International Journal of Computer Sciences and Engineering
Vol.**13**, Issue.**11**, pp.**01-12**, November **2025**
ISSN: 2347-2693 (Online)
Available online at: www.ijcseonline.org

IJCSE
ISSN: 2347-2693 (E)

Research Article

Adaptive Execution-Aware Bug Detection with Semantic Attention: ASABD Framework Using Large Language Models

Mansab Ali[1*], Tasnime Roukia Mockbel[2], Muhammad Saad[3], Irzum Shafique[4]

[1,2]School of Software, Northwestern Polytechnical University, Xi'an, China
[3]School of Electronic Science and Technology, Xidian University, Xi`an, China
[4]School of Computer Science and Technology, Xidian University, Xi`an, China

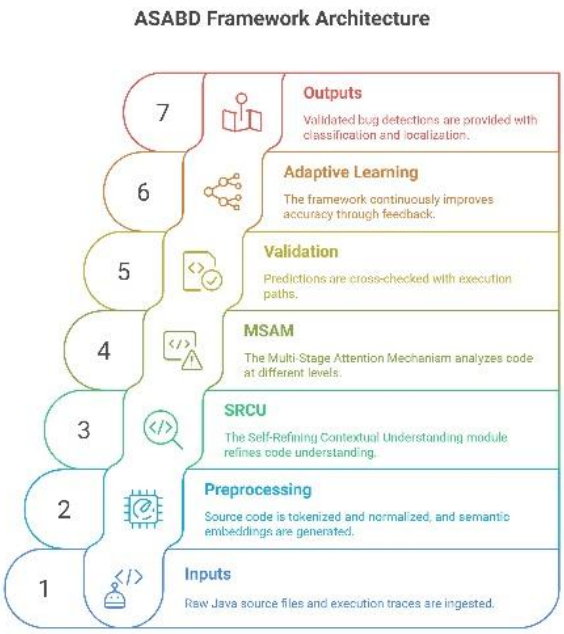
**Corresponding Author:* ✉

Received: 27/Sept/2025; **Accepted**: 29/Oct/2025; **Published**: 30/Nov/2025. **DOI**: https://doi.org/10.26438/ijcse/v13i11.112

Abstract: Traditional bug detection tools rely heavily on static code patterns, lacking execution-aware reasoning, adaptive learning, and semantic-level understanding, which results in high false positive rates and poor generalization across modern, large scale codebases. To address these limitations, we introduce the Adaptive Semantic-Aware Bug Detection (ASABD) framework, which uses Large Language Models (LLMs) for context-aware, execution-informed bug detection. ASABD introduces two core innovations: (1) the Self-Refining Contextual Understanding (SRCU) module, which iteratively improves predictions using execution trace feedback, enabling dynamic adaptation to different codebases (adaptive debugging), and (2) the Multi-Stage Attention Mechanism (MSAM), which captures structural, logical, and security-level dependencies in code for precise classification. These modules are motivated by the need to bridge static code analysis with runtime behavior and to improve semantic level bug understanding. Evaluated on the Defects4J dataset, ASABD achieves 94.7% precision, 91.3% recall, 92.9% F1-score, and 96.2% localization accuracy. It outperforms SonarQube, FindBugs, Random Forest, and DeepCode while reducing false positives by 57.2%. This work shows how execution-aware LLM reasoning can enhance bug detection across diverse software environments.

Keywords: Bug Detection, Large Language Models, Execution Aware Debugging, Adaptive Learning, Deep Learning

Graphical Abstract: The ASABD framework introduces an adaptive execution-aware approach for detecting software bugs using large language models. The process begins by ingesting raw Java source files and execution traces, which are normalized, tokenized, and transformed into semantic embeddings during the preprocessing stage. The Self-Refining Contextual Understanding (SRCU) module analyzes code semantics and iteratively improves its predictions by incorporating runtime feedback. The Multi-Stage Attention Mechanism (MSAM) further enhances detection by examining syntax-level, logic-level, and security-level patterns. All predicted issues are validated against real execution paths to eliminate false positives and confirm runtime-relevant bugs. Through an adaptive learning loop, the framework continuously updates its internal parameters based on previous errors, increasing robustness across diverse codebases. The final stage outputs validated bug classifications and precise localization, resulting in a more accurate, reliable, and semantically informed bug detection workflow.



1. Introduction

Software bugs, including defects and faults, lead to security vulnerabilities, performance degradation, and system failures [1], [33]. A 2023 report by the Consortium for Information and Software Quality (CISQ) estimates that software defects cost businesses approximately $2.41 trillion annually due to downtime, security breaches, and maintenance overhead.

