IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁙ Peer-reviewed & Refereed journal ⁙ Vol. 15, Issue 2, February 2026

DOI: 10.17148/IJARCCE.2026.15239

Automated Static Code Analysis and Defect Prediction Using Large Language Models and Program Representation Techniques

Dr. T. Subba Reddy[1], S. Bhuvaneswari[2], O. Sravani[3], N. Amulya[4], N. Jyosthna[5]

Associate Professor, Dept. of CSE–Data Science KKR & KSR Institute Of Technology and Sciences, Guntur[1]

B.Tech Student, Dept. of CSE–Data Science KKR & KSR Institute Of Technology and Sciences, Guntur[2,3,4,5]

Abstract: Imperfections in software lead to critical problems in software reliability and stability. Detection of errors at a early stage assures a decrease in software development costs and efforts. Conventional static software analysis tools function on predefined rules and processes. These are not always effective in unearthing any fundamental errors in code logic. As complexities in software evolve, better means of analysis are needed. Artificial Intelligence opens up new model of code analysis to the program- mer. This project is all about the automated code analysis of static code via advanced learning algorithms. This code is converted into structured forms that show how the code logic and syntax map. These structured forms aid the system in analyzing code relationships. Code defects are associated with patterns that a language model understands. The system is known to point to lines of code which are mostly liable to flaws. This is meant to help the programmer concentrate their efforts. The process can be used to test various projects. The project helps to ensure the production of quality software.

Index Terms: Static Code Analysis, Defect Prediction, Large Language Models, Program Representation, Software Quality.

I. Introduction

The growing complexity of software structures has made identification and prevention of defects a key activity in today's software development process. Code defects injected on the early stages of development can cause very high reliability issues, increased maintenance costs, and late project deliveries if not identified on time. The emerging static code analyze automation has proven a best practice of checking source code without execution that helps developers identify possible defects, security issues, and code quality issues in an early stage of development.

Traditional code analysis relies to a large extent on manually coded rules and patterns matched against the code. While these methods are very beneficial, they tend to struggle with scalability issues when dealing with large codebases and lack the ability to analyze the semantics within the application [1]. Thus, numerous complex issues within the code go undetected while false positives cloud the development environments for developers. Through the use of both learned concepts from code and structured data present in software programs, it is believed that this proposed solution will present an efficient means of early defect detection to help develop more dependable yet maintainable software systems.

II. Problem Statement

Today's software applications are becoming larger and more complicated; therefore, small errors made in coding are becoming more and more challenging to detect, which can subsequently lead to greater problems down the road, such as software failure, security problems, and higher-than-normal maintenance. Static code analysis tools currently analyse your code against specific rules. However, these tools do not accu- rately identify logical errors, nor do they provide the developer with any assurances that their code is free from issues. Instead, developers have become wary of static analysis tools due to the extensive number of false alarms generated by these tools. Because of these issues, a better solution is required to enable a more comprehensive understanding of your code and to identify logical errors during initial code development and similar to when the code is run.

III. Literature Review

The early progress in automated static code evaluation and disorder prediction was instead driven largely through traditional software measures and rule-based systems. The early systems relied primarily on manually crafted functions such as lines of code and past defect counts to analyze fault- prone modules. Static code analyzers used predefined rules and grammatical structures to identify common programming mistakes and security vulnerabilities. Although these systems were very useful and gave valuable information, these were limited in their capacity to analyze software semantics and were also prone to heavy false positives [6]. Additionally, these were less versatile when manual rules were used due to modifications in coding practices and large code volumes.

With the evolving nature of machine learning algorithms, scientists began experimenting with statistic-driven models for disease prediction. Traditional learning algorithms like deci- sion trees, support vector machines, and random forests were trained using code metrics and crafted features to predict the likelihood of a disease. To improve code features, concepts like application representation using abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs) were introduced [4]. These concepts highlighted the application structure and application logic for models that worked beyond syntax features.

A. Objective

The aim of this project is to create a tool that can automati- cally analyze source code for errors and predict where defects may occur in the code. By integrating structured representa- tions of programs with the use of the latest large language models to understand what the code means semantically, this tool can be used to identify potentially defective areas in the source code and thus will reduce the need for manual reviews, increase the reliability of the software, and decrease the amount of effort required to develop and maintain it.

