class classification to a fine-grained hierarchical classification framework.
- A key objective is to classify defects not only into main categories but also into specific sub-types, enabling:
  Detailed defect identification Better understanding of error causes
- The model is designed to implement a two-level classification architecture:
  Level 1: Main defect class prediction Level 2: Sub-class defect prediction
  This structure improves interpretability and aligns with real-world debugging processes.
- Another objective is to design a dataset with hierarchical labels (main class + subclass). Address class imbalance using augmentation and resampling techniques

The format of the given research work is as follows: section II delivers a recap of the literature analysis, while Section III elaborates on the proposed methodology comprehensively. Section V covers an in-depth discussion of the experimental phase, Section **??** summarizes the results and recommendations for ongoing exploration, and Section **??** discusses about the challenges and limitations.

II. Related Works

Recent advancements in Software Defect Prediction (SDP) have evolved from traditional statistical and machine learning techniques to more sophisticated deep learning-based approaches. Earlier methods primarily relied on handcrafted features and classifiers such as decision trees, support vector machines, and random forests, which mainly performed binary classification of defective and non-defective modules. While these approaches provided baseline performance, they lacked the ability to capture complex semantic relationships within source code. With the emergence of transformer-based models like CodeBERT [4], researchers have been able to leverage contextual and structural information in code, significantly

improving prediction accuracy. Recent studies have further extended SDP to multi-class classification, enabling the identification of different defect categories such as syntax, logical, and runtime errors . However, these models still operate at a coarse-grained level and fail to provide fine-grained insights into specific defect types. Additionally, challenges such as class imbalance, limited labeled datasets, and lack of hierarchical labeling remain prominent in existing literature. To overcome these limitations, current research trends are moving towards hierarchical and fine-grained defect prediction frameworks, integrating sub-class classification, transfer learning, and multi-task learning to improve both accuracy and interpretability. This progression highlights the need for more advanced models that not only detect defects but also precisely classify their underlying causes, thereby supporting efficient debugging and automated software maintenance.

In conventional software defect prediction studies, the main objective has been to detect defects within source code by utilizing features generated from deep learning models. These approaches typically perform binary classification, labeling code as either clean [5] or defective [6]. A wide range of language models have been applied in this domain, starting from traditional neural embedding techniques such as GloVe and Word2Vec [7] to more advanced transformer-based architectures like BERT. Significant research efforts within the software engineering community have explored the use of deep learning methods for source code analysis [8], [9] Modern transformer-based models, including BERT and its variants such as RoBERTa and CodeBERT, are capable of processing both natural language and source code, thereby effectively bridging the gap between code semantics and textual representations of defects.

Transformer architectures form the backbone of these models, employing attention mechanisms to capture long-range dependencies in sequences. This capability enables them to associate semantic elements [10]within code with descriptive textual patterns of defects. Unlike traditional techniques that rely solely on code-level features, contemporary approaches integrate both syntactic and semantic information, resulting in more robust predictions. The process involves encoding input data into feature representations and subsequently decoding them into conditional probabilities through multi-head attention mechanisms and fully connected layers. Furthermore, bidirectional transformer models [11] facilitate context understanding in both forward and backward directions, a capability not present in certain unidirectional models such as GPT.

A. Deep Learning-Based Approaches

Research in software defect prediction using deep learning incorporates both manually engineered features and automatically generated representations. Hand-crafted features, when utilized with deep learning architectures [3] such as fully connected neural networks, have demonstrated improved predictive performance compared to traditional machine learning methods. In contrast, generated features derived directly from source code capture both structural and semantic information, which are effectively leveraged by models such as Long Short-Term Memory (LSTM) networks and Transformer-based architectures for defect prediction.

Abstract Syntax Tree (AST) sequences and path-based representations [11] have also been widely adopted to model source code, as they provide a balance between information richness and computational efficiency, particularly when training data is limited. Recent advancements [11] have introduced several variants of CodeBERT, including CodeBERT-NT, CodeBERT-PS, CodeBERT-PK, and CodeBERT-PT [12], which have demonstrated notable improvements in defect prediction across different projects and software versions. These models are especially effective in binary classification tasks, such as identifying buggy versus non-buggy code, as well as capturing recurring defect patterns.

Li et al. [3] proposed WELL (Weakly Supervised Bug Localization), a method that transforms bug detection models into bug localization systems using weak supervision. Their evaluation across multiple datasets showed superior performance compared to traditional supervised deep learning approaches, particularly in handling variables and binary operator-related defects. By leveraging easily obtainable buggy and non-buggy data, their approach fine-tunes CodeBERT for token-level defect localization. However, most deep learning models are typically designed to jointly perform bug detection and localization without relying on weak supervision derived from binary classification tasks.

Several other studies have explored deep learning techniques for defect prediction and vulnerability detection. Wang et al. [13] introduced a deep belief network for defect prediction, while Choi et al. [14] applied neural networks to detect buffer overflows. VulDeePecker [15] focused on identifying software vulnerabilities, whereas DeepBugs [16] employed a feedforward neural network to detect bugs in function calls and binary expressions. Due to the scarcity of labeled datasets, many approaches rely on synthetic data, where bugs are artificially injected into code to generate annotated datasets with known defect locations.

VarMisuse [17] is one of the most extensively studied tasks in bug localization using deep learning. Seq2Ptr [15] utilizes a sequence-to-pointer architecture to both detect and localize defects, enabling easier correction. Similarly, Graph-Sandwich and GREAT [18] introduced graph-based architectures to generate distributed representations of source code for improved defect localization. CuBERT [19] addressed multiple defect detection and localization tasks, including VarMisuse [20], using a unified framework.