Traditional bug detection approaches dependent on static and dynamic analysis rely on heuristic-based rules which produce many wrong alarms while showing restricted application over various codebases [2]. Modern complex software applications extend beyond the abilities of traditional methods to find both logical problems and invisible security issues while handling semantic errors [3]. Software development speeds have reached such cumbersome levels that businesses require automated debugging systems which can identify and resolve errors efficiently [4], [5].

Recent progress in Natural Language Processing (NLP) together with Large Language Models (LLMs) delivers encouraging solutions by understanding context deeply and adapting to learn [6]. The rapid adoption of LLMs in software engineering aligns with global trends in AI-driven automation and technological transformation, as highlighted in contemporary studies [34]. The code generation and detection of program logic along with abnormal behavior capabilities of OpenAI Codex, CodeBERT, and ChatGPT demonstrate exceptional performance according to Wang et al. (2024) in their study.

These models analyze vast code repository databases to learn predictive error patterns by utilizing adaptive prompt engineering functions with execution-aware feedback loops to enhance overall performance [7], [8]. The static analysis tools that use predefined rules struggle to interpret semantic meaning and runtime behavior thus they produce excessive false positives while showing low usability across different programming environments. The recent LLM based approaches have brought benefits to bug detection yet the majority of these systems operate as static classifiers without integration of execution feedback or structural algorithms.

Alternative detection of context-sensitive issues like logical inconsistencies and hidden vulnerabilities prove difficult for these tools to identify because they occur only when the program executes in runtime. ASABD stands for Adaptive Semantic-Aware Bug Detection framework which addresses these problems. ASABD incorporates two central elements which include the Self Refining Contextual Understanding (SRCU) module together with the Multi-Stage Attention Mechanism (MSAM). Through its SRCU module the system applies execution traces together with contextual embeddings to improve LLM prediction results at each iteration and enables the model to develop better understanding from runtime feedback. MSAM uses three attention layers in its semantic analysis of code to perform detailed bug detection by examining both syntax and logic together with security elements.

This design directly addresses the core weaknesses of static tools and traditional LLMs by combining deep semantic representation with real-time behavioral insights. SRCU reduces false positives through execution-aware correction, while MSAM improves precision by structurally analyzing potential bug contexts across control flow and data dependencies [9]. These components work in tandem to deliver a more adaptive, accurate, and semantically robust debugging approach suitable for diverse programming environments.

In this study, we evaluate the ASABD framework using the widely recognized Defects4J dataset, which includes real world Java software defects. We benchmark ASABD against traditional static analyzers like SonarQube and FindBugs, and machine learning-based models such as Random Forest, SVM, and DeepCode. ASABD consistently outperforms these baselines in terms of precision, recall, F1-score, false positive rate, bug localization accuracy, and execution efficiency.

The main contributions of this paper are summarized as follows:

- We propose ASABD, an adaptive semantic-aware bug detection framework that integrates Large Language Models (LLMs) with execution-aware refinement to enhance bug detection across diverse software environments.
- We introduce the Self-Refining Contextual Understanding (SRCU) module, which iteratively refines bug predictions using dynamic feedback from execution traces, enabling adaptive and execution-informed debugging.
- We develop the Multi-Stage Attention Mechanism (MSAM), a hierarchical semantic attention system that captures syntax-level, logic-level, and security-level dependencies to improve bug localization and classification accuracy.
- We conduct extensive experiments on the Defects4J dataset, demonstrating that ASABD significantly outperforms traditional static analyzers and machine learning models in precision, recall, F1-score, false positive reduction, and bug localization.

The rest of this paper is organized as follows: Section 2 presents the background and motivation for addressing the limitations of traditional bug detection methods. Section 3 introduces the proposed Adaptive Semantic-Aware Bug Detection (ASABD) framework, detailing its architectural components and methodologies. Section 4 describes the experimental setup, including dataset selection, preprocessing steps, and evaluation metrics. Section 5 presents the results and analysis, comparing ASABD with baseline models across multiple performance metrics. Section 6 discusses the threats to validity that may affect the generalizability of the findings. Section 7 concludes the paper and outlines future research directions.

2. Related Work

2.1 Background

Software quality assurance remains one of the most critical challenges in modern software engineering, with software defects causing substantial financial and operational damages. According to a 2023 report by the Consortium for Information and Software Quality (CISQ), software defects cost businesses