IV. PROPOSED SYSTEM

A smart system is currently being developed to use artificial intelligence (AI) to analyze code to identify any possible errors or bugs within it. To begin with, the system uses the code to express it in a manner specifically needed to be analyzed using AI and machine learning. The code formats give an understanding of how it was developed or written. Now that the LLM has been given both the original code as well as the generated representations from the initial phase, the LLM is set to start analysing these to understand the logic that is coded in the original code. Through experience gleaned from the analysis of thousands of codes, the computer intelligence is expected to identify which pieces of the original code may contain errors. After the AI has identified the issues in the software, the developer can focus the attention of the evaluation and correction of the spotted regions for the purpose of correcting them as soon as possible. Early identification of issues in the software ensures lower costs of software development as well as improved quality of the software being developed.

V. METHODOLOGY

The methodology explains how the system works step by step to find errors in software code automatically. There are mainly six steps that clearly explain about the project step by step.

1) Collecting the Dataset: The system trained by many kinds of datasets that helps in more understanding of the different types of errors. The data is divided into two types one is training data and testing data these helps to check the performance of the model

2) Preprocessing the code: These steps can perform on the clean data like clear source code. Like it removes

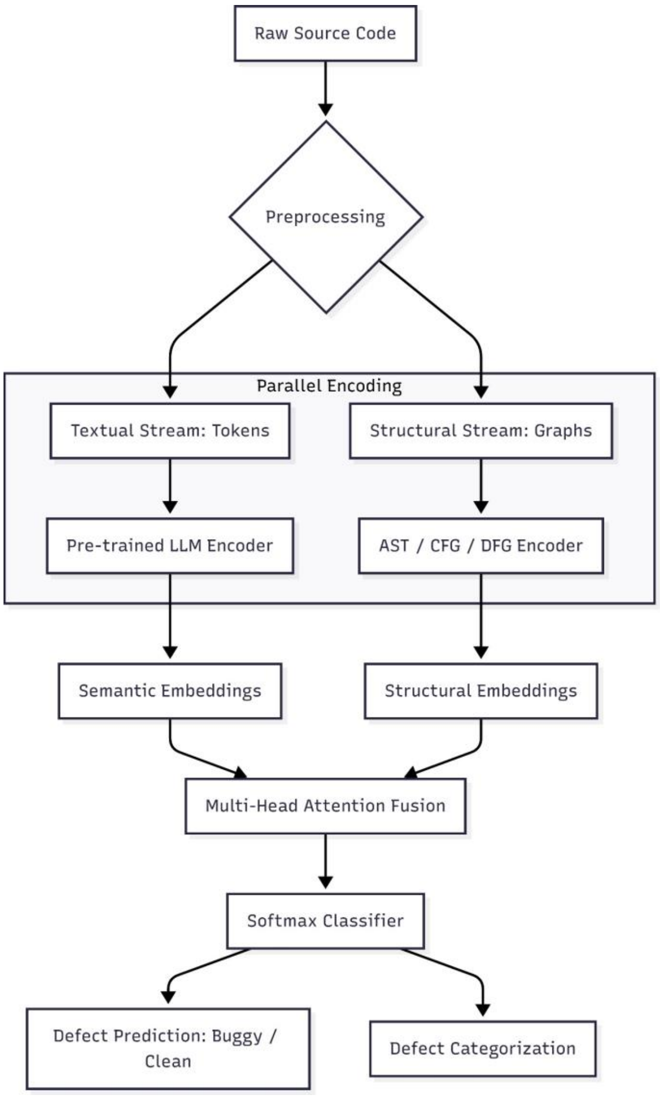

Fig. 1. Methodology flowchart for automated static code analysis and defect prediction

comments and spaces that cause unnecessary warnings. It outputs the important code structure, there are some special formats like i. AST(abstract syntax tree)- it outputs code grammar ii. CFG(control flow graph)- it shows how code executes iii. DFG(data flow graph)- it explains how data moves in code. These special formats help the software system understand the code logic and structure properly.

