Journal of Data Analysis and Critical Management, Volume 02, Issue 02, 2026

AI-Based Automated Software Testing Model for Predicting Bug-Prone Modules in Large Software Systems

Varun Bitkuri

Masters in Software Engineering, Stratford University, USA

Abstract

Predicting software bugs in software development life cycle is a crucial part of improving software quality. Conventional methods of testing can be slow and cannot be used in large-scale systems, necessitating intelligent and automated methods. In the current research, a hybrid model of LightGBM and GRU is suggested, which would contribute to the improvement of defect prediction. The dataset of NASA PROMISE repository JM1 is utilized, and it includes software metrics of complexity and structure of the code. Preprocessing of the dataset involves such steps as data cleaning, data outlier, feature scaling and class imbalance correction using SMOTE. LightGBM model is also useful in capturing feature interactions in structured data but the GRU model learns complex non-linear patterns. The two models are used to produce final predictions by combining the output of both models through a soft voting mechanism. Through experiments, the hybrid model is found to have high accuracy of 95.11 percent and it is superior to current models (LSTM, Random Forest). The results reveal that the suggested solution is a powerful and effective solution in predicting bug-prone modules, which finally assists to increase the reliability of the software and optimization of the testing process.

Keywords: Software Fault Prediction, Hybrid Model, LightGBM, GRU, Software Metrics, Defect Prediction.

Journal of Data Analysis and Critical Management (2026); DOI: 10.64235/w3gqwc58

Introduction

The swift development of large-scale software systems has greatly enhanced their structural and functional complexity, and hence, the software quality assurance is a major issue of concern in contemporary development principles [1]. Prompt detection of the bug-prone modules is a crucial factor in minimizing the maintenance expenses and improving system reliability [2]. The conventional methods of testing that depend on a lot of manual examination and rule-based approaches do not usually scale with enhance in size/ complexity of system [3]. As a result, a broad interest in data-based methods to be able to automate bug detection and enhance testing. The previous research has shown that machine learning models are effective in predicting software bugs [4]. Data obtained in subsequent studies has emphasized the significance of systematic evaluation structures in order to compare the methods of classification in this area [5]. As artificial intelligence (AI) grows, contemporary bug prediction models have since developed to use advanced learning algorithms that can identify the complex relationships within the software metrics. Random Forest, Support Vector machines, and Gradient Boosting- ML models have been extensively used to software bug prediction (SBP) using historical data. Moreover, deep learning methods have attracted interest because of their capability of automatically providing meaningful features of software data [6]. Other recent methods have also demonstrated that deep neural networks could enhance cross-project bug prediction accuracy and scalability to generalization [7]

Regardless of these developments, a number of issues remain unaddressed, such as class imbalance, overfitting, and lack of integration between prediction

Corresponding Author: Varun Bitkuri,Masters in Software Engineering, Stratford University, USA

How to cite this article: Bitkuri V. (2026). AI-Based Automated Software Testing Model for Predicting Bug-Prone Modules in Large Software Systems. Journal of Data Analysis and Critical Management, 02(2):1-9.

Source of support: Nil

Conflict of interest: None

models and automated testing systems [8][9]. The vast majority of the existing methods are aimed to increase the prediction accuracy and do not consider real-life implementation in automated testing conditions. According to recent studies, there is a necessity of smart structures to combine AI-based forecasting with automated tests to improve efficiency and consistency [10]. These difficulties inspired this paper to come up with an AI based automated software testing model that would be used to forecast the presence of bugs in modules of a large software system. In more recent times, DNN and multi-classifier systems have shown considerable developments of detecting bug-prone modules to allow scalable and automated testing procedures in current software systems. The strengths of this paper are as follows:

- Suggested new hybrid model LightGBM -GRU to enhance the performance of software bug prediction.
- Conducted extensive data preprocessing, such as missing values, outliers and class imbalance with SMOTE.
- Did thorough exploratory data analysis, visualization through the use of image data like class distribution, KDE plots, and correlation heatmaps.
- Performed better than existing models, and accuracy, precision, recall, and F1-score are improved.
- Tested approach on cross-validation as well as several metrics of evaluation, which ensure robustness and reliability to real world practices.