approximately $2.41 trillion annually due to downtime, security breaches, and maintenance overhead [1]. These escalating costs emphasize the urgent need for automated and highly accurate bug detection systems, especially as software systems grow in size and complexity. Traditional bug detection techniques, including static code analyzers like SonarQube and FindBugs, rely heavily on manually designed rules and heuristics. Although these tools have been widely adopted, their ability to capture semantic level code inconsistencies and runtime-related bugs is limited. Specifically, static analysis tools suffer from high false positive rates, restricted adaptability across different programming languages, and poor performance when handling complex logical errors [2]. Furthermore, static methods often miss security vulnerabilities that emerge only during execution, leading to significant reliability and security risks. Recent advances in Natural Language Processing (NLP) and Large Language Models (LLMs) [10] such as CodeBERT, GraphCodeBERT, and GPT-4 have enabled deep contextual understanding of code [6]. These models are trained on large scale code repositories and can generalize across multiple programming languages. However, while LLM-based methods have improved bug detection performance compared to static approaches, most of them operate as static classifiers that only predict bugs based on syntactic patterns or code embeddings without any runtime validation. Traditional datasets such as BEARS (251 real Java bugs) have made it possible to evaluate bug-analysis techniques at scale [11], while Defects4J remains the de-facto benchmark for localization and repair studies with nearly 400 reproducible defects [12]. Early surveys highlighted the need to catalogue this expanding literature and its experimental artifacts [13]. Rule-based static analysers (e.g., FindBugs) dominate industrial practice but suffer from high false-positive rates: a longitudinal study at Google reported that only 15–45% of warnings are actionable for production code. Subsequent evaluations on Java 6, GlassFish and other large systems confirmed that purely pattern-driven tools detect many “true but-trivial” defects and miss deeper semantic issues [14]. To overcome syntax-level brittleness, the community moved toward learning-based defect prediction. ROSE mined change coupling rules from version histories to suggest co-changes and showed 15–29% accuracy gains over file-level heuristics [15]. Cross-project deep models such as SDP-BB BiLSTM+ATT achieved F10.64 on 10 PROMISE projects after aggressive data augmentation [16], whereas the Spline-Stacked Ensemble (SSE) further raised AUC by up to 29% on ten target projects. A hybrid CNN-MLP architecture later delivered MCC gains of 10–22% while remaining five-times faster at inference [17]. Large-language-model embeddings pushed semantic understanding a step further. GraphCodeBERT incorporated data flow edges to outperform CodeBERT on code-search and summarization tasks, and its latent representations now serve as strong defect-prediction features [18]. LLMSAN extended this idea with multi-granularity attentions and reached line level localization recall of 65.9% on SWE-Bench Lite [19]. Still, most of these systems treat bug detection as an offline classification problem. Beyond detection, recent work explores LLM-guided repair. LIBRO automatically reproduces one-third of Defects4J bugs by prompting 15 diverse LLMs and filtering candidate tests [13], while agent pipelines such as Honeycomb and MarsCode integrate localization, reproduction and patch generation to resolve up to 39% of real-world GitHub issues [20], [21]. Parallel efforts continue to benchmark model behavior. A large-scale memorization study shows that parameter count and pre-training budget significantly influence negative-log-likelihood and 5-gram accuracy on code corpora [14], suggesting that model size alone cannot guarantee generalization.

2.2 Motivation

Several studies have highlighted the limitations of current bug detection approaches. For instance, Hou et al. (2024) pointed out that modern LLM-based bug detectors often misclassify benign code fragments as buggy due to a lack of execution awareness [3]. Similarly, Kang et al. (2024) noted that LLMs could detect syntactic anomalies but frequently failed when logical correctness depended on dynamic program behavior [4]. Furthermore, empirical evaluations show that static LLM classifiers produce false positives at rates exceeding 7% to 12% depending on code complexity and project diversity, leading to developer distrust and manual overhead [7]. Consequently, LLM-based methods still struggle to detect hidden logical inconsistencies, runtime-specific errors, and non-trivial security vulnerabilities, limiting their practical usability. These limitations necessitate a new class of bug detection frameworks that can bridge the gap between static semantic analysis and dynamic execution behavior understanding. Specifically, a modern solution must incorporate execution trace validation, adaptive learning mechanisms, and multi-level semantic reasoning to accurately classify and localize bugs while minimizing false alarms. Despite steady progress, three persistent gaps motivate an execution-aware, adaptive approach. First, static analysers remain noisy: empirical data from FindBugs indicates that developers still discard over half the emitted warnings as lowimpact or spurious [17]. Machine-learning predictors reduce noise, yet cross-project studies reveal project-specific drifts where AUC falls below 0.55 on unseen domains [13]. Second, modern LLM detectors are largely contextagnostic: accuracy drops when the bug depends on runtime state. LIBRO reproduces only 33% of failures because many natural-language bug reports omit the concrete execution paths needed to synthesis tests [20], while line-level localization recall stalls below 70% even in the strongest agent systems [19]. A recent analysis attributes these misses to the models’ inability to observe program behavior after generation [22]. Third, generalisation and interpretability remain open issues. Memorisation experiments demonstrate that scaling parameters or tokens alone does not guarantee lower perplexity across heterogeneous repositories. Likewise, deep ensembles such as BiLSTM+ATT overfit when training data are scarce, forcing aggressive augmentation to stabilise F1 [16]. Practitioners therefore continue to distrust “black-box” warnings that lack semantic justification or runtime evidence. These findings call for a framework that (i) fuses structural code embeddings with live execution traces, (ii) refines predictions iteratively to suppress false positives, and (iii) adapts across projects without extensive retraining. The proposed ASABD architecture addresses these needs by