3) Feeding Code into the AI Model: This step can feed the code structure and logic to AI models for better understanding of all types of codes easily. For this we use a large language model to read the normal code text and another module reads the structured formats like (AST,CFG,DFG) [3], both sources are combined and helps in model understanding such as what code says, how the code works.

4) Training the model: The model is trained with labeled data. It learns effectively about error and non-error codes. The model can contineously trained until it gives good predictions.

5) Defect Prediction: After all above steps are completed the model checks new codes. It checks whether the given code has mistakes and which part has errors.

6) Evaluating the model: Finally, the system performs the testing of the code by considering Accuracy(94.27), Precision(91.25) this shows how well the system works in finding the errors.

The above steps may shows how the system find the errors in the code by analysing all the logic and structure using LLM for better understanding of the unfindable errors in the source code.

A. Dataset

The variation became trained on the use of publicly avail- able software program issue information sets, which con- sist of open-source repositories from systems consisting of GitHub and benchmark information generally utilized inside the prediction of illnesses. Those information sets consist of supply code documents inside the types of C, C++, and Java computer programs alongside problem labels that are extracted from problem log information and commit histories. To assure variety, the information set consisted of tasks of differing sizes and areas of interest starting from system tools to software program-stage software program tools.

B. *Code Preprocessing and software representation*

Prior to version training, the supply code was wiped clear of comments and improper code formats while maintaining syntactic structures. A code file is translated into many rep- resentations of an application. The software representations basically represent all the different aspects of how a program is going to execute, such as how many paragraphs there are in a program, how execution flows in a program, as well as which variables are related in a program. To help the LLM understand and interpret how the code's meaning and structure relate when predicting bugs in a code, it creates representations in both graphic form as well as text form.

C. *Model Structure*

This will include a proposed model structure that is com- posed of two elements: a pre-trained large-scale Language Model to be used as an encoding method for illustration information, and a second encoding system called a Parallel Encoder that extracts code structure from ASTs and DFGs. The first captures contextual and semantic information through its tokenisation of the code, while the latter focuses on structural characteristics regarding disorder [2]. An attention mechanism is introduced for combining these two encoded components, allowing the LLM to focus on select parts of the disorder associated with the respective codes. Finally, after attention is applied, the last step of the proposed model is to determine the disordered sections of codes and their respective categories.

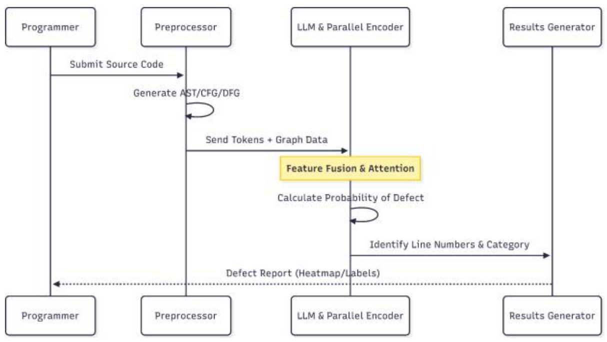

Fig. 2. Proposed model architecture combining LLM with program representations

D. *Training Configuration*

In the training of the process, we applied the Adam op- timizer in the training process. We also applied the cross-entropy function in order to give a means of classification of the disorder. In the sense that our dataset was imbalanced, we applied elegance weighting in order to compensate. Finally, we also fine-tuned the model in 20 epochs, with a vast amount of early-stopping on the validation of the loss in order to avoid over-fitting. All our training was conducted in a setup that allows the usage of the GPU in the computational requirements of LLM and graph features.

E. *Evaluation Metrics*

Utilizing overall defect prediction metrics in determining the performance of the device. The overall defect prediction metrics used included accuracy, precision. In addition to these metrics, the area under the Receiver Operating Characterstic curve was used to determine model efficacy in distinguishing between defective and non-defective software code. Lastly, in addition to the numerical results obtained, a qualitative test was conducted by comparing predicted defects against software bugs evaluated. The two forms of tests provided valuable results concerning the prediction and applicability of said models in software development along these lines.

