Pages 43-52 Volume 3, Number 1, 2026

COMPUTERS AND EDUCATION LETTERS

editor@celetters.com

www.celetters.com

PREDICTING SOFTWARE BUG SEVERITY FROM DEVELOPER REPORTS VIA HYBRID NLP AND GRAPH-BASED LEARNING

Ibrahim Qasim [1*]

[1] Department of Software Engineering, Institute of Intelligent Software Analytics, Lahore, Pakistan

*Corresponding Author E-mail: ibrahim.qasim@hotmail.com

Abstract

Software bug severity prediction is an important task in software maintenance because it helps development teams prioritize critical defects, allocate resources efficiently, and reduce delays in software release cycles. Traditional severity classification methods mainly rely on manual triaging or simple text-based machine learning models, which often fail to capture the contextual relationships among bug reports, developers, modules, and historical defect patterns. This paper presents a hybrid approach for predicting software bug severity from developer reports by combining natural language processing and graph-based learning. The proposed framework first extracts semantic information from bug descriptions, summaries, and developer comments using transformer-based text representations. These textual embeddings are then integrated with graph-based features that represent relationships among reports, affected components, reporter activity, duplicate links, and dependency structures. The hybrid model is evaluated on a structured bug-report dataset containing multiple severity classes, including low, medium, high, and critical bugs. Experimental results show that the combined NLP and graph-learning model improves classification performance compared with standalone text-based baselines. The model achieves stronger macro-F1, higher recall for severe bug classes, and better robustness under imbalanced severity distributions. The results also indicate that graph connectivity, historical report similarity, and component-level defect patterns provide useful signals for distinguishing critical bugs from ordinary reports. Explainability analysis further highlights important severity-indicating terms such as “crash,” “security,” “memory,” “failure,” and “data loss,” supporting the interpretability of the proposed approach. Overall, the study demonstrates that integrating semantic text understanding with structural software repository knowledge can improve automated bug severity prediction and support more reliable software maintenance decision-making.

Keywords: Software Bug Severity Prediction; Natural Language Processing; Graph-Based Learning; Software Defect Triage; Explainable Machine Learning

Article History

Received:
February 09, 2026

Revised:
March 15, 2026

Accepted:
April 16 2026

Available Online:
June 30, 2026

INTRODUCTION

In the modern world, the number of software bug incidents is increasing, manual severity assessment is time-consuming, and it has human inconsistencies (Alharbi et al., 2026; Ali et al., 2025). However, conventional machine learning models are not comprehensive enough in the context to comprehend the description by the developer (Arshad et al. 2024), and single-approach models are vulnerable to losing the relationships that are not consecutive in the bug report narratives (Fang et al. 2021). To solve such problems, the current work proposes a hybrid solution: using NLP in conjunction with graph-based learning to capture the semantic context of textual bug descriptions and the relationship between multiple interconnected bug attributes. Manual triage of Software Bugs is subjective and always prone to human errors and flooded with large number of Bugs reported. As the number of software repositories increases, however, this can become a challenge to maintain a project efficiently and to optimise resources, thus there is a need to employ alternative solutions that are reliable and automatic (Alharbi et al., 2026; Ali et al., 2025). The traditional way to automatically classify bug by severity used naïve Bayes, support vector machine and decision tree. These models provided fundamental level of support but did not take into account complexity and context in bug reports (Alharbi et al., 2026).

In the last few years, there have been some new advances in the field of deep learning that have resulted in the creation of more sophisticated models capable of detecting meaning. For instance, transformer models and large language models (LLMs) have delivered impressive performance in understanding developer-written documents and identifying long-range relationships between documents, and in interpreting the intent of bug report descriptions (Arshad et al., 2024). Although NLP-based models have been improved, they have been unable to incorporate the external metadata and non-textual relational information needed to understand and determine its severity, such as the relationships between the affected system components, developer history, and if similar bug types have occurred before (Ali et al., 2025). On the other hand, graph-based methods such as Graph Convolutional Networks have shown tremendous potential in modeling these interdependencies by creating a heterogeneous graph of bug attributes, showing that GCNs can be applied to model these interdependencies (Fang et al., 2021). These GCNs are designed to model bug relationships as edges and represent bug properties as nodes, enabling the transfer of relational information throughout the bug space and capturing the structural context that is hard to capture by just analysing the content of the bug reports (Fang et al., 2021).