The innovativeness of the work is in the creation of a hybrid system of software bug prediction that incorporates a high-level ML model (LightGBM) and a DL model (GRU) to improve the effectiveness and resilience of the prediction. Although existing research has already covered a single MLor DL strategy, most of it is based on single-model structures or lacks efficient integration mechanisms. However, in contrast, this paper has the best of both by using a soft voting mechanism, which allows both the systematic interactions among features and the non-linear fine details of software metrics to be captured. This integration results in the best generalization, balanced performance, and more sound identification of bug-prone modules, and is thus a considerable addition to the quality assurance of any given software. The study is organized as follows: Section II describes relevant work; Section III describes suggested strategy; Section IV presents experimental outcomes as well as comparative findings; and Section V summarizes research.

Literature Review

Recent research on predicting software defects and predicting bugs is briefly reviewed in this section. The different ML and DL methods are discussed and their performance and weaknesses.

Organizations rely on software bug prediction, also known as defect prediction, to recognize software issues early in development process, when engineers are best able to pinpoint potential weak spots. A. Gupta et al. (2021) found the statistical strategy for bug prediction after comparing several methods, such as LR, NB, RF, DT, ANN etc. Performance metrics such as rec, acc, prec, and F1 are used for comparison [11].According to the theory that software faults and graph properties are related, T. Takeda et al. (2022) propose new static testing metrics for control flow graphs produced from three-address code in implementations. These metrics are based on mathematical graph analysis techniques. There is a substantial correlation between the software problems and five graph properties. Consequently, when compared to a model that relies on the conventional metrics for bug prediction complexity, lines of code (steps), and CRUD, the approach outperforms with 0.08 accuracy [12]functional improvement, and refactoring. When software changes, it is difficult to define the scope of test cases, and software testing costs tend to increase to maintain software quality. Therefore, change analysis is a challenge, and static testing is a key solution to this challenge. In this study, we propose new static testing metrics using mathematical graph analysis techniques for the control flow graph generated from the three-address code of the implementation codes based on our hypothesis of the existing correlation between the graph features and any software bugs. Five graph features are strongly correlated with the software bugs. Hence, our bug prediction model exhibits a better performance of 0.25 FN, 0.04 TN ratio, and 0.08 precision than a model based on the traditional bug prediction metrics, which are complexity, line of code (steps.

N. Srivastava et al. (2022) use a small set of ML algorithms that yield accurate results on both the training and testing datasets. This approach demonstrates the applicability of four machine learning algorithms—NN, SVM, DT, and Cubist. It determines the best outcome-based method for a bug report using the PROMISE repository's diversion dataset. SVM produced the best accuracy out of all the algorithms evaluated on the ANT dataset. This finding contributes to the literature on software bug detection by shedding light on the applicability of the aforementioned approaches to bug

Table 1: Summary of Existing Literature on Software Bug Prediction

Authors & Year	*Techniques Used*	*Dataset / Approach*	*Key Results*	*Limitations*
A. Gupta et al. (2021)[11]	LR, NB, RF, RT, ANN	Comparative analysis using standard ML models	Identified best-performing model using Accuracy, Precision, Recall, F-measure	Does not explore advanced/deep learning models
T. Takeda et al. (2022)[12]	Graph-based static testing metrics (CFG analysis)	Control Flow Graph from three-address code	Improved performance: FN=0.25, TN=0.04, Precision=0.08; strong correlation of 5 graph features with bugs	Limited comparison with broader ML models
N. Srivastava et al. (2022)[13]	Neural Network, SVM, Decision Tree, Cubist	PROMISE repository (ANT dataset)	SVM achieved highest accuracy; evaluated using R, R^2, RMSE, Accuracy	Limited dataset diversity; lacks deep learning comparison
Kiran & Ponnala (2023)[14]	Ensemble Boosting + SMOTE	Handles class imbalance problem	Accuracy: 86%, G-mean: 0.85, AUC: 0.92	Small dataset; lacks interpretability and deep learning comparison
Wei, Zhang & Ren (2023)[15]	KICL (Pre-trained LM + Contrastive Learning)	Bug severity prediction	Improved weighted F1-score by 30.68%	High computational cost; scalability concerns
Achmad & Yuhana (2024) [16]	Isolation Forest, LOF, One-Class SVM	Outlier detection approach	One-Class SVM: Accuracy 0.84, F1-score 0.91	No comparison with supervised models; scalability issues
Hadad & Capretz (2024)[17]	Deep Learning + ML (JIT + traditional models), SHAP, Integrated Gradients	Combined bug prediction approach	Accuracy: 80–86%, AUC up to 78%	Moderate AUC; limited dataset diversity

prediction. Attention, study academics and solution providers: the effort presented showcases the accuracy achieved by the current methodologies [13].