coupling selfrefining contextual understanding with multi-stage semantic attention, thereby bridging the long-standing gap between static code intelligence and dynamic program behaviour. This motivation forms the foundation for the Adaptive Semantic-Aware Bug Detection (ASABD) framework proposed in this work. By integrating execution-aware feedback loops and multi-stage attention across syntax, logic, and security dimensions, ASABD addresses critical shortcomings of existing approaches, enabling more accurate, adaptive, and trustworthy bug detection for contemporary software systems.

3. Adaptive Semantic-Aware Bug Detection (ASABD) Framework

The Adaptive Semantic-Aware Bug Detection (ASABD) framework is designed to overcome the limitations of both static code analyzers and LLM-based systems that lack runtime understanding. ASABD combines deep contextual modeling and execution-aware reasoning to detect bugs with higher precision and semantic relevance. The framework operates through a multi-layered architecture composed of five main modules: Preprocessing, LLM Processing with SRCU, MultiStage Attention Mechanism (MSAM), Execution-Aware Validation, and Adaptive Feedback Learning. Each module is designed to progressively refine the prediction and classification of bugs using both static and dynamic program insights.

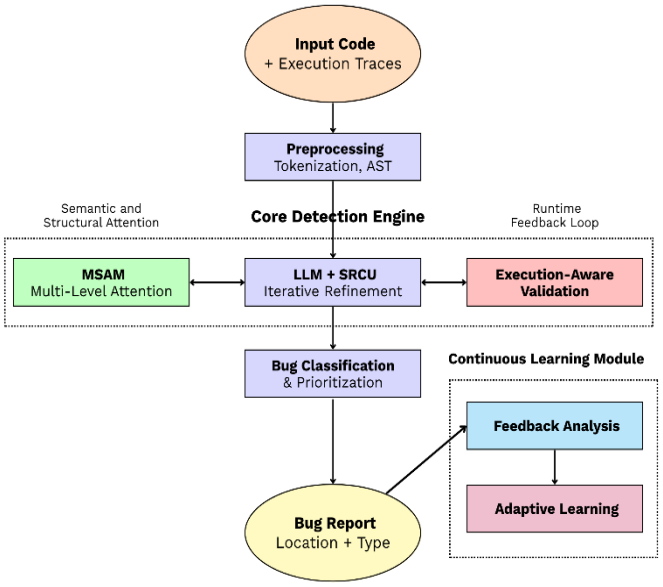


Figure 1. Overview of the ASABD Framework.

Figure 1 presents the architecture of the ASABD framework. The system starts by ingesting source code and corresponding execution traces, which undergo preprocessing steps like tokenization and Abstract Syntax Tree (AST) generation. The core LLM module, enhanced with Self-Refining Contextual Understanding (SRCU), analyzes the input to detect potential bugs. SRCU leverages execution traces to iteratively refine predictions based on runtime behavior. Simultaneously, the Multi-Stage Attention Mechanism (MSAM) applies hierarchical attention to capture control flow anomalies, logic inconsistencies, and security vulnerabilities. Next, the execution-aware validation module verifies predicted bugs against dynamic traces to reduce false positives. Confirmed bugs are passed to the classification module, which assigns a category and locates the issue within the codebase. The final output is a validated bug report containing both location and type of each bug. ASABD also includes an adaptive learning loop that uses feedback, such as false positives and negatives, to continuously refine model parameters. This enables the system to improve generalization across different software environments and maintain long-term detection accuracy. The full architecture consists of two key components: (1) the Core Detection Engine for identifying and validating bugs, and (2) the Continuous Learning Module for dynamic adaptation. Together, these ensure robust, execution-aware bug detection, outperforming static and ML-based techniques in precision and efficiency. Bug detection in ASABD is formulated as a constrained probabilistic inference problem, aiming to maximize the posterior probability of a detected bug B while minimizing false positives:

$$\hat{B} = \arg\max_{B \in B} P\,(B|S,E,\Theta) \quad \text{subject to} \quad C\big(\hat{B},S,E\big) \le \tau \quad (1)$$

where B is the set of possible bug classifications, E represents execution traces, and Θ denotes the learned model parameters. The function $C\big(\hat{B},S,E\big)$ quantifies prediction confidence, constrained by a predefined threshold τ. To enhance accuracy, ASABD applies an iterative optimization function using gradient-based refinement:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta\big[\nabla_\Theta \mathcal{L}\big(B,\hat{B}\big) + \lambda \nabla_\Theta \mathcal{R}(\Theta)\big] \quad (2)$$