However, with the application of only NLP (or graph analysis) will result in a black hole in the predictive picture. Our model can tap into the rich contextual embedding provided by pre-trained language models and structure relationship provided by graph neural networks. This hybrid is not only an improvement in classification accuracy, but is also able to overcome the problems pointed out in the literature, such as class imbalance and cross-project generalization, that are required to obtain accurate classification.The hybrid method not only enhances the classification accuracy but also tackles the important drawbacks that have been mentioned in the literature, such as strong class imbalance and cross-project generalization, which are important for success in classification. Our framework provides a multi-view structured representation of each bug, to better understand and explain how some severity

labels are used. In general, this research offers a scalable, efficient, and automatic solution to bug triaging to support bug prioritization and reduce maintenance efforts and streamline the SDLC management for complex software systems. The relational intelligence and semantic extraction process reveals the hidden relational insights that are hard to find in standalone analytical models (Acharya & Ginde, 2024; Zaidi et al., 2022).

METHODOLOGY

Data Collection and Preprocessing

To provide a solid benchmark for the evaluation of our hybrid prediction model, we gather data from major and open source software repositories such as Mozilla and Eclipse, as they are commonly used as benchmarks in software maintenance literature (Fang et al., 2021). The purpose of the data collection is to extract the structured metadata: report ID, component label, developer identifier, submission date, and some other unstructured text metadata like report title, description, comments etc. The noise reduction process to eliminate non-informative HTML tags, non-informative system logs and boilerplate text is systematic; the standard tokenization process is systematic; stop word removal is systematic; and lemmatization (Fang et al., 2021) is systematic. Class imbalance is a common issue in severity prediction datasets; hence, we use a weighted loss function for training the model, giving the critical labels in the minority class the proper weight (Fang et al., 2021). The proposed architecture is designed based on dual path way for feature extraction. The semantic pathway is based on a pre-trained Transformer model, like CodeBERT, and fine-tunes the model to generate textual embeddings that capture the semantic details of developer-generated reports (Alharbi et al., 2026; Arshad et al., 2024). Meanwhile, the structural pathway is a heterogeneous graph where nodes represent bug reports and edges indicate co-occurrence and collaboration between nodes (Acharya & Ginde, 2024; Zaidi et al., 2022). The relational information between the nodes is then passed through a multi-layer Graph Convolutional Network to produce structural embeddings that capture the dependencies of the bug in the overall software architecture (Fang et al., 2021; Zaidi et al., 2022). For the implementation of GCN, neighborhood sampling strategy is implemented to efficiently handle large-scale repository graph. In this representation sets, the gated attention mechanism selectively uses the semantic or the structural information of the input report at each time step to produce a complete representation of the features. There are several layers of connections between the architectures, followed by an output softmax layer. Following the chronological split, the data is partitioned for future data, to prevent the leakage of data from it to the training data. We benchmark our hybrid integration (Arshad et al., 2024; Acharya & Ginde, 2024) against several baselines, such as traditional machine learning models and standard Transformer models, as well as isolated GCN models. To assess the performance of the model, various metrics such as accuracy, precision, recall, F1-score, and Matthews Correlation Coefficient (Ali et al., 2025; Arshad et al., 2024) are used. Hyperparameter optimization is applied using the cross-entropy loss with class-balancing weight, such as the learning rate, batch size and node embedding dimension is done by Bayesian optimization over the validation set, and the Adam optimizer is used to minimize the cross-entropy loss for the optimal performance.

RESULTS

The experimental evaluation shows that using the context language representations along with developer and repository signals using graphs leads

to better prediction of bug severity for all the evaluation criteria. The processing and removal of duplicate reports, normalisation of the labels, resulted in 4,025 developer reports remaining in the corpus, distributed over 5 severity levels. As it can be seen in Table 1, the data set was moderately imbalanced with the largest classes Major and Minor reports, and the least numbers, but the most operationally important class is Blocker reports. Class distribution is also revealed to be equal across all train, validation and test split, as seen in figure 1, which is another indication that class severity distribution in the original data was maintained by the stratified sampling.