Demirci et al. [19] applied stacked Bidirectional LSTM (BiLSTM) and GPT-2 models for detecting malicious code. Their approach involved analyzing assembly instructions extracted from executable files, treating them as textual data. They constructed multiple datasets evaluated using document-level and sentence-level analysis models. In another study [17], a malware classification technique was proposed using a text-based BiLSTM model [21], where API calls and opcodes extracted from disassembled files were used

B. Machine Learning-Based Approaches

Machine learning (ML) techniques have been extensively explored in software defect prediction (SDP), focusing on improving prediction accuracy through effective feature engineering and model optimization. McMurray and Sodhro [19] conducted a comprehensive study comparing various feature extraction (FE) methods, such as Principal Component Analysis (PCA) and Partial Least Squares (PLS), along with feature selection (FS) techniques including Fisher Score, Recursive Feature Elimination (RFE), and Elastic Net. These methods were evaluated in combination with multiple ML algorithms, including Support Vector Machine (SVM), Logistic Regression (LR), Na¨ıve Bayes (NB), K-Nearest Neighbors (KNN), Multi-layer Perceptron (MLP), Decision Tree (DT), and ensemble approaches such as Bagging, AdaBoost, XGBoost, Random Forest (RF), and Stacking. The findings indicate that FE and FS techniques can significantly influence model performance, with PLS combined with FS methods [22] consistently yielding strong improvements, while PCA with Elastic Net also demonstrated acceptable gains.

Wei et al. [18] proposed a novel defect prediction approach using a Local Tangent Space Alignment Support Vector Machine (LTSA-SVM), where LTSA is employed for dimensionality reduction and SVM for classification. Parameter tuning was performed using grid search and ten-fold cross-validation, demonstrating the model's effectiveness. Choi and Nam [1] introduced WINE, a technique designed to reduce validation costs by identifying representative warnings with structural similarities, thereby aiding in the detection of false positives and negatives during regression testing.

Zhao et al. [18] developed DouBiGRU, a defect detection model that integrates Bidirectional Gated Recurrent Units (BiGRU) with an attention mechanism, evaluated on datasets such as NVD and SARD. Chen et al. [23] compared supervised and unsupervised SDP approaches, revealing that unsupervised methods based on metrics like Lines of Code (LOC) and Response for a Class (RFC) perform competitively, particularly in cross-project and cross-version scenarios. Although supervised methods benefit from differential evolution optimization, LOC-based unsupervised approaches remain strong baselines. DPDF [12], a deep forest-based model, utilizes a cascade structure to enhance defect feature representation, achieving improved AUC scores over traditional ML methods. Strdowski and Madeyski [24] analyzed the application of ML in SDP from a business perspective, highlighting the limited use of industrial datasets and emphasizing the need to align research with practical industry requirements.

Further advancements include big data-driven approaches [25], which significantly reduce defect detection time while improving efficiency and accuracy. Wang and Yuan [26] proposed an automated defect classification system (DACS) [27] that achieves high accuracy and faster processing compared to traditional methods. Alkhasawneh [27] introduced a model combining feature selection with Radial Basis Function (RBF) classification, evaluated using NASA datasets. Mustaqeem and Saqib [22] developed a hybrid PCA-SVM model that improves performance by reducing dimensionality and optimizing classification, although SVM's lack of probabilistic interpretability remains a limitation.

Nassif et al. [28] applied Decision Trees and Logistic Regression to classify defective modules using PROMISE repository datasets. Abbineni and Thalluri [20] evaluated Learning-to-Rank (LTR) models, analyzing the effects of imbalance learning and feature selection, and found that bug count serves as a more stable ranking criterion than bug density.

Recent studies have also explored active learning strategies to reduce labeling effort by selecting the most informative training instances. When combined with ensemble learning, these approaches can reduce training data requirements significantly while maintaining prediction accuracy, as measured by metrics such as AUC, Kappa, and MCC.

Overall, software defect prediction aims to identify potential defects early in the development lifecycle, enabling efficient allocation of quality assurance resources. Ongoing research continues to focus on improving the practicality, interpretability [29], and scalability of ML-based defect prediction models, while addressing emerging challenges and advancing future methodologies [30].

Recent advancements in software defect prediction have explored hybrid and ensemble-based approaches to improve predictive performance. In [22], a CNN-MLP model was proposed that combines semantic features with traditional metrics using a hybrid architecture. The model employs a gated fusion mechanism to effectively integrate features, resulting in superior performance compared to state-of-the-art methods in both non-effort-aware metrics (such as F1-score and AUC) [12] and effort-aware measures (such as PofB20). This demonstrates its capability to enhance defect detection while simultaneously reducing development effort.

Similarly, the hybrid machine learning framework presented in [21], which integrates K-Nearest Neighbors, Gaussian Na¨ıve Bayes, Support Vector Machines, and Neural Networks, has shown consistent improvements across multiple datasets. The approach achieves strong performance in key evaluation metrics, including accuracy, precision, recall, and F1-score, while maintaining a balance between false positives and false negatives, thereby contributing to more reliable defect prediction.

An ensemble-based approach [31] further advances this domain by combining four diverse classifiers through a voting mechanism. The resulting VESDP model demonstrates superior performance on benchmark datasets, such as those provided by NASA, outperforming several contemporary methods and highlighting the effectiveness of ensemble learning in improving prediction accuracy.

Despite the growing number of approaches and advancements in deep learning models, many existing methods are limited by their reliance on binary classification, which restricts their ability to fully capture the complexity of software defects. To address this limitation, the proposed approach extends defect prediction to a multi-class framework, enabling more detailed defect identification and improved understanding of

code behavior. This facilitates efficient bug detection, classification, and resolution, ultimately aiming to enhance prediction accuracy while reducing time and resource costs in software testing and development processes. Such advancements contribute to a deeper understanding and effective management of diverse defect types, thereby improving overall software quality.