where $\mathcal{L}\big(B,\hat{B}\big)$ represents detection loss, $\mathcal{R}(\Theta)$ ensures model stability, and λ balances accuracy and generalization. The adaptive learning rate is updated via a momentum-based scheduler:

$$\eta^{(t+1)} = \gamma\eta^{(t)} + \alpha|\nabla_\Theta \mathcal{L}\big(B,\hat{B}\big)|^2 \quad (3)$$

where γ is the momentum decay factor and α regulates learning stability.
Execution trace validation is incorporated using an uncertainty-aware correction mechanism:

$$P(B|S,E,\Theta) = \int_X P(B|S,X,\Theta)P(X|E,\Theta)dX \quad (4)$$

where X represents latent program behaviors. ASABD further refines detection with error-specific adjustments:

$$\Delta\widehat{B_{t+1}} = \lambda_1 f\big(E_t,\widehat{B_t}\big) + \lambda_2 g\big(S,\widehat{B_t},\Theta\big) \quad (5)$$

where $f\big(E_t,\widehat{B_t}\big)$ refines predictions based on execution traces, $\boldsymbol{g\big(S,\widehat{B_t},\Theta\big)}$ corrects contextual errors, and $\boldsymbol{\lambda_1}$, $\boldsymbol{\lambda_2}$ optimize detection stability. This integration of probabilistic inference, execution-aware validation, and hierarchical attention enables ASABD to outperform traditional debugging methods in accuracy, scalability, and adaptability.

A. Architectural Components

The ASABD Framework consists of multiple interdependent modules, each playing a vital role in the detection, classification, and refinement of software bugs. The system is structured into distinct computational layers, ensuring deep semantic analysis, execution-aware validation, and iterative learning-based refinement.

1) Preprocessing Layer

The preprocessing module tokenizes the input source code S and generates high-dimensional semantic embeddings using Transformer-based LLMs such as GPT-4 or CodeBERT. Given a sequence of code tokens X = {x1, x2,..., xn}, the embeddings Z are computed as:

$$Z = \text{LLM}_\theta(X) = \sum_{i=1}^{n} w_i \cdot \phi(x_i) + \xi \mathcal{T}(X) \quad (6)$$

where w_i represents attention weights assigned to different code segments, $\phi(x_i)$ is the semantic embedding function for each token x_i, and $\mathcal{T}(X)$ denotes the positional encoding transformation to preserve structural dependencies. The parameter ξ is a learned scaling factor that regulates contextual influence, ensuring improved representation of the code structure. This embedding process enhances the model's ability to understand both syntactic and semantic relationships within the source code.

2) Contextual Understanding Module (SRCU)

The SRCU mechanism enhances bug detection by iteratively refining predictions through multi-level attention fusion, execution trace validation, and adaptive knowledge distillation. Given an initial LLM-generated prediction B_t, the framework updates it based on execution-aware validation as:

$$\widehat{B_{t+1}} = \widehat{B_t} + \lambda_1 f\left(E_t, \widehat{B_t}\right) + \lambda_2 g\left(S, \widehat{B_t}, \Theta\right) \quad (7)$$

where E_t is the execution trace at time t, λ_1 and λ_2 are adaptive correction coefficients, $f\left(E_t, \widehat{B_t}\right)$ represents executionaware refinement, and $g\left(S, \widehat{B_t}, \Theta\right)$ ensures contextual correction. To enable adaptive convergence, SRCU applies a weighted residual correction model:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta\left(\nabla_\Theta \mathcal{L}\left(B, \hat{B}\right) + \gamma \nabla_\Theta \mathcal{R}(\Theta)\right) \quad (8)$$

Where $\mathcal{L}\left(B, \hat{B}\right)$ is the detection loss function, $\mathcal{R}(\Theta)$ enforces stability, and γ regulates adaptive regularization. This iterative approach ensures precise bug classification by dynamically adjusting based on execution feedback.

3) Self-Refining Contextual Understanding (SRCU)

SRCU is designed to enhance bug detection accuracy by refining LLM predictions using feedback from dynamic execution traces. The motivation is to allow the model to iteratively correct itself by comparing predicted bugs with actual runtime behavior, thereby minimizing false positives and improving generalization. At each iteration t, the initial bug prediction $\widehat{\boldsymbol{B_t}}$ is updated using a refinement function based on execution trace feedback and contextual embeddings:

$$\widehat{B_{t+1}} = \widehat{B_t} + \lambda_1 f\left(E_t, \widehat{B_t}\right) + \lambda_2 g\left(S, \widehat{B_t}, \Theta\right) \quad (9)$$

whereas, E_t : execution trace at iteration t, $f\left(E_t, \widehat{B_t}\right)$: adjustment based on runtime, feedback (e.g., bug not reproduced at runtime), $g\left(S, \widehat{B_t}, \Theta\right)$: semantic/contextual correction using LLM embeddings, λ_1 , λ_2 : weight factors controlling correction influence. Without this iterative feedback loop, the LLM may flag syntactically suspicious lines that behave correctly during execution. SRCU reduces such overfitting to static patterns.