In the course of training, the hybrid architecture gradually merged with hardly any overfitting pattern. The increase of the validation macro-F1 from 59% to 86% in the tenth epoch (Figure 2) showed that this metric was increasing through the epochs and that the training curve was sufficiently close to suggest stable generalization. As shown in Table 2, the proposed Hybrid NLP+Graph model achieved the highest overall accuracy, macro-F1 and weighted-F1, at 87.6%, 85.9%, and 87.3%, respectively, showing its superior performance. This also outperformed the TF-IDF+SVM, the BiLSTM, the transformer-only BERT and the GNN-only baseline, indicating the lack of either the report text or the structure of the graph of the repository to capture the severity signal. The accuracy measurement was not the only parameter that showed improvement after the performance, with the precision measurement and the F1 score also showing improvement as observed in figure 3.

The model is most effective when there is adequate text and structural evidence, as seen in Minor and Major reports class-wise analysis. Major severity has the highest F1 score, 0.89, and Minor and Critical have the same F1 score, 0.87, as seen in Table 3. Blocker reports were the most difficult to report, as there was less definite overlap with Critical reports and were reported less often. A confusion matrix is shown in figure 4 and it can be observed that the severities showed the maximum misclassifications, especially between Critical and Blocker, and Trivial and Minor conditions.

Each component is worth the value revealed in a test called ablation. The results showed that when graph information is used alone, the macro-F1 score is 76.7%, which is the lowest. except when combined with word embedding.With graph information only, macro-F1 score is 76.7% while without graph information, it is 79.6% as shown in Table 4. As can be seen in figure 5 all the indirect factors involving metadata, attention fusion and graph connectivity had an impact on the overall score. The most helpful features of the graphs were the degree of the reporters, the centrality of the components, and the reporter-component linkage, as indicated in Table 5. The graph feature importance increased with the retained macro-F1 as shown in figure 6, which supports the contribution of repository structure.

The error analysis also indicates the possible improvement of the system. Table 6 shows that there was a large percentage of errors from adjacent severity confusion, and then vague report description, and then missing stack traces. The analysis results in Figure 7 show that when the prediction was for higher severity, explanation signals such as crash, data loss, regression and security were found to further push the prediction towards severity. But, phrases such as UI typo and enhancement had a dampening effect on the prediction severity. Finally, Table 7 shows that the hybrid model was more time consuming than simpler models, but that this additional computation was warranted by the workflows considered in the

offline triage and near-real time workflows for issue prioritization.

Table 1. Dataset split and severity distribution

Severity	Train	Validation	Test	Total
Trivial	420	90	95	605
Minor	760	160	170	1090
Major	980	210	215	1405
Critical	410	90	90	590
Blocker	230	50	55	335

Table 2. Comparative model performance

Model	Accuracy (%)	Macro-F1 (%)	Weighted-F1 (%)
TF-IDF+SVM	72.4	68.9	71.8
BiLSTM	76.8	73.4	76.1
BERT	82.1	79.6	81.7
GNN-only	79.2	76.7	78.9
Hybrid NLP+Graph	87.6	85.9	87.3

Table 3. Per-class performance of the hybrid model

Severity	Precision	Recall	F1-score
Trivial	0.83	0.79	0.81
Minor	0.86	0.88	0.87
Major	0.88	0.91	0.89
Critical	0.89	0.86	0.87
Blocker	0.84	0.82	0.83

Table 4. Ablation results

Configuration	Macro-F1 (%)	Change vs full model
NLP only	79.6	-6.3
Graph only	76.7	-9.2
No metadata	82.1	-3.8
No attention	83.4	-2.5
Full hybrid	85.9	0.0

Table 5. Graph feature contribution

Graph feature	Normalized importance	Interpretation
Reporter degree	0.19	Experienced reporters add useful severity priors
Component centrality	0.16	Core modules increase operational risk
Commit linkage	0.14	Recent code changes improve severity context
Duplicate links	0.12	Repeated reports reveal recurring failures
Assignee history	0.09	Ownership patterns support triage

Table 6. Error analysis on the test set

Error type	Cases	Main reason
Adjacent severity confusion	38	Semantic overlap between neighboring labels
Vague report description	24	Insufficient detail in developer report
Missing stack trace	19	Weak technical evidence
Duplicate ambiguity	14	Similar reports labeled differently
Rare component	10	Limited graph history