The study performed by Kiran and Ponnala (2023) is effective in using ensemble boosting methods with SMOTE class balance in software defect prediction, which obtains high results with 86% acc, 0.85 G-mean and 0.92 AUC. Nevertheless, it does not discuss the interpretability of models and generalization to different datasets. Furthermore, it depends on small datasets and is not compared to modern models of deep learning, which diminishes its overall strength [14].KICL is a grouping of pre-trained language model and knowledge-intensified and contrastive learning developed by Wei, Zhang and Ren (2023) to enhance prediction of the severity of bugs. It has a maximum improvement of 30.68 weighted F1-score over baselines. The method can however be costly in terms of computation and does not discuss the issues of scalability and real-world implementation [15].

Achmad and Yuhana (2024) used outlier detection methods to predict software defects by using the Isolation Forest, Local outlier factor and one-class SVM in predicting defects. The findings indicate that One-Class SVM has the highest acc and F1 of 0.84 and 0.91 respectively. Nevertheless, the research process is not compared to supervisedlearning models and does not consider the problem of scalability and diversity of dataset [16]. Hadad and Capretz, (2024) The authors combine the traditional and Just-in-Time predicting defects with DL and ML models. It has an accuracy of 80-86 percent and AUC of up to 78 percent, and the feature importance is conducted through SHAP and integrated gradients. Nevertheless, the moderate scores of AUCs and the lack of diversification of the dataset might influence the generalization and practical implementation [17].

In spite of the improvements made in software defect and bug prediction, there are a number of gaps. Current research pays more attention to accuracy and

ignores interpretability, scalability, and practicality. Inconsistent treatment of class imbalance and limited datasets have a negative impact on generalization of projects. Also, a complex computation requirement and fragmented frameworks underscore the necessity to have a stronger and more scalable solution as can be seen in Table I.

Methodology

The proposed methodology focuses on developing an effective software bug prediction model using a hybrid LightGBM + GRU approach. First, the JM1 data is gathered and processed, and then, data preprocessing such as data cleaning, dealing with missing values, duplication, outlier treatment, target encoding, and feature scaling through standardization is performed. Stratified sampling is then used to split the dataset into train/test sets, with SMOTE used to solve class imbalance. Afterwards, LightGBM model is trained to identify feature interactions using structured data, whereas GRU model learns non-linear patterns. Both models are combined using soft voting mechanism to produce final predictions. To assess performance of the proposed model with measures are used to ensure model's robustness and effectiveness. The methodology is presented in Fig. 1.

Data Gathering

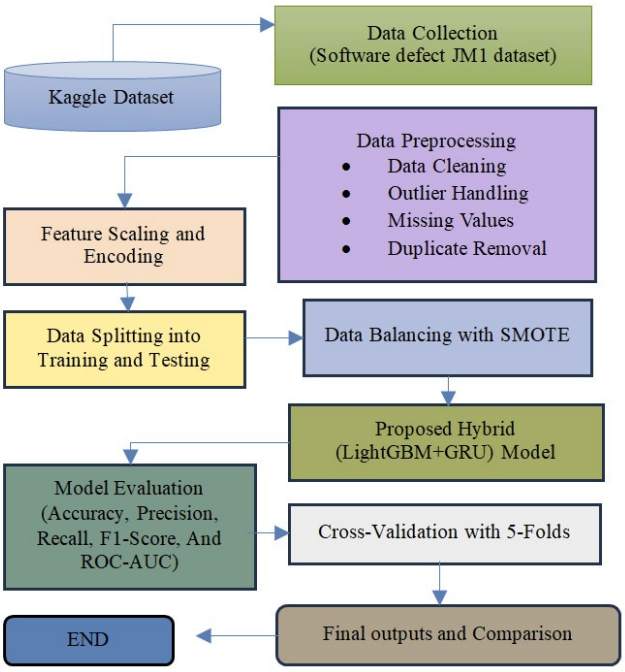

Figure 1: Proposed Flow chart of Software Bug prediction

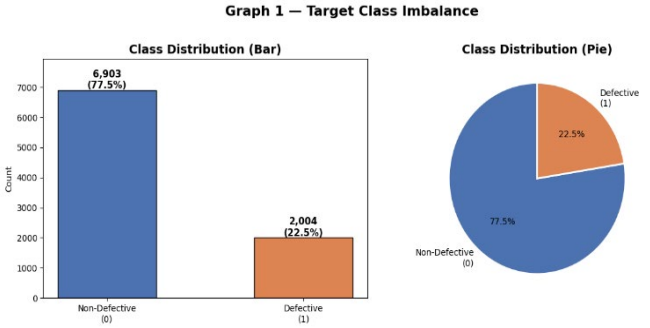

Figure 2: Class Distribution

The JM1 software defect dataset, which is publicly available on Kaggle and available in NASA PROMISE repository, was applied in this research. It has 10,885 software modules with 23 qualities, including 22 software metrics. The dataset comprises complexity, size and Halstead metrics and thus it can be used in binary classification effort. It is chosen because it is a real-world system, has an extensive size, and finds application in many studies on SBP, allowing the assessment and comparison of ML models.

