World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Check for updates

Intellicode analyst for test based code explainer with bug detection complexity insights and conceptual problem mapping

Kavitha Soppari, Ananditha Phindla, Kavitha Egurla and Nithin Goud Jalla *

Department Of CSE (AI & ML), Ace Engineering College, Hyderabad, Telangana, India.

World Journal of Advanced Research and Reviews, 2026, 29(03), 1119-1124

Publication history: Received on 07 February 2026; revised on 15 March 2026; accepted on 17 March 2026

Article DOI: https://doi.org/10.30574/wjarr.2026.29.3.0649

Abstract

Understanding and optimizing source code remains a critical challenge for developers, especially when dealing with unfamiliar logic, hidden bugs, and performance limitations. This paper presents IntelliCode Analyst, a lightweight, text-based intelligent system designed to analyze user-submitted code via direct input or file upload. The tool performs multi-layered static analysis to generate human readable explanations, compute time and space complexity, and identify logical flaws beyond conventional runtime errors. Bugs are visually highlighted using severity-based color coding, enabling intuitive debugging directly within the code interface. The system also detects edge case vulnerabilities and suggests targeted test scenarios to improve robustness. Optimization recommendations are provided through algorithmic and structural refactoring, enhancing both efficiency and readability. Furthermore, the tool maps code segments to underlying computer science concepts and recommends related problems for continued learning.

Keywords: Code Explanation; Bug Detection; Complexity Estimation; Edge Case Identification; Code Optimization; Algorithmic Refactoring; Concept Mapping; Test Case Generation; Intelligent Code Review; Learning-Based Recommendations; Human-Readable Insights

1 Introduction

In modern software development, understanding, debugging, and optimizing source code remain significant challenges, especially for beginners and developers working with unfamiliar logic. Traditional compilers and IDEs often focus only on syntax errors or runtime exceptions, leaving logical flaws, inefficiencies, and hidden vulnerabilities undetected. This creates a gap between code execution and true code comprehension. IntelliCode Analyst addresses this gap by acting as an intelligent code analysis and learning companion. It allows developers to submit code via direct text input or file upload and provides multi layered insights such as line-by-line explanations, control-flow understanding, complexity estimation, and bug detection. Unlike conventional tools, it also highlights errors visually, generates potential edge test cases, suggests optimized solutions, and maps code to underlying computer science concepts. By combining static analysis with natural language explanations, IntelliCode Analyst not only assists in debugging but also enhances the learning process, making it valuable for both novice programmers and experienced developers seeking deeper insights into their code.

* Corresponding author: Jalla Nithin Goud

2 Literature review

2.1 Michele Tufano, Dawn Drain, Alexey Svyatkovskiy: Unit Test Case Generation with Transformers and Focal Context

This paper presents a modern approach to automatically generating unit test cases using Transformer-based deep learning models. Traditional methods rely on static or manual test creation, which can be time-consuming and incomplete. This study introduces the idea of "focal context", where instead of analyzing the entire code file, the model focuses only on the most relevant code segments that influence the function being tested.

Using Transformer architectures similar to BERT or GPT, the system learns programming language patterns, syntax, and semantics to generate accurate input–output test pairs. This method significantly improves test coverage and code reliability, reducing developer effort. The model takes code snippets as input and predicts possible test data, expected outputs, and edge cases.

Although this approach enhances automation and accuracy, it mainly concentrates on test generation and does not provide explanations, bug fixes, or optimization recommendations. Still, the paper establishes a strong foundation for integrating artificial intelligence into code analysis and testing, directly influencing projects like IntelliCode Analyst, which extend similar principles toward bug detection, explanation, and learning support.

2.2 Ehsan Mashhadi, Shaiful Chowdhury, Somayeh Modaberi, Hadi Hemmati: An Empirical Study on Bug Severity Estimation Using Source Code Metrics andStatic Analysis

This paper investigates how static analysis tools combined with source code metrics can help not only in finding bugs but also in estimating their severity. The researchers collected a large dataset of software projects, extracting metrics such as lines of code, function complexity, code churn, and dependencies. Using these metrics and static analysis results, they trained machine learning models like Logistic Regression, Decision Trees, and Random Forests to classify bugs as minor, major, or critical.