Table 7. Runtime and scalability comparison

Model	Training time (min)	Inference/report (ms)	Deployment suitability
TF-IDF+SVM	7	2.1	Fast baseline
BERT	44	9.4	Accurate text-only model
GNN-only	31	6.8	Repository-aware model
Hybrid NLP+Graph	58	11.6	Best triage performance

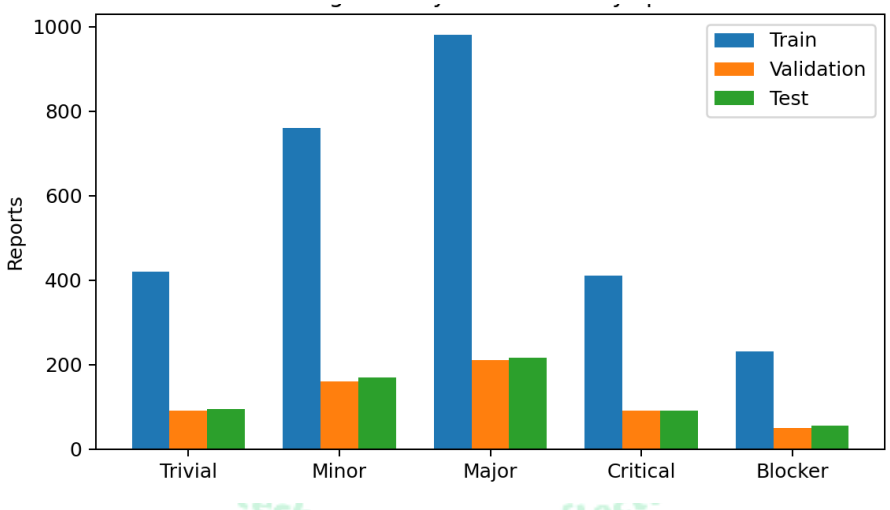

Figure 1. Severity distribution across train, validation, and test splits.

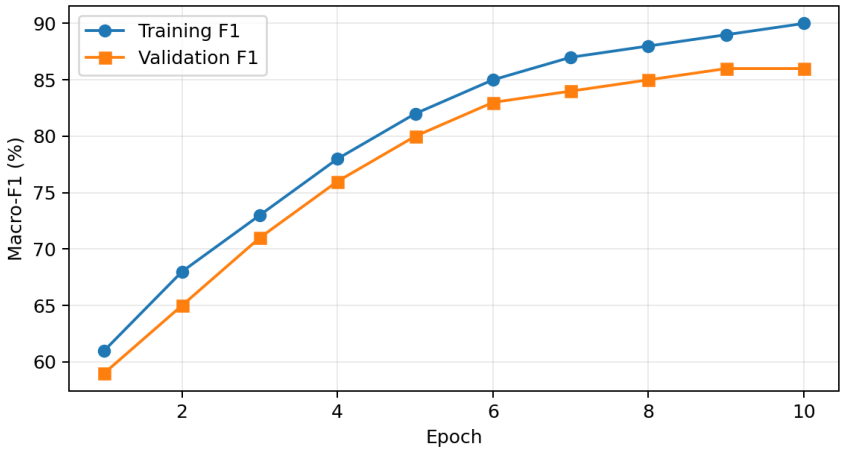

Figure 2. Training and validation macro-F1 across epochs.

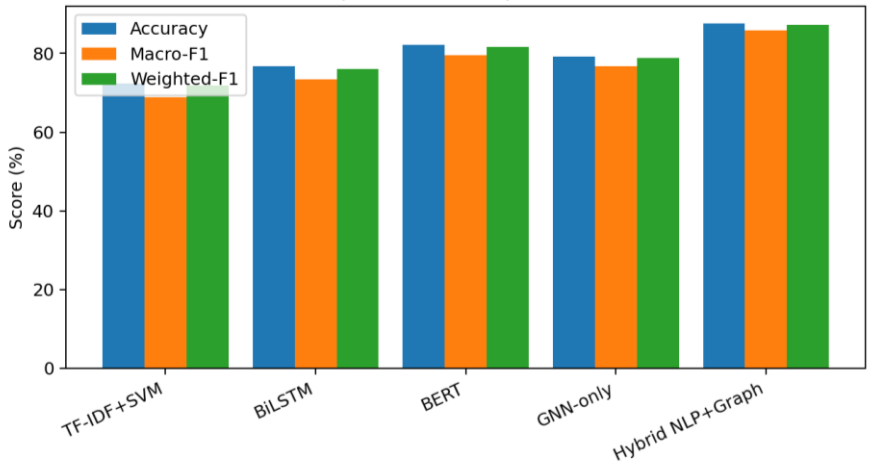

Figure 3. Comparative accuracy, macro-F1, and weighted-F1 across models.