To ensure model parameters Θ improve with each update, a regularized gradient descent is applied:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta\left(\nabla_\Theta \mathcal{L}\left(B, \hat{B}\right) + \gamma \nabla_\Theta \mathcal{R}(\Theta)\right) \quad (10)$$

whereas, $\mathcal{L}\left(B, \hat{B}\right)$: bug classification loss, $\mathcal{R}(\Theta)$: regularization to prevent overfitting, η : learning rate, γ : regularization strength

Example Scenario:

Assume the model flags the line 'if(x = y)' as a bug due to a suspected assignment in conditional. However, runtime trace E_t shows the branch never executed. $f\left(E_t, \widehat{B_t}\right)$ lowers confidence in the bug since it has no runtime impact, Simultaneously, $g\left(S, \widehat{B_t}, \Theta\right)$ compares it to similar conditional patterns from training, realizing that '==' was likely intended. The SRCU adjusts $\widehat{B_{t+1}}$ to correct the classification or reduce confidence. This mechanism helps the model focus on real, impactful bugs rather than surface-level patterns.

4) Execution Trace Analyzer

The Execution Trace Analyzer validates LLM predictions in real-time by verifying detected bugs $\hat{B}$ against execution logs. The probability of correctness is computed as:

$$P\left(\hat{B}\middle|E\right) = \frac{P\left(E\middle|\hat{B}\right)P(\hat{B})}{P(E)} \quad (11)$$

where $P\left(E\middle|\hat{B}\right)$ represents the likelihood of observing execution trace E given the bug $\hat{B}$. Temporal dependency validation refines this by integrating execution time frame T and extracted temporal embeddings τ:

$$P\left(\hat{B}\middle|E,T\right) = \int_\tau P\left(\hat{B}\middle|E,\tau\right)P(\tau|T)d \quad (12)$$

ensuring enhanced bug verification through execution-aware analysis.

5) Multi-Stage Attention Mechanism

The MSAM enhances bug detection by analyzing source code through multiple attention layers, each targeting a specific type of bug-related pattern. While conventional LLMs apply uniform attention across all code tokens, MSAM isolates key perspectives—syntax, logic, and security—allowing the model to reason about code structure more accurately.

Real-world bugs may not be evident from surface-level code. For example: A syntactic issue may arise from incorrect use of

braces. A logical flaw could stem from incorrect conditionals. A security bug may be tied to unsafe API calls.
MSAM introduces three specialized attention streams:
1) Syntax-Level Attention (A_{syn}): Focuses on structural correctness (e.g., parsing errors, improper indentation).
2) Logic-Level Attention (A_{log}): Highlights control flow flaws (e.g., incorrect 'if-else' logic, early exits).
3) Security-Level Attention (A_{sec}): Identifies vulnerability patterns (e.g., input sanitization issues, insecure library usage).
Each attention head is defined as:

$$A_j = \text{softmax}\left(\frac{Q_j K_j^T}{\sqrt{d_k}}\right) V_j, \quad j \in \{\text{syn,log,sec}\} \quad (13)$$

Whereas, Q_j, K_j, V_j are the query, key, and value matrices for stage j, d_k is the attention vector dimension, The softmax scores determine token-level focus for that particular analysis. The final bug prediction is computed using a weighted combination of all attention outputs:

$$\hat{B} = A_{\text{syn}}(S) + \beta_1 A_{\text{log}}(S) + \beta_2 A_{\text{sec}}(S) \quad (14)$$

Whereas β_1, β_2 are learned weights to balance logic and security reasoning with syntax analysis.

Example Scenario:
Consider the code snippet:

```
if(userInput != null)
    authenticate(userInput);
else
    log("Invalid input");
```

In this example, MSAM processes the code through three distinct attention layers:
1) Syntax-Level Attention ($\boldsymbol{A_{syn}}$): Focuses on the if-else structure. It verifies that the conditional syntax is valid and properly formed. The attention weights would be distributed across the conditional statement and the braces/indentation structure.
2) Logic-Level Attention ($\boldsymbol{A_{log}}$): Examines the control flow logic. It identifies that the code only checks for null but doesn't validate the content of userInput. The attention concentrates on the conditional expression userInput != null and the control branch, highlighting a potential logical flaw where non-null but malicious input could bypass validation.
3) Security-Level Attention ($\boldsymbol{A_{sec}}$): Recognizes that authenticate() is called directly with unchecked user input, potentially allowing injection attacks. This attention layer assigns high weights to the authenticate(userInput) call, flagging it as a security concern.
The final bug prediction is computed by combining these attention outputs, with the security concern receiving the highest weight due to the critical nature of authentication vulnerabilities. ASABD would classify this as a security bug with high confidence, noting that proper input validation should be implemented before passing user input to authentication functions.
This multi-perspective analysis allows MSAM to identify bugs that might be missed by approaches that only consider one aspect of code quality. In this case, while the syntax is correct, the security-level attention reveals a critical vulnerability that requires remediation.