Fig. 2 illustrates the bar and pie charts showing distribution of defective and non-defective modules in JM1 data. It demonstrates that, of an overall total of 8,907 instances, 6,903 modules (77.5%) are not defective, and 2,004 modules (22.5%) are defective. This implies an imbalance in classes, with most classes being the majority. This imbalance may introduce bias in the model toward non-defective predictions, necessitating the use of techniques such as SMOTE to achieve balanced learning and better prediction results.

Fig. 3 shows KDE plots of some of the features (loc, v(g), ev(g), n, totalOp, and totalOpnd). This have

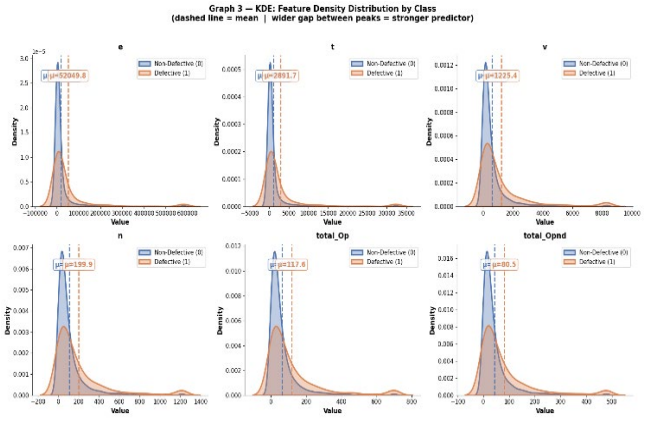

Figure 3: KDE-Based Feature Distribution Analysis by Class

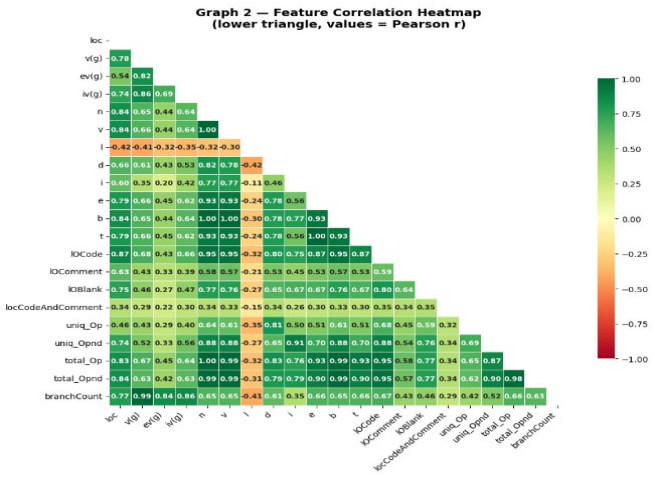

Figure 4: Feature Correlation Heatmap of numerical Columns

significant density and mean differences in terms of distribution between the two classes (as indicated by dashed lines). The defective modules tend to be more widespread and varied, which implies that these characteristics are effective in pinpointing bug-prone modules.

Fig. 4 gives the correlation heatmap of software metrics in terms of Pearson correlation coefficients. It exhibits good positive correlations among the various features (i.e., totalOp and totalOpnd) (≈ 0.99), v and n (≈ 1.00), and branch Count and v(g) (≈0.99), and thus they are very interdependent. Also, there are also features that appear to have negative correlations (e.g., loc ≈ -0.42). The analysis contributes to the elimination of redundant features and effective selection of features to minimize multicollinearity in the model.

Data Pre-Processing and Cleaning

To improve the quality of raw data and prepare it for model training, data preprocessing entails cleaning, converting, as well as preparing the data.

Initial Data Inspection

This step involves checking the dataset shape, viewing the first few rows, analyzing data types, and obtaining basic statistical and structural information to understand the data before preprocessing.