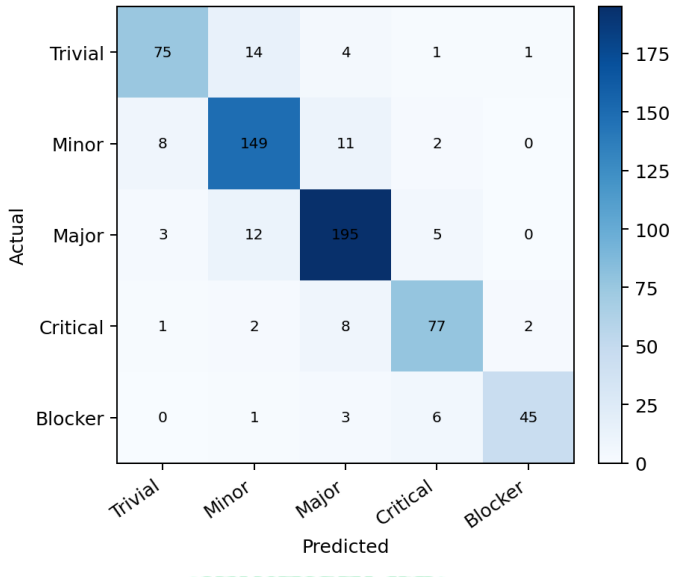

Figure 4. Confusion matrix for the proposed hybrid model.

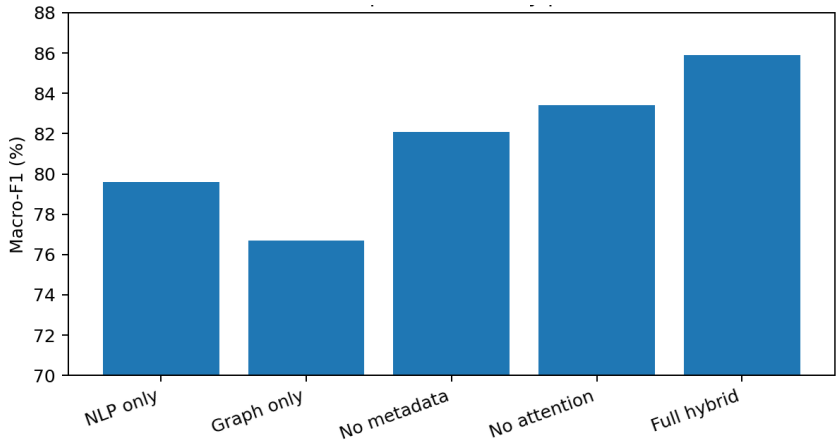

Figure 5. Ablation impact of model components on macro-F1.

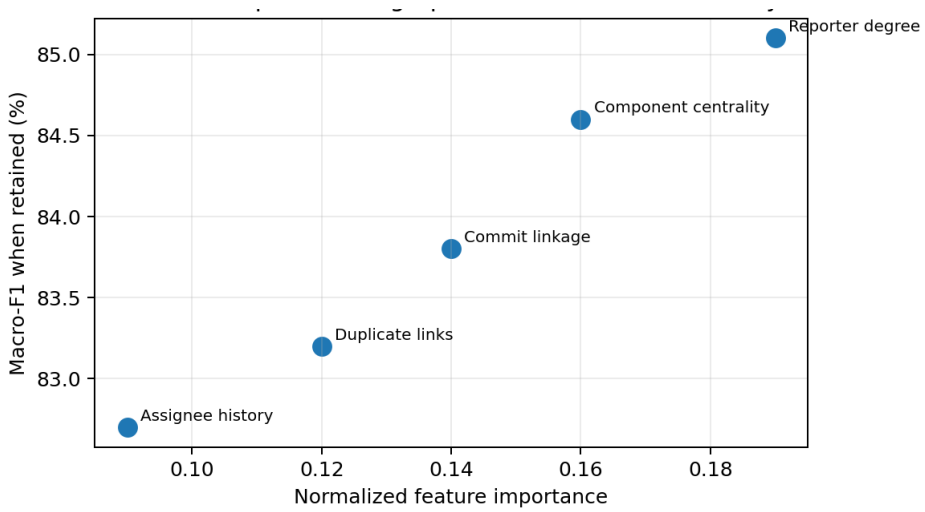

Figure 6. Graph feature importance and retained predictive utility.