III. Proposed Methodologies

In the present work, a multi-class software defect prediction model (MSDP) is introduced to effectively identify various types of software defects spanning nine distinct categories. Unlike prior studies that limit their scope to binary classification (buggy versus non-buggy), this approach leverages a dataset containing nine defect classes, enabling more detailed and accurate classification of defects, as depicted in Fig. 1. 1

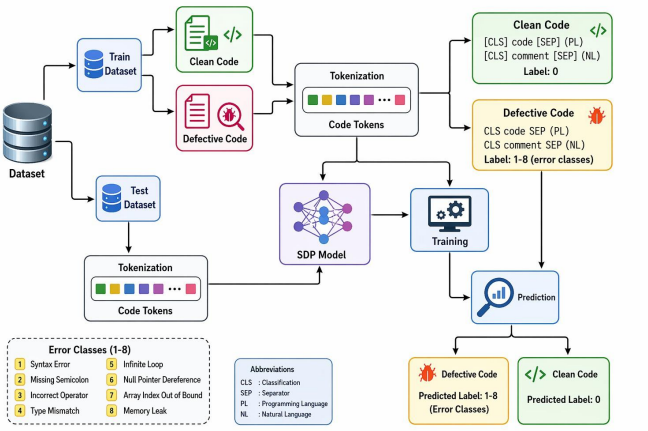


Fig. 1. Complete process of the proposed MSDP model for software defect prediction.

The proposed approach employs a CodeBERT-based pre-trained model, widely recognized for its capability in handling source code representations, to perform defect classification on the dataset. A schematic overview of the model is presented in Fig. 1.

A. Architectural Overview of the System

The high-level architecture of the proposed method illustrates the sequential steps involved in predicting software defects. The process begins with tokenization, where code snippets from the dataset are segmented into meaningful tokens representing different programming components, such as keywords, operators, identifiers, and comments. These tokens are subsequently mapped to a predefined vocabulary that captures a wide range of programming language elements, including reserved keywords like ‘if’ and ‘else’, as well as identifiers such as variable and function names.

Once tokenization is completed, each token is mapped into a numerical embedding, generating high-dimensional vector representations that allow the model to efficiently interpret and analyze the source code. These embedded vectors are then supplied to the model for further processing, where it learns the contextual relationships among tokens and produces predictions for defect identification. Although Fig. 1 depicts a general software defect prediction process, this research extends the framework to address multi-class defect classification. The adoption of multi-class classification represents a significant contribution of this work, enabled through the careful design of the dataset. A detailed description of the proposed model is provided in our research, we curated a bespoke dataset to facilitate soft-ware defect predictions, encompassing nine distinct classes: eight classes representing common programming errors and one class denoting error-free code. These errors were chosen as they are frequent, recurring issues in programming, often causing substantial time overheads. Table 1 showcases the distribution of code snippets across the nine defect classes collected specifically for this dataset, all compiled in the C++ language. Opting for C++ was deliberate due to its prevalence in critical systems, including finance and embedded systems, and its extensive usage in systems software, gaming, and performance-centric applications. By utilizing C++ code, we aimed for a realistic representation of codebases encountered in practical scenarios, exploiting its diverse coding paradigms encompassing object-oriented, procedural, and generic programming. the following section.

B. Dataset Generation

In this work, a tailored dataset was curated to facilitate effective software defect prediction, encompassing nine distinct classes, including eight categories of commonly occurring programming errors and one class representing error-free code. These defect types were chosen due to their recurring nature and the considerable time overhead they introduce during software development and debugging processes.

The distribution of code snippets across these nine classes is presented in Table 1, with all samples developed in the C++ programming language. The selection of C++ was deliberate, owing to its widespread use in critical systems such as finance and embedded environments, as well as its prominence in systems software, gaming, and performance-critical applications. Furthermore, its support for multiple programming paradigms—including object-oriented, procedural, and generic programming—ensures a realistic and diverse representation of real-world codebases. C++ codebases are inherently complex, primarily due to factors such as manual memory management, extensive use of pointers, and sophisticated syntax. Identifying defects within such codebases plays a crucial role in mitigating the challenges associated with modern software development.

The dataset includes various error categories such as SIGFPE, SIGABRT, NZEC, logical, syntax, SIGSEGV, semantic, and linker errors, along with instances of error-free code. These categories range from basic syntax issues to more complex problems like segmentation faults and logical inconsistencies, offering a broad representation of programming challenges. By incorporating these real-world error types,

TABLE I
DEFECT CLASS DISTRIBUTION

No.	Defect Class	No. of Instances (Total 5318)
1	No Error	546
2	SIGFPE	503
3	SIGABRT	515
4	NZEC	541
5	LOGICAL	552
6	SYNTAX	612
7	SIGSEGV	503
8	SEMANTIC	1047
9	LINKER	499

the dataset is designed to closely reflect practical software engineering environments, thereby improving its usefulness and real-world applicability.

The dataset was collected from various online sources, including GitHub and ChatGPT, and enriched with problem-based code samples from the MBPP dataset [34]. This approach ensured diverse and non-redundant C++ code covering a wide range of concepts. To maintain quality, extensive preprocessing was performed, including error correction, duplicate removal, and data normalization.

After collecting the code samples, they were processed through a scripting phase to convert them into a standardized dataset format. During this stage, each snippet was assigned a unique identifier and categorized according to its corresponding defect class. Relevant attributes were also initialized to ensure uniformity and completeness across the dataset. Figure 2 illustrates the formatted output, highlighting the workflow from data collection to cleaning and structuring, where IDs and target labels are assigned. Overall, the input code is systematically organized, refined, and classified into nine distinct labeled categories.