Handling Missing Values and Data Conversion

Selected string-based columns are converted into numeric format by replacing invalid values (e.g., '?') with NaN. Missing values introduced during this process are then identified and removed by dropping affected rows, resulting in a cleaned dataset with consistent data types and reduced size.

Duplicate Data Handling

Duplicate rows are identified in the dataset and removed using a conditional check. This ensures elimination of redundant data and improves reliability of dataset for model training.

Outlier Handling

The outliers of data are addressed with IQR-based winsorization method, which involves a capping on the values of features to the 1st and 99th percentiles. This minimizes the effects of extreme values but does not alter the overall data manner, resulting in more reliable model behavior.

Target Variable Encoding

The target variable defects is transformed to numerical format with False (non-defective) and True (defective) coded as 0 and 1 respectively. The data is transformed such that it is applicable to ML algorithms which take numerical input.

Normalization

StandardScaler, that employs a mean of 0 and standard deviation of 1 to ensure that features participate evenly to model while enhancing performance, is used to apply standardization. This transformation is provided in Equation (1).

$$X' = \frac{X-\mu}{\sigma} \quad (1)$$

Where X is initial feature value, is mean, and is standard deviation.

Data Splitting

Stratified sampling is employed to divide dataset into train/test set (80:20) to maintain distribution of classes to evaluate models well.

Handling Class Imbalance (SMOTE)

The Synthetic Minority Oversampling Technique (SMOTE), which creates synthetic samples of minority class, is applied to training data in order to balance class distribution. In order to give an objective assessment, the test set is simultaneously maintained. By expanding the training data, this step improve model's capacity to learn from imbalanced data.

Fig. 5 shows SMOTE, the data is skewed, with 5,522 non-defective and 1,603 defective. Using SMOTE, the classes are balanced, with 5,522 instances each. This

Figure 5: Class Distribution Before and After SMOTE

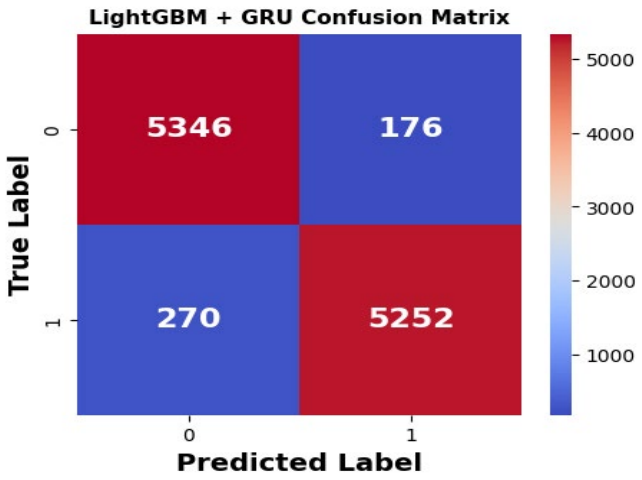

Figure 6: Confusion Matrix of Hybrid Model (LightGBM + GRU)

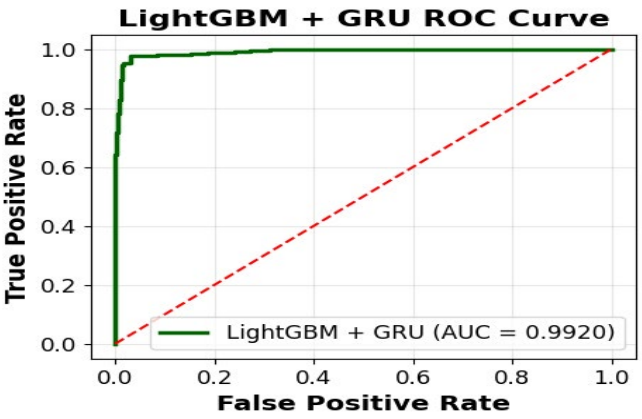

Figure 7: ROC Curve of Hybrid Model (LightGBM + GRU)

shows that SMOTE is useful in creating synthetic samples of the minority group, creating a balanced dataset that

Hybrid Model (LightGBM + GRU)

By utilizing the advantages of both ML and DL, the LightGBM + GRU hybrid model seeks to improve software fault prediction. LightGBM is efficient at handling structured tabular data and modeling feature interactions via gradient boosting, whereas GRU (Gated Recurrent Unit) detects more complex nonlinear patterns and hidden dependencies in the data. The combination of these two models in the hybrid approach increases prediction accuracy and strength.