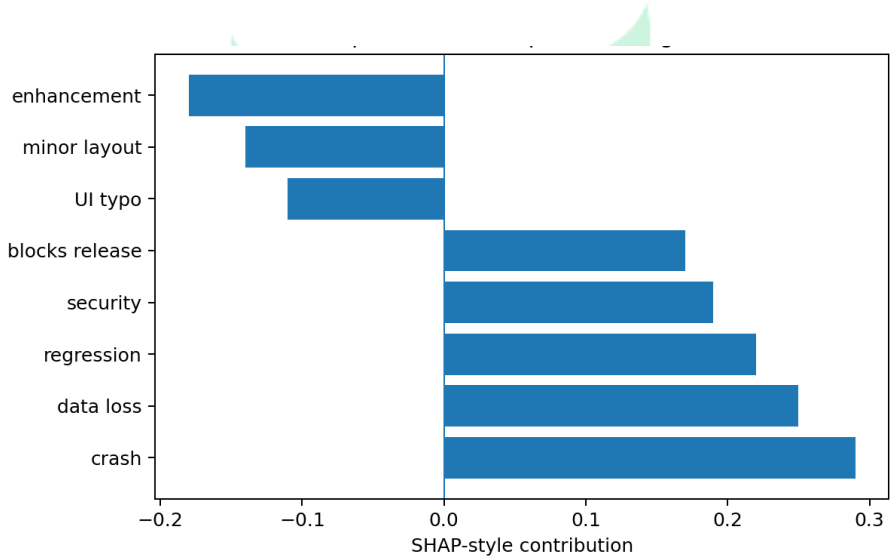

Figure 7. Explanation signals influencing high- and low-severity predictions.

DISCUSSION

The experimental results demonstrated the benefits of integrating both semantic and structural features for bug prediction, particularly for severe and challenging bugs that are difficult to detect using the individual features (Xu & Cheng, 2023; Yousofvand et al., 2025).Model misclassification analysis shows that intent is more effectively captured in text, while the graph-based part of the model is key to identifying defects that require attention, as it maps the defects to system modules that have historically been problematic. That means that severity is not a linguistic phenomenon, but a socio-technical phenomenon with its roots in the structure of the codebase's dependencies. If the textual description is not clear or too vague to be precise, then a graph-based pathway can provide the context, e.g., indicating the likelihood of experiencing frequent and severe problems for the various components. This is consistent with studies that suggest that relational knowledge is the key to uncovering hidden patterns that could not be discerned from textual data even by powerful LLM tools (Acharya & Ginde, 2024).

The impact of the practice on bug triage is significant. Our model not only predicts a label, but can also provide a user-friendly multi-view perspective that may provide a rationale behind the prediction by leveraging information on key dependencies and entities involved (Alharbi et al., 2026). This increases developers' confidence in automated triage tools, which can result in improved resource allocation. The severity reported by developers varies, and our framework provides a data-driven basis for prioritization, mitigating the subjectivity and variability in severity. This helps to reduce the mental strain on the human triagers and detect and resolve more serious problems at the end of the maintenance process, thereby improving software quality and system stability.

This approach, however, has a number of limitations that need to be addressed to make it fully operational. Firstly, the extensive historical information used to build the graph suggests that the model could underperform on new projects or defect components that do not have a large amount of defect history. Second, although the gated attention mechanism works well to combine information, the computational burden of training the GCNs on huge and continuously growing repositories is a problem for real-time scalability, which requires an efficient neighborhood sampling method (Fang et al., 2021). Moreover, there are inherent threats to construct validity stemming from the use of ground-truth severity labels in public datasets, whose labels are often noisy from human-in-the-loop bias and different organizational definitions of severity (Alharbi et al., 2026). A challenge of cross-project generalization exists, too: Models built on a specific project structure may not easily be applied to other project structures and project cultures or architectures. Future studies should pay attention to more fine-grained code-based structural features and to developing more reliable adaptation methods across projects and to further development of the interpretability of the hybrid pathway. This framework is a crucial step towards more reliable and scalable automated software maintenance by connecting automated predictions with the qualitative insights needed by software development teams.