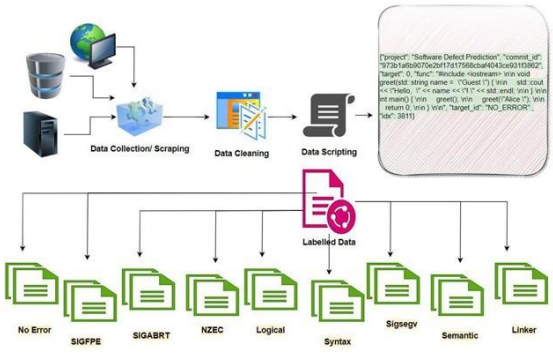


Fig. 2. Collection and Evaluation of the Software Defect Dataset

TABLE II
SOFTWARE DEFECT DATASET FOR BINARY CLASSIFICATION (BALANCED)

No.	Defect Class	No. of Instances (total 1046)
1	No Error	546
2	Error	500

This process was designed to systematically organize the data, making it suitable for effective analysis and model training. In addition, two more datasets were created, as shown in Tables 2 and 3. Although they use the same code snippets as the primary dataset, the classification is simplified to distinguish between buggy and non-buggy code. This setup facilitates comparison with existing models and helps evaluate the effectiveness of the proposed approach. This dataset is intended to support benchmarking across a range of defect prediction models and techniques. It provides a solid basis for evaluating how effectively different approaches handle predefined error categories, enabling meaningful comparison and analysis. For our experiments, we maintained fairness and reliability by carefully splitting the data into 60% for training, 20% for validation, and 20% for testing.

TABLE III
SOFTWARE DEFECT DATASET FOR BINARY CLASSIFICATION (IMBALANCE)

No.	Defect Class	No. of Instances (total 5318)
1	No Error	546
2	Error	4772

C. Token Representations

After splitting the dataset into training and testing sets, tokenization is carried out to create token embeddings that serve as input to the model. During this step, special tokens such as "[CLS]" (for classification) and "[SEP]" (for separation) are incorporated into the tokenized sequences.

1.[CLS] Token:The [CLS] token is inserted at the beginning of the input code sequence to represent the entire sequence in a single vector form. In classification tasks, the final hidden state of this token serves as a comprehensive representation of the input and is used for making the classification decision.

2.[SEP] Token:The [SEP] token is placed between different segments of the input code sequence to indicate boundaries. It helps the model distinguish between separate parts of the code and understand the overall structure, especially when multiple segments are present.

These tokens are incorporated during tokenization to help the model interpret the input sequence and support the classification task. They indicate the boundaries of the code sequence, such as its beginning and separation points. Afterward, the model is trained to perform software defect prediction by classifying the code into nine categories, including one class representing error-free code.

D. classification

CodeBERT achieved excellent results in binary software defect classification, outperforming several other deep learning models. It was chosen as the base model for its strong performance, effective understanding of code structure, and comparatively lower complexity among available approaches.

Because of limited data and the challenges involved in collecting and labeling software defect datasets, we adopted transfer learning in our approach. As a result, the model

parameters are initialized from a pre-trained model and then fine-tuned, allowing the model to be effectively applied to our generated dataset for software defect prediction.

For the proposed approach, we have to do software defect classification, so the configuration of the model is based on RoBERTa for sequence classification, which is more suitable as it is designed to handle classification tasks directly, leveraging the [CLS] token for predictions. It extends RoBERTa by adding a classification head on top of the base model. It includes additional layers tailored for classification tasks. Whereas for the Microsoft CodeBERT, the model is configured based on the RoBERTa model.

The token embeddings are fed into a model composed of 12 layers, each containing multiple sub-layers. These layers are stacked sequentially, allowing the model to progressively refine the representations of the input code tokens. With each layer, more complex patterns and dependencies within the source code are captured.

The Multi-Head Self-Attention mechanism allows the model to dynamically assign importance to different parts of the code sequence during token processing. With multiple attention heads, it can simultaneously focus on various positions and capture diverse relationships within the code. Following this, each layer incorporates a feedforward neural network that transforms the attention output through linear operations and activation functions such as ReLU. To enhance training stability and maintain effective information propagation, layer normalization and residual connections are applied after each sub-layer, including both the attention and feedforward components.

For defect prediction, the final representation of the [CLS] token is used and passed to a fully connected classification layer. The input IDs are tokenized code snippets that generate hidden representations, with outputs taken from the last (12th) layer as raw scores. A sigmoid function converts these scores into probabilities for binary classification, using binary cross-entropy loss. For multi-class classification, softmax activation is applied with categorical cross-entropy loss.

Algorithm: Preprocessing of Dataset for Software Defect Prediction

Input: Raw code snippets gathered from multiple sources (e.g., GitHub, ChatGPT, MBPP dataset)

Output: Preprocessed and labeled dataset for software defect prediction **Step 1: Data Cleaning and Normalization**

1) Set up the dataset with the collected raw code snippets.
2) For each code snippet:
   - Fix issues such as typographical mistakes, incomplete code segments, and improperly structured code.
   - Eliminate duplicate entries to prevent over-representation.
   - Standardize the code format, including indentation, structure, and style.
3) Return cleaned dataset

Step 2: Error Labeling and Class Categorization

1) Define defect classes (SIGFPE, SIGABRT, SYNTAX, LOGICAL, etc.)
2) For each cleaned snippet:
   - Perform manual labeling (e.g., segmentation fault, logic error)
   - Check class balance across categories
3) Assign unique identifiers
4) Return labeled dataset

Step 3: Data Augmentation

1) For each underrepresented class:
   - Apply mutation techniques:
     – Insert/delete lines of code
     – Modify control structures (e.g., for → while)
2) Add augmented samples to dataset
3) Return augmented dataset

Step 4: Standardization and Scripting

1) For each code snippet:
   - Assign unique identifier
   - Initialize attributes (project name, target, etc.)
   - Convert to standardized format
2) Return standardized dataset

Step 5: Binary Dataset Generation

1) Duplicate labeled dataset
2) Relabel:
   - Error classes → "Error"
   - Clean code → "No_Error"
3) Ensure class balance (if required)
4) Return binary dataset

IV. Evaluation

We meticulously tuned our model using various hyperparameters to attain optimal accuracy and results. Our dataset deliberately encompasses a broad spectrum of C++ language concepts, including pointers, memory management, object-oriented programming (OOP) concepts, and common types of defects. This comprehensive coverage ensured robustness and accuracy during experimentation.