Here, the LightGBM and GRU are both trained on the same dataset, and their forecasts are synthesized by means of a soft voting approach. It enables the model to leverage LightGBM's rapid, precise learning and GRU's ability to learn more complex patterns. Consequently, the hybrid model is superior in generalization and better performance when detecting bug-prone modules in large software systems.

Hybrid Model Training Process

The LightGBM + GRU hybrid model is trained by first splitting the dataset into train/test sets, after which LightGBM is trained on structured features using hyperparameters such as n_estimators = 100, learning_rate = 0.1, max_depth = 7, and num_leaves = 31 to learn feature interactions. In parallel, the GRU model is trained on the same dataset (reshaped for sequential input) using 64 units, 2 layers, batch size = 32, epochs = 20, and the Adam optimizer (learning rate = 0.001) to capture complex patterns. The Adam optimizer updates weights as Equation (2):

$$\theta_t = \theta_{t-1} - \frac{\eta . m_t}{\sqrt{v_t + \epsilon}} \qquad (2)$$

In the above equation, represents updated parameter at iteration t, while denotes parameter from previous iteration. The learning rate that controls step size of update is denoted by η. The mean of gradients is represented by first moment estimate, , while the variance of gradients is represented by second moment estimate, . In order to prevent division by zero and guarantee numerical stability during optimization, the term ϵ is included.

Performance Matrix

Evaluation measures assess a model's performance, including accuracy and precision. Several classifier assessment metrics have been proposed throughout the years, but accuracy, prec, rec, and F1 are among the most widely employed in Equations (3) to (6). The

percentage of accurate forecasts compared to all predictions is known as accuracy. Precision, ratio of actual positives to all expected positives, measures prediction accuracy. The ability of a model to identify positive cases is called its sensitivity or recall [18]. A suitable compromise between recall and precision may be found in an F1Score, which provides a harmonic mean of the two. The last metric is the AUROC, which measures the model's classification ability per threshold.

$$Accuracy = \frac{TP+TN}{TP+Fp+TN+FN} \quad (3)$$

$$Precision = \frac{TP}{TP+FP} \quad (4)$$

$$Recall = \frac{TP}{TP+FN} \quad (5)$$

$$F1-score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (6)$$

In this case, TP stands for the number of correct positive class forecasts, FP for false positive class forecasts, TN for accurate negative class predictions, as well as FN for false negative class forecasts.

Results and Discussion

The experiments were carried out on a machine with an Intel Core i9 processor, running Windows, with 32GB of RAM, which is sufficient for computing and training models relatively quickly. Table II gives performance analysis of hybrid model (LightGBM + GRU) on train/test data. The proposed model achieves high acc (0.9694 on training and 0.9511 on testing), indicating good generalization. Acc, rec and F1 are all high and balanced for classification. Also, the ROC-AUC score of 0.9864 has been used to indicate good discriminative ability. Low standard deviation values specify that model is stable and reliable across runs.

Table 2: Performance Evaluation of the Proposed Hybrid Model (LightGBM + GRU)

Metric	*Training*	*Testing*	*Mean*	*Std. Deviation (±)*
Accuracy	0.9694	0.9511	0.9583	0.0007
Precision	0.9682	0.9676	0.9776	0.0005
Recall	0.9657	0.9509	0.9365	0.0016
F1 Score	0.9586	0.9562	0.9575	0.0008
ROC AUC	0.9983	0.9920	0.9864	0.0006

Table 3: Five-Fold Cross-Validation Hybrid Model Performance

Fold	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*ROC AUC*
Fold 1	0.9586	0.9778	0.9381	0.9572	0.9879
Fold 2	0.9601	0.9786	0.9412	0.9595	0.9887
Fold 3	0.9598	0.9784	0.9405	0.9589	0.9885
Fold 4	0.9589	0.9779	0.9379	0.9576	0.9878
Fold 5	0.9600	0.9793	0.9410	0.9600	0.9886

Table 4: Exisinf vs proposed (LightGBM + GRU) model comparison for bug prediction

Metric	*LightGBM + GRU*	*LSTM[19]*	*RF[20]*
Accuracy	0.9511	0.8746	0.9300
Precision	0.9676	0.6970	0.8900
Recall	0.9509	0.5656	0.5700
F1 Score	0.9562	0.6240	0.6900

The performance analysis of the suggested LightGBM + GRU hybrid model in Figs. 6 and 7. Fig. 6 illustrates that the model has good and balanced classification with 5346 TN and 5252 TP and the remaining 176 FP and 270 FN are low and thus the model is correct in its classification. Fig. 7 shows the ROC curve that tends towards the top-left corner and with AUC stands at 0.9920 which shows good discriminative power. In general, these findings endorse efficiency and strength of hybrid model that is being proposed towards prediction of software bugs.