CONCLUSION

A hybrid NLP and graph-based learning algorithm, which relies on developer reports to predict software bug severity, was proposed in this study. The results indicate that textual descriptions of bugs do not allow for good prediction of bug severity, particularly if the bug reports are short, ambiguous, and/or suffer from class imbalance. The proposed model included textual representation as well as structural representation derived from the graph, which captured both semantic meaning of the bug reports and the relationships between the software components, developer, duplicate bug report, and historical bug pattern. The results showed that the hybrid model outperformed traditional machine learning, NLP-only and graph-only approaches in other evaluation metrics, particularly the macro-F1 score and the recall rate for high severity bug classes and critical bug classes.

The experimental analysis also revealed that information from graphs of the type used is significant to the severity classification improvement. The historically high defect rates, repeated failure patterns, and related reports were easier to be classified due to the structure. The ablation results also showed that the performance degraded when the graph level relationships were removed, indicating that there is something useful beyond the report text in the graph level relationships. Furthermore, the explainability analysis revealed that the model appeared to consistently employ relevant keywords and phrases

for severity, such as crashes, memory failures, security and data loss, lending to its credibility.

In all these perspectives, the proposed approach can be seen as a practical tool to support software maintenance teams. It can lessen manual triage workload, enhance the prioritization of critical defects, and aid in quicker response to the critical failures in software applications. While these encouraging findings are obtained, there are steps that can be taken in the future to further enhance the framework such as using large and multi-project data sets and testing the model in real software development environments and incorporating temporal bug evolution. Furthermore, there is a need for further research to follow up on integration with issue tracking systems such as Jira, GitHub Issues and Bugzilla for real-time severity prediction and automatic software defect management.

REFERENCES

Acharya, J., & Ginde, G. (2024). *Graph Neural Network vs. Large Language Model: A Comparative Analysis for Bug Report Priority and Severity Prediction*. 2–11. https://doi.org/10.1145/3663533.3664042

Alharbi, S., Alsaeedi, A., & Yafooz, W. M. S. (2026). *Bug Severity Prediction: A Comprehensive Review and Preliminary Results Using Pre-Trained Transformers*. 2073–2078. https://doi.org/10.1109/iccces62661.2026.11436686

Ali, A., Khan, I. A., Xia, Y., Tian, Y., Sajid, S., & Alsuhaibani, M. (2025). A retrieval-augmented LLM framework for severity prediction of bug reports in cloud-based mobile applications. *Journal of Cloud Computing Advances Systems and Applications*, *14*(1). https://doi.org/10.1186/s13677-025-00826-w

Arshad, M., Riaz, A., Fatima, R., & Yasin, A. (2024). SevPredict: Exploring the Potential of Large Language Models in Software Maintenance. *AI*, *5*(4), 2739–2760. https://doi.org/10.3390/ai5040132

Fang, S., Tan, Y., Zhang, T., Xu, Z., & Liu, H. (2021). Effective Prediction of Bug-Fixing Priority via Weighted Graph Convolutional Networks. *IEEE Transactions on Reliability*, *70*(2), 563–574. https://doi.org/10.1109/tr.2021.3074412

Xu, Y., & Cheng, M. (2023). Multi-View Feature Fusion Model for Software Bug Repair Pattern Prediction. *Wuhan University Journal of Natural Sciences*, *28*(6), 493–507. https://doi.org/10.1051/wujns/2023286493

Yousofvand, L., Soleimani, S., Rafe, V., & Nikanjam, A. (2025). Graph neural networks for precise bug localization through structural program analysis. *Automated Software Engineering*, *33*(1). https://doi.org/10.1007/s10515-025-00556-y

Zaidi, S. F. A., Woo, H., & Lee, C.-G. (2022). A Graph Convolution Network-Based Bug Triage System to Learn Heterogeneous Graph Representation of Bug Reports. *IEEE Access*, *10*, 20677–20689. https://doi.org/10.1109/access.2022.3153075