In our model setup, we employed the Adam Optimizer with a learning rate of 1e-9 for multi-class classification and 1e-7 for binary classification. Tokenization was performed with a token size set to a block size of 400 for each code example. Padding was applied for smaller code snippets, while truncation was implemented for longer code sequences to maintain uniformity. The model underwent experimentation for a total of 5 epochs, assessing the loss function after each epoch. We incorporated an early stopping mechanism that stops the process if the loss value surpassed the calculated average loss for that epoch, exceeding a predefined patience value. The last accuracy obtained before stopping was recorded as the final result. For reference, Table 4 details the specific hyperparameter values utilized in our experimentation

A learning rate range of 1e-9 to 1e-7 balances precise updates and faster convergence. Six epochs provide sufficient training while managing computational cost. A batch size

TABLE IV
HYPERPARAMETERS USED DURING TRAINING AND EVALUATION OF PROPOSED APPROACH

No.	Hyperparameters	Values
1	Learning rate	1e-7, 1e-9
2	No. of epochs	6
3	Batch size	15
4	Seed	123456
5	Block size	400
6	Max grad norm	1.0

of 15 ensures a good trade-off between efficiency and generalization. A fixed seed guarantees reproducibility, while a block size of 400 captures long-range dependencies efficiently. Gradient clipping at 1.0 helps maintain stable training and prevents gradient explosion.

By thoughtfully choosing these hyperparameters and their values, we seek to optimize the model's training by balancing efficiency, performance, and reproducibility. Continuous monitoring during training and adjusting these parameters based on empirical results is crucial for achieving optimal performance.

A. Metrics

TABLE V
METRICS USED FOR PERFORMANCE EVALUATION

S.No.	Metrics
1.	Accuracy
2.	F1 measure
3.	Precision
4.	Recall
5.	Confusion matrix

The different classification metrics, which are obtained by the confusion matrix, evaluate the efficacy of the proposed model. The components of the confusion matrix are TP, TN, FP, and FN, which are applied to derive the metrics. Sensitivity, or recall, refers to the number of positive classifications by the classifier. It is expressed as

$$\text{PSEN} = \frac{A_1}{A_1 + A_4} \tag{1}$$

Specificity, evaluates the competency of the framework to correctly classify non-defective modules and is given by

$$\text{PSPE} = \frac{A_2}{A_2 + A_3} \tag{2}$$

precision, which is the ratio of true positive predictions to the total number of positive predictions made by the model.

$$\text{PPRE} = \frac{A_1}{A_1 + A_3} \tag{3}$$

The F1-score achieves equilibrium between false positive and false negative rates. It even yields a more reliable assessment of the model's effectiveness.

$$\text{PF1} = \frac{2 \times \text{PPRE} \times \text{PSEN}}{\text{PPRE} + \text{PSEN}} \tag{4}$$

Accuracy is the ratio of correct predictions (both positive and negative) to the total number of predictions made. It indicates the overall correctness of the classifier and is defined as

$$\text{PACC} = \frac{A_1 + A_2}{A_1 + A_2 + A_3 + A_4} \tag{5}$$

Confusion Matrix: This matrix offers a comprehensive evaluation of classification output quality. Diagonal elements represent accurately predicted values (true error and true no-error instances), while off-diagonal values denote misclassifications, aiding in assessing the accuracy and robustness of the classification output Utilizing these metrics collectively provides a holistic evaluation of the model's performance, ensuring a comprehensive understanding of its classification abilities across error and no-error instances.

TABLE VI
COMPARISON WITH OTHER METHODS FOR BINARY CLASSIFICATION (BALANCED)

S.No.	Metrics	MSDP (Ours)	GPT2	Roberta	MS CodeBERT
1.	Accuracy	**64.44**	44.00	44.12	44.44
2.	F1 score	**0.638009**	0.296844	0.307671	0.307692
3.	Precision	**0.692972**	0.211111	0.222200	0.222222
4.	Recall	**0.667500**	0.499900	0.500000	0.500000

B. EXPERIMENTAL RESULTS

In our experimentation we use different kinds of evaluations. Each of the evaluation performed is discussed in this section

1) BINARY CLASSIFICATION :

A) EXPERIMENTS ON BALANCED DATASET

In our experiments, we addressed a binary classification task by merging all eight defect categories into a single class labeled ' "Error," while instances without defects were categorized as "No_Error." To ensure a fair evaluation, the dataset was balanced between these two classes and split into training (60%), testing (20%), and validation (20%) sets.

The obtained accuracy and other evaluation metrics are reported in Table 6. Furthermore, our method is compared with established models such as RoBERTa, GPT-2, and Microsoft CodeBERT. Table 6 presents a detailed comparison across key metrics, including accuracy, precision, recall, and F1 score, along with their corresponding confusion matrices. Our approach clearly outperforms the baseline models, achieving an approximate 20% improvement in accuracy on the balanced binary dataset. Figure 4 shows the confusion matrix, highlighting the performance of the proposed model in the binary classification task.

A key feature of our model is its ability to predict both classes, "Error" and "No_Error," unlike other models that primarily focus on a single class. However, as observed in Figure 4, the model shows a tendency to predict the ' "Error" class more frequently than "No_Error." This behavior may be attributed to the complex and varied nature of error patterns, which can influence the model to favor error predictions. Software errors encompass a broad range of issues, from simple syntax mistakes to more complex logical faults, each

presenting distinct challenges for accurate classification. The diversity and intricacy of these patterns make them harder to distinguish, potentially causing the model to bias toward predicting errors. In contrast, instances of error-free code are generally more consistent and less complex, making them comparatively easier to identify.