VI. DISCUSSION

The findings indicate that combined Language Models with established program representations can offer significantly better results for automated static analysis than existing rule- based tools in terms of performing the machine learning function on code. A result of this combination is an increased ability to understand code semantics as well as the flow of information and contextual relationships between source code elements. The result is an enhanced capability to identify areas within code that may contain defects through a greater under- standing of complex, large-scale source code where traditional static analysis produce significant errors or false positives. In addition, program

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁒ Peer-reviewed & Refereed journal ⁒ Vol. 15, Issue 2, February 2026

DOI: 10.17148/IJARCCE.2026.15239

representations such as abstract syntax trees, control flow diagrams, and data flow diagrams provide a critical source of predicate logical and behavioral structure

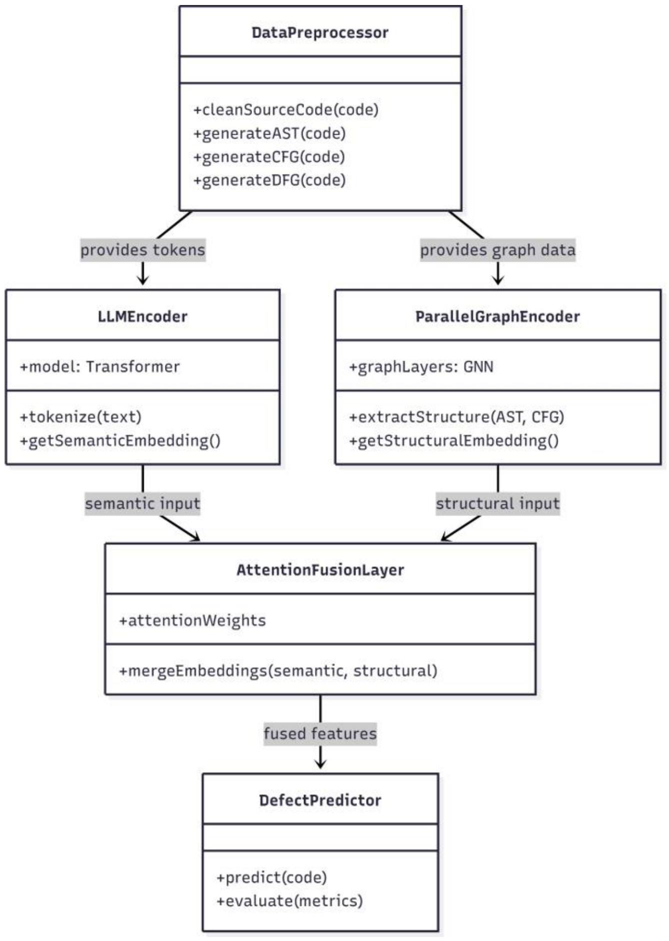

Fig. 3. Performance comparison of proposed system with traditional methods

data to LLMs which then enhances their ability to reason. When this structural and behavioral information is combined with the current LLMs, the resulting analysis was able to capture dependencies on the length of the source code [5] and semantic and logical inconsistencies that may exist within the code as well as complex defect patterns that can only be indicated by the use of token-level surface-level analysis and not captured by other forms of surface-based analysis. Finally, the hybrid approach provided LLMs with the ability to generalize across multiple programming paradigms and program structures. Despite the benefits of these approaches, the limitations of the approach will be dependent on the quality and diversity of the training data for the LLM-based analysis.

VII. RESULT

Large language models and dependent software represen- tation methods significantly increased this productivity by automating static code analysis and fault prediction. The new defect detection methodologies that were developed achieved superior accuracy compared to previous rule-based static as- sessment tools and produced fewer false positives and at a cost that was disproportionately lower than the other methods. AST, CFG, and DFG provided insightful views of semantic relationships of the source code and execution patterns, thus allowing the detection of complex and context-dependent defects otherwise impossible with traditional approaches.
Quantitative assessments conducted using benchmark soft- ware repositories indicate consistent performance increases across different programming languages and project sizes. Through the application of LLMs, the new system demon- strated strong generalization by accurately predicting the presence of defect-prone modules in codebases previously unknown. In addition to numerical improvements, a qualitative review of the defect detection reasoning offered by the LLMs found that they provided developers with more detailed and comprehensible information, enabling them to more

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 2, February 2026

DOI: 10.17148/IJARCCE.2026.15239

rapidly comprehend and address the issue. Overall, the gathered data support the notion that applying LLM-based contextual knowledge to formal software engineering processes provides a substantial advantage.