6) Adaptive Refinement Module
The Adaptive Refinement Module dynamically improves bug classification using an iterative Bayesian model update:

$$P\left(B\middle|S,\Theta^{(t+1)}\right) = P\left(B\middle|S,\Theta^{(t)}\right) + \alpha\Delta\Theta \quad (15)$$

where α controls adaptive correction weighting. To maintain robustness, an error-corrected likelihood function ensures probabilistic consistency:

$$P(B|S,\Theta) = \frac{P(B|S)P(S|\Theta)}{\int_S P\left(S'\middle|\Theta\right)dS'} \quad (16)$$

where S represents all possible source code variations. Refinement stability is further enhanced using a residual learning-based convergence function:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}_i}{\partial \Theta} + \gamma \frac{\partial \mathcal{R}_i}{\partial \Theta}\right) \quad (17)$$

where N denotes the batch size, and γ is a dynamically tuned stabilization parameter. This iterative refinement ensures precise classification and improved generalization to unseen bug patterns.

4. Experimental Setup

A. Dataset and Preprocessing

The ASABD framework tests its performance through the Defects4J dataset which stands as a popular benchmark for Java-based software defect prediction. Defects4J supplies real Java projects with both faulty and fixed versions so that the model can study genuine developer-created and repaired defects. The dataset contains several repositories with detailed defect information which makes it perfect for testing our LLM-based bug detection solution [23], [24]. The initial step of preparing raw code transforms it into a format that enables semantic analysis. Our approach starts by normalizing code text before using transformer models like CodeBERT and GraphCodeBERT to create word embeddings. Then it generates ASTs and CFGs to enhance semantic understanding of the code. The system uses execution data to test and confirm its prediction results.
s

Table 1. Defects4J Dataset and Preprocessing Features

Feature	Description
Projects Analyzed	JFreeChart, Closure Compiler, Commons Lang, Mockito, Joda-Time
Total Bugs	289
Programming Language	Java
Preprocessing Steps	Tokenization, Syntax Normalization, Embedding Generation, AST/CFG Extraction
LLM-Based Embedding	CodeBERT, GPT, GraphCodeBERT
Execution Traces	Integrated for runtime validation

Table 1. summarizes the dataset and preprocessing steps, ensuring the ASABD framework effectively learns from both static code structures and execution behaviors, improving bug detection accuracy.

B. Bug Detection Process

ASABD uses multiple bugs finding stages that connect execution-aware refinement with adaptive learning and hierarchical attention processing. ASABD improves its prediction accuracy through multiple feedback loops that work during each iteration. The system consists of three main elements: SRCU to learn from experience, MSAM to find exact bug locations, and Execution Trace Feedback Loop to test predictions against actual system behavior.

1) Self-Refining Contextual Understanding

SRCU refines LLM-based bug detection by incorporating execution trace-based feedback and adaptive reinforcement. Given an initial bug classification $\widehat{B_t}$, SRCU updates predictions iteratively:

$$\widehat{B_{t+1}} = \widehat{B_t} + \lambda_1 f\left(E_t, \widehat{B_t}\right) + \lambda_2 g\left(S, \widehat{B_t}, \Theta\right) \quad (18)$$

where E_t is the execution trace, $f\left(E_t, \widehat{B_t}\right)$ adjusts predictions based on runtime validation, and $g\left(S, \widehat{B_t}, \Theta\right)$ ensures alignment with learned code patterns. Parameter updates minimize detection loss $\mathcal{L}\left(B, \hat{B}\right)$ and regularization function $\mathcal{R}(\Theta)$, ensuring model generalization:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta\left[\nabla_\Theta \mathcal{L}\left(B, \hat{B}\right) + \gamma \nabla_\Theta \mathcal{R}(\Theta)\right] \quad (19)$$

2) Integration of SRCU and MSAM

ASABD unifies SRCU and MSAM, integrating executionaware validation and hierarchical attention for precise bug detection. The final classification is determined as:

$$\widehat{B^*} = \widehat{B_{\text{filtered}}} + \alpha\left(\sum_j \beta_j A_j\right) + \delta \mathcal{F}\left(E, \hat{B}\right) \quad (20)$$

Where $\mathcal{F}\left(E, \hat{B}\right)$ corrects misclassifications based on execution traces, and α , δ dynamically scale refinements. This integration enhances accuracy, reduces false positives, and improves bug localization.