This trend underscores the difficulty in correctly identifying "No_Error" instances, which may be influenced by the relatively simpler structure of such code segments. Nevertheless, the model's strong overall performance, along with its capability to classify both categories, reflects a notable improvement in binary classification for software defect prediction, as depicted in Figure 5.

B) EXPERIMENTS ON IMBALANCED DATASET

In our experiments with imbalanced data, all eight defect categories were merged into a single class labeled ' "Error," while the number of ' "No_Error" instances remained unchanged for the binary classification task. This resulted in a skewed dataset, where over 4500+ out of 5318 samples were labeled as "Error". The dataset was then divided into training (60%), testing (20%), and validation (20%) sets.

The achieved accuracy and other evaluation metrics are presented in Table 6. In addition, we performed a comparative study by evaluating all models against our proposed approach on the imbalanced dataset. Table 6 summarizes the results for key metrics, including accuracy, precision, recall, and F1 score. In this scenario, our proposed method achieved lower overall accuracy than the baseline models. Nevertheless, a key strength of our approach is its ability to identify both classes ''Error' and 'No_Error'—a capability that other models lack, as evidenced by the results in Table 7 and the confusion matrix. Figure 4 presents the confusion matrix, highlighting the performance of the proposed method in binary classification on an imbalanced dataset. It provides a detailed view of the classification outcomes, showing that instances of the 'Error' class were predicted more accurately than those of the 'No_Error' class. This behavior is primarily due to the significant class imbalance present in the dataset.

The imbalance in the dataset has significantly influenced the model's performance, resulting in a bias toward the majority 'Error' class. Specifically, the dataset contains a substantially larger number of 'Error' instances compared to 'No_Error', leading to an uneven sample distribution. This imbalance creates challenges during both training and inference, as the model tends to focus on learning patterns associated with the dominant class in order to minimize the overall loss. Consequently, the model becomes more proficient at identifying and predicting 'Error' instances, yielding higher accuracy for this class. In contrast, the relatively limited number of 'No_Error' samples restricts the model's ability to learn distinguishing features effectively, resulting in reduced predictive performance for that class.

2) MULTI CLASS CLASSIFICATION: Furthermore, the tendency of the model to favor the majority 'Error' class emphasizes the difficulty in accurately identifying instances of

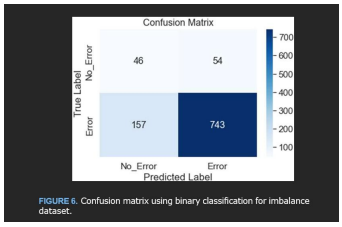
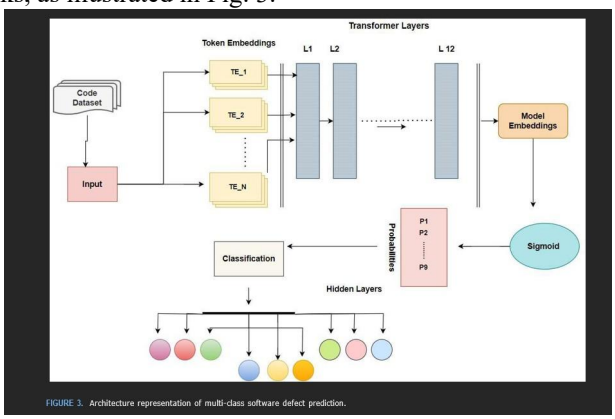


Fig. 3. Enter Caption

the minority 'No_Error' class. This bias reduces the model's sensitivity to subtle patterns that distinguish error-free code, often leading to misclassification. Such behavior highlights the need to address class imbalance during the data preprocessing stage to reduce its negative impact on model performance. Overall, the model's higher accuracy in predicting 'Error' instances compared to 'No_Error' reflects the strong influence of dataset imbalance on learning outcomes. To overcome this issue, it is essential to apply techniques such as resampling, data augmentation, or specialized loss functions to reduce bias and enhance classification performance across all classes. Nevertheless, despite its lower overall accuracy, the model's distinct ability to predict both classes demonstrates its value in handling imbalanced datasets in software defect prediction tasks, as illustrated in Fig. 3.



Architecture representation of multi-class software defect prediction..

C. DISCUSSION ON RESULTS

The findings of this study are important for several reasons: Research Innovation: This work presents a novel approach to software defect prediction. It is the first effort to create and evaluate a dataset specifically designed to predict nine different types of software defects, making it a significant contribution to the field. Addressing a Complex and Unexplored Problem: Predicting multiple defect types in software systems is a challenging task. Achieving a 7% increase in accuracy in such a relatively unexplored area reflects meaningful progress despite the complexity involved. Development of a Unique Dataset: The dataset introduced in this research has not been previously tested and provides a valuable resource for studying and predicting various software defects. Its broad coverage of defect categories

offers a solid base for future investigations. Improvement Over Existing Methods: The proposed approach outperforms established techniques by 7%, indicating advancement in defect prediction methods. Although the gain is moderate, it demonstrates a step toward more accurate and reliable models. Scope for Future Improvement: The promising results suggest that further refinements are possible. This research lays the groundwork for future efforts aimed at improving multi-class defect prediction systems.

In our experiments on the primary dataset comprising nine categories (eight defect classes and one 'No_Error' class), we performed multi-class classification. The data set was partitioned into training subsets (60%), testing (20%), and validation (20%). The resulting precision and other evaluation metrics are reported in Table 8. Additionally, we compared the performance of our model with several standard approaches, using evaluation measures such as precision, recall, F1-score, and accuracy to provide a comprehensive assessment of its effectiveness in software defect prediction.