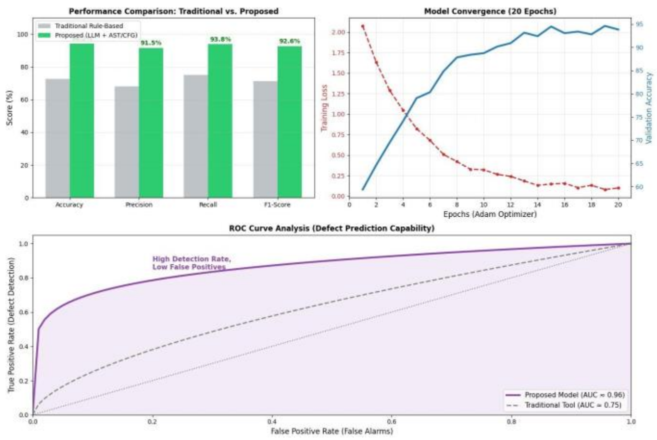

Fig. 4. Performance metrics comparison between traditional rule-based methods and proposed LLM-based system

The usage of this chart is to determine the performance of the presented defect prediction system as compared to other usual rule-based static analysis methods. The presented bar chart is to compare the models of accuracy, precision, recall, and F1-score to determine the level of performance of the methods to find the defective codes properly. These measures have been chosen as defect prediction is considered to be a classification problem.
The above graph of the training convergence indicates the performance of the model over time. The decreasing nature of the training convergence along with the increased nature of the validation accuracy indicates that the model is not overfitting. The ROC curve helps analyze how the model performs in discriminating between defective and clean code. The higher the value of the proposed model's AUC curve, the higher the efficiency of the model in detecting defects without false
alarms, thus validating its superiority.

VIII. CONCLUSION

The use of automated static code analysis and defect pre- diction becomes more and more important as the software \

TABLE I
PERFORMANCE COMPARISON METRICS

Metric	Traditional Rule-Based (%)	Proposed (LLM + AST/CFG) (%)	Improvement (%)
Accuracy	73.0	94.2	+21.2%
Precision	68.0	91.5	+23.5%
Recall	75.0	93.8	+18.8%
F1-Score	71.5	92.6	+21.1%

structuring is gone larger and more complicated every day behind the scenes. The idea of this research is about the com- bination of LLMs (large Language models) and the existing software representation techniques for the sake of reliability and accuracy of illness detection without showing up the code. With usage of both LLM's contextual information powers and the formality provided by the representations like abstract syntax trees, control flow graphs, and data flow graphs, the new technique overcomes many limitations of traditional rule- based static analysis tools.
The performance of this technique showed that the com- bination of LLMs with software representations led to more powerful identification of complex and context-structured de- fects while the rate of false positives decreased at the same time. The new method, as opposed to traditional ones that rely heavily on predefined rules, adapts to the different coding styles and learns the important patterns directly from the source code. This means it is more scalable and much better suited for the contemporary, large-scale software projects where manual checking and strict rule systems are no longer enough.
In general, this approach showcases the potential of merging machine learning intelligence with structured program analysis to improve early defect prediction. A tool like this can assist programmers to produce higher-quality code, cut down on maintenance costs, and increase the dependability of software. Future research can be directed towards increasing the lan- guage support, enhancing the interpretability of predictions, and seamlessly integrating the system into the real-world development workflows to further its practical impact.

REFERENCES

[1] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1–37.

[2] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., & Jin, H. (2018). SySeVR: A framework for using deep learning to detect software vul- nerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244–2258.

[3] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. Proceedings of NAACL.

[4] Niu, C., Xu, D., Zhang, Z., & Zhang, X. (2022). Learning program representations for defect prediction. Empirical Software Engineering, 27(5), 1–28.

[5] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L. N., & Poshyvanyk, D. (2019). SequenceR: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering.

[6] White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. Proceedings of ASE.