Cross-Validation Analysis

The outcomes of hybrid model (LightGBM + GRU) are assessed on principle of five-fold cross-validation, which is provided in Table III. The performance is very high in all of the folds with the accuracy values of between 0.9586 and 0.9601. All the folds have precision values that are greater than 0.977 which implies that the model can correctly classify non-defective and defective modules. Similarly, the values of recall are consistent ranging within 0.938 and 0.941 showing the successful identification of bug-prone modules.

Moreover, the ROC-AUC values range from 0.9878 to 0.9887, indicating that hybrid model has excellent performance capability. The minimal variation in performance measures across folds demonstrates the model's stability and generalization capacity. In general, the cross-validation findings validate the fact that the suggested hybrid model is reliable and consistent, and it is applicable to real-world software bug prediction tasks.

Comparison and Discussion

Table IV represents assessment of hybrid model (LightGBM + GRU) with existing models, including LSTM,

and RF to predict software bugs. The proposed model presents highest performance of all metrics, maximum acc (0.9511), highest prec (0.9676), maximum rec (0.9509), and highest F1(0.9562), which means that model has higher and balanced level of predicting. Comparatively, LSTM model exhibits much lesser performance, particularly in rec (0.5656) and F1 (0.6240), whereas RF model have moderate performance yet outperform the proposed solution. The results obtained support the efficiency of the hybrid model in enhancing the accuracy and reliability of bug prediction.

The proposed LightGBM + GRU hybrid model has several strengths, including high predictive accuracy, strong generalization, and balanced performance across prec, rec, and F1, making it effective for detecting bug-prone modules in software engineering. ML and DL are combined to ensure that the model can learn both linear interactions between features and non-linear and complex interactions. The model, however, has some drawbacks, including computational and training complexity due to its hybrid architecture, and potential interpretability issues compared to less complex models. Moreover, performance may change when applied to other datasets or domains. Notwithstanding these shortcomings, the research study has significant implications, in that it offers a sound and scalable method to enhance software quality, test optimization and automated bug forecasting in the actual software development environment.

Conclusion and Future Scope

This study offerings an effective AI-based hybrid model that combines LightGBM and GRU to predict bug-prone modules in large software systems. The suggested solution leverages the strengths of ML and DL to extract structured feature interactions and intricate patterns, resulting in stronger predictive performance. The investigational outcomes reveal that hybrid model achieves high accuracy (95.11%) and strong precision, recall and F1-score, outperforming existing models such as LSTM and RF. The model's applicability and reliability are further supported by real-world software metrics, which demonstrate its usefulness and effectiveness in practice. On the whole, the suggested model will help improve software quality by enabling early bug identification and facilitating effective testing strategies. To improve the model's performance, it is possible to incorporate more complex deep learning architectures, such as Transformers, in future applications. Moreover, to consider using explainable AI methods, like SHAP or LIME, to increase model interpretability and gain a more insightful understanding of feature importance. The method may also be tested on multiple, cross-project datasets to enhance generalization. Moreover, the model can be applied to real-time deployment in continuous integration pipelines and optimized for use in industrial contexts.

References

S. P. Kalava, "Enhancing Software Development with AI-Driven Code Reviews," *North Am. J. Eng. Res.*, vol. 5, no. 2, pp. 1–7, 2024.

S. K. Chintagunta and S. Amrale, "AI in Code, Testing, and Deployment: A Survey on Productivity Enhancement in Modern Software Engineering," *Int. J. Curr. Eng. Technol.*, vol. 13, no. 6, pp. 1–8, 2023.

P. Chandrashekar and M. Kari, "A Study on Artificial Intelligence in Software Engineering with Methodologies, Applications, and Effects on SDLC," *TIJER – Int. Res. J.*, vol. 11, no. 12, pp. 932–937, 2024.

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008, doi: 10.1109/TSE.2008.35.

R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, Feb. 2015, doi: 10.1016/j.asoc.2014.11.023.

S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2016, pp. 297–308. doi: 10.1145/2884781.2884804.

I. Velásquez, A. Caro, and A. Rodríguez, "Kontun: A Framework for recommendation of authentication schemes and methods," *Inf. Softw. Technol.*, vol. 96, pp. 27–37, Apr. 2018, doi: 10.1016/j.infsof.2017.11.004.