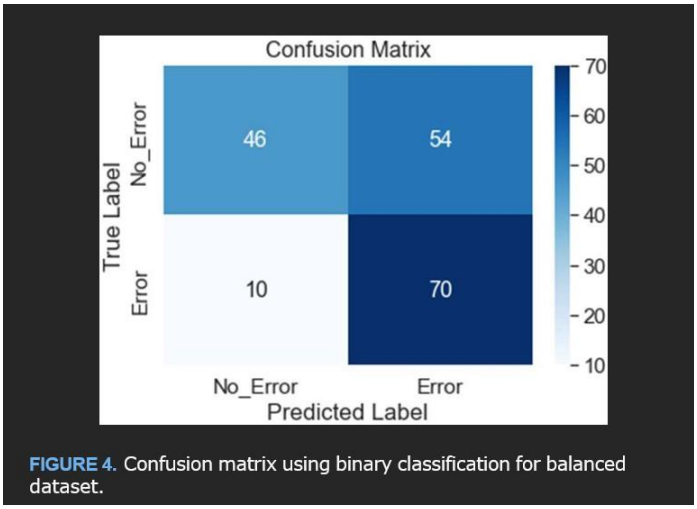


FIGURE 4. Confusion matrix using binary classification for balanced dataset.

Figure 4 illustrates the confusion matrix, which presents the performance of the proposed method in all nine classes. Although the overall accuracy does not indicate a substantial improvement in the multi-class setting, it is important to note a meaningful enhancement of approximately 7% compared to existing methods, highlighting the potential of our approach. The research findings are highly significant for several reasons:

- **Innovative Contribution** : This study presents a pioneering approach to software defect prediction. It is the first effort to create and evaluate a dataset specifically designed to predict nine different types of software defects, marking a notable advancement in the field.
- **Exploration of a Complex Area**Predicting nine distinct defects in software development is inherently complex. Achieving a 7% improvement in accuracy within such an underexplored domain highlights meaningful progress despite the challenges involved.
- **Distinctive Dataset**:The newly developed dataset, which has not been previously tested, serves as a valuable resource for analyzing and predicting multiple software defects. Its comprehensive nature enhances its potential usefulness for future research.
- This work presents a novel approach to software defect prediction. It is the first effort to create and evaluate a dataset specifically designed to predict nine different types of software defects, making it a significant contribution to the field.
- Addressing a Complex and Unexplored Problem: Predicting multiple defect types in software systems is a challenging task. Achieving a 7% increase in accuracy in such a relatively unexplored area reflects meaningful progress despite the complexity involved.
- Development of a Unique Dataset: The dataset introduced in this research has not been previously tested and provides a valuable resource for studying and predicting various software defects. Its broad coverage of defect categories offers a solid base for future investigations.
- Improvement Over Existing Methods: The proposed approach outperforms established techniques by 7%, indicating advancement in defect prediction methods. Although the gain is moderate, it demonstrates a step toward more accurate and reliable models.

Scope for Future Improvement: The promising results suggest that further refinements are possible. This research lays the groundwork for future efforts aimed at improving multi-class defect prediction systems.

V. Conclusion and Future Work

This study presents an innovative approach to software defect prediction through the use of a CodeBERT-based model, referred to as MSDP. The proposed method demonstrates strong capability in identifying defects across eight commonly occurring categories in software development, thereby contributing to more efficient and effective software testing processes. The experimental setup involved the construction of both binary and multi-class datasets, primarily utilizing code samples derived from C++ programs. The experimental results highlight the effectiveness of leveraging the pre-trained CodeBERT model to enhance both prediction accuracy and overall productivity. In the case of balanced binary classification, the proposed model achieved an approximate 20% improvement in accuracy compared to existing approaches, successfully distinguishing between 'Error' and 'No_Error' classes. Although the model showed a slight bias toward predicting 'Error' instances—likely due to the inherent complexity of error patterns—it still maintained strong overall performance.

When applied to imbalanced datasets, the model exhibited a reduction in overall accuracy; however, it continued to reliably identify both classes, demonstrating resilience to class imbalance and adaptability in challenging scenarios. For multi-class classification, the model achieved an improvement of approximately 7% in accuracy over baseline

methods, indicating moderate yet meaningful gains. The proposed methodology shows considerable potential in reducing development time while improving the quality of software systems. Furthermore, the analysis of patterns extracted from concise code snippets contributed to improved prediction outcomes, emphasizing the model's effectiveness in precise defect detection.

Overall, the findings indicate a 20% increase in accuracy for binary classification and a 7% improvement for multi-class classification. The model's capability to handle imbalanced datasets and its inclination toward detecting error-prone instances reflect the inherent complexity of software defect patterns. In practical applications, the MSDP model enhances software testing efficiency and contributes to the development of higher-quality software systems.

The proposed approach improves software quality by enabling precise defect detection and targeted remediation, which helps in optimizing resource utilization and accelerating the overall development lifecycle.

Currently, our focus is on expanding the dataset to obtain more diverse and comprehensive data, allowing for deeper analysis, improved prediction performance, and broader contributions to software engineering research. A key objective is to enhance defect prediction accuracy by introducing both generic and subclass-level defect classification.

By organizing defects into high-level generic categories and further dividing them into more specific subclasses, we can achieve a detailed understanding of the variety of defects present within a codebase. This fine-grained classification facilitates accurate identification of root causes and supports more effective corrective actions. Additionally, subclass-level categorization enables the design of specialized models tailored to specific defect types, utilizing domain-specific insights to further improve prediction accuracy.

Examining the distribution and characteristics of defects across both generic and subclass categories can provide valuable insights into frequent programming mistakes, potential areas for improving development practices, and evolving trends in software quality. Such insights support better decision-making during development, ultimately leading to more reliable defect prediction and higher-quality software systems.

To further enhance the model, we are exploring the integration of additional attention mechanisms into the existing architecture, as well as the incorporation of complementary neural network techniques to improve performance.