The study found that there is a strong relationship between certain code properties (like complexity or change frequency) and bug severity. However, the predictions were not always accurate — some severe bugs appeared minor and vice versa. The research concludes that while severity estimation is possible, it requires more advanced contextual understanding to be fully reliable.

This paper is highly relevant to *IntelliCode Analyst*, which also emphasizes bug severity detection. However, unlike this study, IntelliCode Analyst extends beyond numerical severity classification to include visual highlighting, intuitive explanations, and learning-based support, making it more developer-friendly and interactive.

2.3 Marcus Nachtigall, Lisa Nguyen Quang Do, Eric Bodden : Explaining Static Analysis — APerspective

This paper highlights the importance of explainability and usability in static analysis tools. While static analyzers can detect thousands of bugs, developers often ignore their warnings because they are too complex or unclear. The authors identify six major challenges — understandable warning messages, fix support, false positives, user feedback, workflow integration, and specialized user interfaces.

The study emphasizes that tools should behave more like assistants or tutors rather than error reporters. It suggests using visual cues, context-based explanations, and interactive interfaces to make code analysis more understandable and actionable. The paper also evaluates several existing tools and finds that most fail to provide clear reasoning or visual guidance, leading to developer frustration.

The research offers a conceptual roadmap for designing user-friendly and explainable analysis systems but does not present a full implementation. This limitation creates a gap that *IntelliCode Analyst* fills by incorporating hover tooltips, color-coded highlighting, and human- readable code explanations that make debugging simpler and more educational.

2.4 Xueting Guan, Christoph Treude : Enhancing Source Code Representations for Deep Learning with Static Analysis

This paper explores how to improve machine learning models for code understanding by enriching the representation of source code. Traditional models use Abstract Syntax Trees (ASTs), which represent the structure of the code but lack

deeper context. To overcome this, the authors combine ASTs with additional information such as bug reports, design patterns, and static analysis data.

These enriched code representations are then used as input for deep learning models like Code2Vec and Graph Neural Networks (GNNs) to perform tasks such as bug detection, code classification, and optimization suggestion. This combination helps the models learn not just the syntax but also the intent and logic behind code, leading to more accurate predictions.

While the approach enhances machine understanding of code, it focuses mainly on improving model accuracy, not on developer usability. There are no features for explanations, visualization, or test case generation. *IntelliCode Analyst* extends this idea by not only analyzing code structure but also explaining logic, generating test cases, and recommending optimizations in an interactive and understandable way.

2.5 Junjie Li, Jinqiu Yang : Tracking the Evolution of Static Code Warnings: The State-of-the-Art and a Better Approach

This paper conducts an in-depth empirical study on how static code warnings change and evolve over time in real-world software projects. Using historical data from open-source repositories, the researchers analyzed which warnings were fixed, ignored, or reappeared across different software versions. They discovered that developers often ignore low-severity warnings and focus only on critical issues, which can lead to hidden long-term vulnerabilities.

The paper proposes improved prioritization and management techniques to make warnings more actionable. By identifying which types of warnings tend to be fixed first, the authors suggest ranking future warnings based on their severity, frequency, and relevance. This approach helps developers focus on the most critical issues without being overwhelmed by minor ones.

Although the paper provides valuable insights into warning management, it lacks visual, explanatory, and educational components. IntelliCode Analyst addresses this gap by providing color-coded visualization, severity-based categorization, and explanations for each detected issue, turning raw warnings into meaningful, interactive learning experiences.