S. S. S. Neeli, "Key Challenges and Strategies in Managing Databases for Data Science and Machine Learning," *Int. J. Lead. Res. Publ.*, vol. 2, no. 3, 2021.

R. Patel and P. B. Patel, "The Role of Simulation & Engineering Software in Optimizing Mechanical System Performance," *Tech. Int. J. Eng. Res.*, vol. 11, no. 6, pp. 991–996, 2024, doi: 10.56975/tijer.v11i6.158468.

S. Ajorloo, A. Jamarani, M. Kashfi, M. Haghi Kashani, and A. Najafizadeh, "A systematic review of machine learning methods in software testing," *Appl. Soft Comput.*, vol. 162, p. 111805, Sep. 2024, doi: 10.1016/j.asoc.2024.111805.

A. Gupta, R. K. Shukla, A. Bhola, and A. S. Sengar, "Comparative Analysis of Supervised Learning Techniques of Machine Learning for Software Defect Prediction," in *Proceedings of the 2021 10th International Conference on System Modeling and Advancement in Research Trends, SMART*

2021, 2021. doi: 10.1109/SMART52563.2021.9676307.

T. Takeda, S. Masuda, and K. Tsuda, "Software Bug Prediction Model Based on Mathematical Graph Features Metrics," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, Apr. 2022, pp. 229–235. doi: 10.1109/ICSTW55395.2022.00047.

N. Srivastava, M. Agarwal, S. Arora, and T. Lamba, "OPABP-Optimizing Parameters, to Improve Accuracy in Bug Prediction using Machine Learning," in *2022 4th International Conference on Artificial Intelligence and Speech Technology (AIST)*, IEEE, Dec. 2022, pp. 1–6. doi: 10.1109/AIST55798.2022.10064852.

D. S. Kiran and R. Ponnala, "Ensemble Boosting Algorithms for Software Defect Prediction," in *2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*, 2023, pp. 321–326. doi: 10.1109/ICAICCIT60255.2023.10466047.

Y. Wei, C. Zhang, and T. Ren, "Improving Bug Severity Prediction With Domain-Specific Representation Learning," *IEEE Access*, vol. 11, pp. 62829–62839, 2023, doi: 10.1109/ACCESS.2023.3279205.

R. M. Achmad and U. L. Yuhana, "Software Defect Prediction using Outlier Detection Algorithm," in *2024 2nd International Conference on Software Engineering and Information Technology (ICoSEIT)*, 2024, pp. 204–209. doi: 10.1109/ICoSEIT60086.2024.10497461.

Nadella, N. V. M. (2025a). Evolution of fault management in telecommunications: From reactive response to AI-driven predictive analytics. World Journal of Advanced Engineering Technology and Sciences, 15(2), 1338– 1344. https://doi.org/10.30574/wjaets.2025.15.2.0671

S. Haldar and L. F. Capretz, "Feature Importance in the Context of Traditional and Just-In-Time Software Defect Prediction Models," in *2024 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2024, pp. 818–822. doi: 10.1109/CCECE59415.2024.10667167.

Nadella, V. M. (2024). AI-Native 6G Network Management. *American International Journal of Computer Science and Technology*, *6*(1), 23-37.

R. Chennappan and Vidyaathulasiraman, "An automated software failure prediction technique using hybrid machine learning algorithms," *J. Eng. Res.*, vol. 11, no. 1, p. 100002, Mar. 2023, doi: 10.1016/j.jer.2023.100002.

Nadella, V. M. (2025). AI/ML-Driven Service Assurance: 2024 breakthroughs transforming telecom operations. *Journal of Computer Science and Technology Studies*, *7*(4), 01-07.

W. Albattah and M. Alzahrani, "Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach," *AI*, vol. 5, no. 4, pp. 1743–1758, Sep. 2024, doi: 10.3390/ai5040086.

N. B. Revanasiddappa, "AI-powered quality assurance : Enhancing software infrastructure through intelligent fault detection," *World J. Adv. Res. Rev.*, vol. 23, no. 03, pp. 3199–3213, 2024.

Sannapureddy, R., Nadella, V. M., & Nelavelli, S. (2024). Edge–Cloud Continuums for Latency-Sensitive Tasks. *International Journal of AI, BigData, Computational and Management Studies*, *5*(4), 189-201.

Madhav Nadella, V. (2025). under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 (CC BY-NC-ND 4.0) International License Real-Time Monitoring Systems (5G/6G & Beyond): A Technical Review. Sarcouncil Journal of Engineering and Computer Sciences, 4(08).