VI. Challenges and Limitations

One of the primary challenges in software defect prediction is the collection of sufficient instances across multiple error categories. Generating diverse code samples that cover a wide range of programming concepts, each containing specific defects, is both time-consuming and labor-intensive. Ensuring a balanced dataset with an equal number of samples for each error type adds further complexity to the data preparation process.

Another limitation is the lack of extensive research focused on multi-class defect classification, which results in a scarcity of publicly available datasets for this purpose. Consequently, significant manual effort is required for data collection, annotation, and preprocessing, increasing the overall workload.

Furthermore, the availability of advanced benchmark models for comprehensive comparison is limited. This restricts the breadth of evaluation and reduces the ability to perform in-depth comparative analysis, thereby constraining the overall assessment of the proposed approach.

References

[1] A. Kumar, A. Singh, A. K. Nayak, S. Prakash, D. Pradhan, and J. K. Mishra, "An extreme gradient boosting based automated software defect prediction model using dimensionality reduction technique," in *Computing, Communication and Intelligence*, pp. 40–43, CRC Press, 2026.

[2] R. Tamilkodi, P. S. Rani, M. D. Sirisha, G. L. Divya, K. Sagar, and A. Raghuvamsi, "Predicting software defects with an intelligent ensemble-based engine," in *2025 International Conference on Next Generation Communication & Information Processing (INCIP)*, pp. 419–424, IEEE, 2025.

[3] L.-f. Chen, K.-x. Cao, S.-p. Zhang, and Q. Dai, "Bagging gradient nearest neighbor-based ivy algorithm optimized svm ensemble learning for heterogeneous software defect prediction," *Cluster Computing*, vol. 29, no. 1, p. 72, 2026.

[4] X. Fan, J. Mao, L. Lian, Y. Li, W. Zheng, and Y. Ge, "Software defect prediction method based on stable learning," *Computers, Materials, & Continua*, vol. 78, no. 1, p. 65, 2024.

[5] A. Abdu, Z. Zhai, H. A. Abdo, *et al.*, "Semantic and traditional feature fusion using hybrid cnn-mlp," *Scientific Reports*, vol. 14, 2024.

[6] M. Ali, T. Mazhar, Y. Arif, *et al.*, "Software defect prediction using ensemble-based model," *IEEE Access*, vol. 12, pp. 20376–20395, 2024.

[7] Y. Zhang and N. Liu, "Investigation and research on several key issues of software defect prediction," *IET Software*, vol. 2025, no. 1, p. 6615496, 2025.

[8] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Programming Languages*, vol. 3, 2019.

[10] O ̈. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[11] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect prediction: Unsupervised vs supervised methods," *Information and Software Technology*, vol. 106, pp. 161–181, 2019.

[12] S. McMurray and A. H. Sodhro, "Ml-based software defect detection for smart healthcare," *Sensors*, vol. 23, no. 7, p. 3470, 2023.

[13] M. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," in *IJCAI*, pp. 1546–1553, 2017.

[14] Y. H. Choi and J. Nam, "Wine: Warning miner for improving bug finders," *Information and Software Technology*, vol. 155, 2023.

[15] D. Demirci, N. Sahin, M. Sirlancis, and C. Acarturk, "Static malware detection using stacked bilstm and gpt-2," *IEEE Access*, vol. 10, pp. 58488–58502, 2022.

[16] X. Dong, J. Wang, and Y. Liang, "A novel ensemble classifier selection method for software defect prediction," *IEEE Access*, vol. 13, pp. 25578–25597, 2025.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, pp. 1536–1547, 2020.

[18] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.

[19] B. Isong and E. Igo, "Ensemble learning for software defect prediction: Performance, practicality and future directions," *Journal of Information Systems and Informatics*, vol. 7, no. 3, pp. 2245–2291, 2025.

[20] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Information and Software Technology*, vol. 114, pp. 204–216, 2019.

[21] H. Z. Zaidi, U. Ullah, M. Arshad, H. Aljuaid, M. A. Rauf, N. Sarwar, and R. Sajid, "Machine learning approaches for software defect prediction," *Applied Computational Intelligence and Soft Computing*, vol. 2025, no. 1, p. 7933078, 2025.

[22] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Programming Languages*, vol. 2, 2018.

[23] S. Kwon, J.-I. Jang, S. Lee, D. Ryu, and J. Baik, "Codebert based software defect prediction for edge-cloud systems," in *International Conference on Web Engineering*, pp. 11–21, Springer, 2022.

[24] Z. Li, H. Zhang, Z. Jin, and G. Li, "Well: Applying bug detectors to bug localization via weakly supervised learning," 2023. arXiv:2305.17384.

[25] S. Mehta and K. S. Patnaik, "Improved prediction of software defects using ensemble machine learning techniques," *Neural Computing and Applications*, vol. 33, no. 16, pp. 10551–10562, 2021.

[26] 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," *IEEE Access*, vol. 12, pp. 22582–22594, 2024.

[27] J. K. Mishra, D. Pradhan, C. R. Sahoo, R. S. Jyothi, and L. Das, "An automated software defect prediction model using machine learning approaches," in *International Conference on Machine Learning, IoT and Big Data*, pp. 176–187, Springer, 2025.

[28] K. Shi, Y. Lu, J. Chang, and Z. Wei, "Pathpair2vec: An ast path pair-based code representation method for defect prediction," *Journal of Computer Languages*, vol. 59, 2020.

[29] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[30] J. Zhao, S. Guo, and D. Mu, "Doubigru-a: Software defect detection using attention and bigru," *Computers & Security*, vol. 111, 2021.

[31] X. Yang, L. Xiao, J. Su, and B. Huang, "Coco-gan: Codebert-driven collaborative generative adversarial learning for software defect prediction," *Software Quality Journal*, vol. 34, no. 2, p. 12, 2026.