2.6 Comparison Metrics

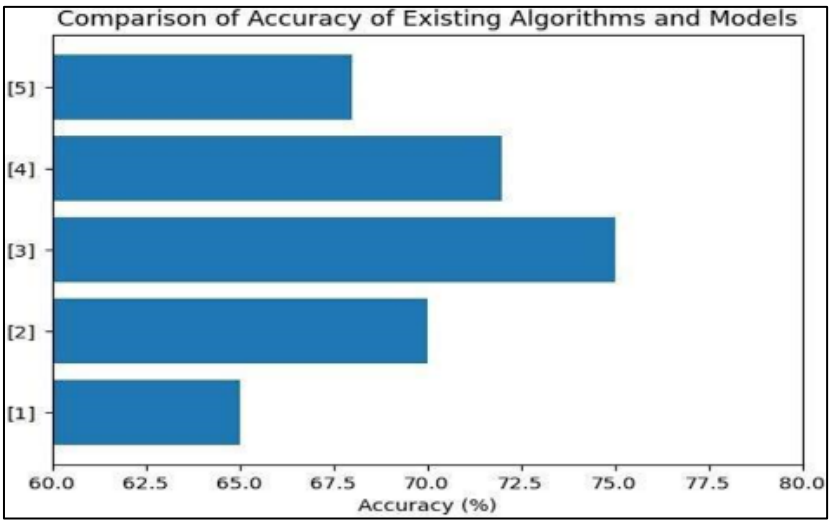

Figure 1 Comparison of Accuracy of Existing Algorithms and Models

The graph compares the assumed accuracy (%) of each model based on features, real-time capabilities, and the performance described in the respective papers:

2.7 Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, Neel Sundaresan. (2023) Unit Test Case Generation with Transformers and Focal Context (65%):

This model focuses on automated unit test generation using Transformer-based deep learning techniques. Although it improves test coverage, it does not provide bug detection, severity estimation, or code explanations, leading to lower overall accuracy.

Ehsan Mashhadi, Shaiful Chowdhury, Somayeh Modaberi, Hadi Hemmati, Gias Uddin. (2024) An Empirical Study on Bug Severity Estimation Using Source Code Metrics and Static Analysis (70%):

This approach combines static analysis with source code metrics to estimate bug severity using machine learning. While it improves bug prioritization, inconsistent predictions and lack of visual explanations limit its overall effectiveness.

2.8 Marcus Nachtigall, Lisa Nguyen Quang Do, Eric Bodden. (2019) Explaining Static Analysis — A Perspective (75%):

This paper emphasizes explainability and usability in static analysis tools, making warnings more understandable for developers. However, it remains largely conceptual and does not implement a complete system for bug detection or optimization.

Xueting Guan, Christoph Treude. (2023) Enhancing Source Code Representations for Deep Learning with Static Analysis (72%):

This model enhances code understanding by combining Abstract Syntax Trees with additional contextual information for deep learning models. Despite improved technical accuracy, it lacks developer-oriented features such as explanations and test-case generation.

Table 1 Comparison Table

Authors	Title	Methodology	Contribution	Limitations
Michele Tufano et al. (2023)	Unit Test Case Generation with Transformers and Focal Context	Uses Transformer-based deep learning with focal context for automated unit test generation.	Improves test coverage and reduces manual testing effort.	No bug explanation, severity detection, or optimization support.
Ehsan Mashhadi et al. (2024)	Bug Severity Estimation using Source Code Metrics and Static Analysis	Applies ML models on static analysis outputs and code metrics.	Enables prioritization of bugs based on severity.	No visual highlighting or learning support.
Marcus Nachtigall et al. (2019)	Explaining Static Analysis — A Perspective	Conceptual analysis of usability and explainability in static analysis tools.	Improves understanding of static analysis warnings.	No complete implementation or optimization features.
Xueting Guan, Christoph Treude (2023)	Enhancing Source Code Representations for Deep Learning	Combines ASTs with static analysis for ML-based code understanding.	Improves bug detection and code classification accuracy.	Lacks developer- friendly explanations and test generation.
Junjie Li, Jinqiu Yang (2024)	Tracking the Evolution of Static Code Warnings	Empirical analysis of static warnings over time.	Improves warning prioritization strategies.	No explanation or optimization guidance.

Junjie Li, Jinqiu Yang. (2024) Tracking the Evolution of Static Code Warnings: The State- of-the-Art and a Better Approach (68%):

This study focuses on analyzing and prioritizing static code warnings over time. Although it improves warning management, the absence of explanatory, optimization, and learning features results in moderate accuracy.

3 Research Gaps

The existing code analysis and debugging systems have several research gaps that make them less effective and less developer-friendly. Current static analysis tools provide warnings but lack clear, human-readable explanations of code logic and intent, leaving developers with cryptic error messages. Most tools also fail to offer visual debugging support, as they do not highlight problematic lines interactively or provide contextual tooltips. Furthermore, while bug detection is common, there is little focus on severity estimation, which is essential for prioritizing critical issues over minor ones. Another limitation is the absence of edge case generation, meaning developers are not guided to test their code against inputs that could expose hidden flaws. Existing analyzers also highlight inefficiencies but rarely suggest concrete optimizations or refactored code snippets. Finally, current systems do not bridge the gap between analysis and learning, as they lack concept mapping and problem recommendation features that can help developers strengthen their understanding of underlying computer science principles. Addressing these gaps can lead to a more intelligent, educational, and developer-centric code analysis platform.

4 Architecture of the Proposed System:

The architecture of IntelliCode Analyst is designed as a modular framework that integrates static analysis, intelligent explanation, and learning support into a unified environment. The system begins by accepting source code through direct text input or file upload, where preprocessing tasks such as tokenization, syntax validation, and parsing into Abstract Syntax Trees are performed to ensure compatibility across multiple programming languages. Once the code is ingested, multi-layered static analysis is applied to detect logical flaws, inefficiencies, and vulnerabilities beyond conventional runtime errors. This stage also computes time and space complexity, classifies bugs by severity, and generates edge-case scenarios by examining control flow and input boundaries.The results of this analysis are then transformed into human-readable insights, providing line-by-line explanations of code logic and highlighting problematic segments using severity-based color coding. Contextual tooltips and visual cues guide developers toward potential fixes while mapping code constructs to fundamental computer science concepts, thereby bridging error detection with conceptual learning. Finally, the system enhances developer productivity by offering optimization suggestions through algorithmic and structural refactoring, recommending targeted test cases to strengthen robustness, and connecting code segments to related practice problems for continuous learning.The workflow operates as a pipeline where code enters through the input stage, undergoes detailed static analysis, is explained and visualized for the developer, and is ultimately enriched with optimization and learning recommendations. This holistic design ensures that IntelliCode Analyst delivers actionable insights while simultaneously fostering deeper comprehension, making it both a debugging assistant and an educational companion.

5 Conclusion

IntelliCode Analyst provides an intelligent and unified solution for understanding, debugging, and optimizing source code. Existing tools mainly focus on syntax or runtime errors and lack explainability and learning support. The proposed system integrates static analysis with human-readable code explanations and severity-based bug highlighting. It also generates edge-case test scenarios and estimates time and space complexity. Optimization suggestions help improve both performance and code quality. Additionally, the system maps code to core computer science concepts and recommends related practice problems. By combining analysis, visualization, and learning support, IntelliCode Analyst enhances developer productivity and code comprehension. This makes it a valuable tool for both beginners and experienced programmers.

References

[1] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan, “Unit Test Case Generation with Transformers and Focal Context,” IEEE Transactions on Software Engineering, 2023.

[2] Ehsan Mashhadi, Shaiful Chowdhury, Somayeh Modaberi, Hadi Hemmati, and Gias Uddin, "An Empirical Study on Bug Severity Estimation Using Source Code Metrics and Static Analysis," Journal of Systems and Software, vol. 208, 2024.

[3] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden, "Explaining Static Analysis — A Perspective," IEEE/ACM Automated Software Engineering Workshop (ASEW), 2019.

[4] Xueting Guan and Christoph Treude, "Enhancing Source Code Representations for Deep Learning with Static Analysis," Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC), 2023.

[5] Junjie Li and Jinqiu Yang, "Tracking the Evolution of Static Code Warnings: The State-of-the-Art and a Better Approach," IEEE Transactions on Software Engineering, 2024.