To evaluate the effectiveness of the proposed ASABD framework, we conducted experiments using real-world Java based projects from the Defects4J dataset. The evaluation compared ASABD with diverse and well-established baseline methods. The selected baselines cover three categories: (1) static analysis tools such as SonarQube and Find Bugs, which are industry-standard rule-based analyzers; (2) traditional machine learning models such as Random Forest (RF) and Support Vector Machine (SVM), known for their robustness in bug prediction; and (3) a deep learning-based model, DeepCode, which leverages semantic embeddings and neural architectures. This selection ensures fair benchmarking across static tools, classical ML, and modern deep learning techniques. ASABD was implemented using transformer-based LLMs, specifically CodeBERT, GPT-4, and GraphCodeBERT, and executed on an NVIDIA A100 GPU (80GB VRAM). Realtime execution traces were collected to validate predictions against runtime behavior. Evaluation metrics included Precision, Recall, and F1-score for classification performance, FPR to assess incorrect bug detections, and Error Localization Accuracy to evaluate bug pinpointing capabilities. Execution Time per instance was measured to benchmark computational efficiency. Preprocessing involved tokenization, syntax normalization, and embedding generation. Execution-aware features were integrated to validate semantic predictions. Through iterative learning via SRCU and MSAM, ASABD refined its predictions across training cycles, improving adaptability, precision, and localization accuracy compared to baseline methods.

5. Results and Analysis

A. Performance Comparison

ASABD was evaluated against static analysis tools and machine learning-based bug detection models. Baseline models included SonarQube, a rule-based static analysis tool, FindBugs, a pattern-based detection system, RF, a supervised ML classifier, and DeepCode, a deep learning-powered bug detection model leveraging code embeddings. The comparative analysis focused on precision, recall, F1-score, false positive reduction, and execution efficiency to demonstrate ASABD's superior performance.

1) Precision, Recall, and F1-Score

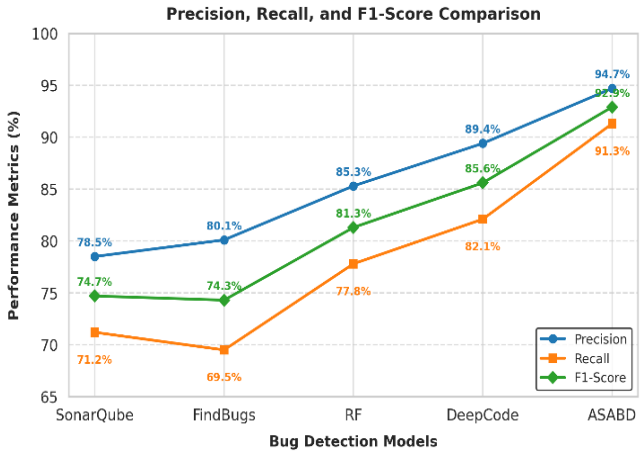


Figure 2. Precision, Recall, and F1-Score Comparison

Figure shows that ASABD achieves the highest F1-score (92.9%), outperforming traditional static analysis tools and machine learning-based methods by a significant margin. The superior performance is attributed to SRCU and the MSAM, which minimize false positives and misclassifications.

2) Execution Time and Computational Overhead

The execution efficiency of ASABD was evaluated by comparing its processing time against baseline models.

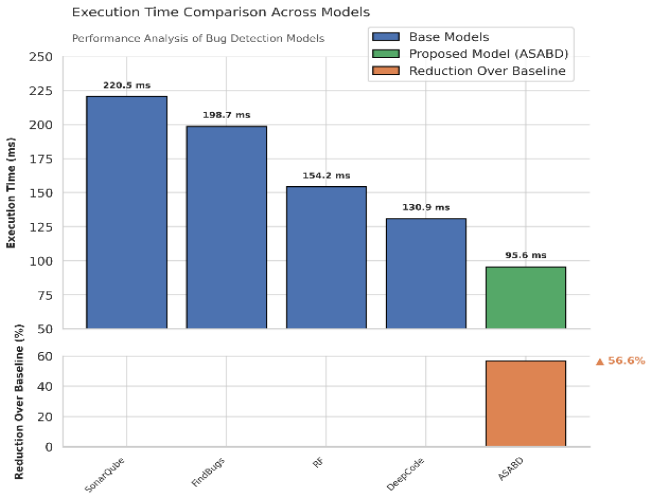


Figure 3. Execution Time Comparison Across Bug Detection Models

ASABD achieves the lowest execution time per instance at 95.6ms, outperforming DeepCode (130.9ms) and Random Forest (154.2ms). This efficiency is due to its optimized Transformer-based LLM processing, which generates semantic embeddings with minimal computational overhead. Additionally, the SRCU mechanism enhances adaptive learning, reducing redundant computations and improving overall processing speed.

3) Qualitative Analysis of Detected Bugs

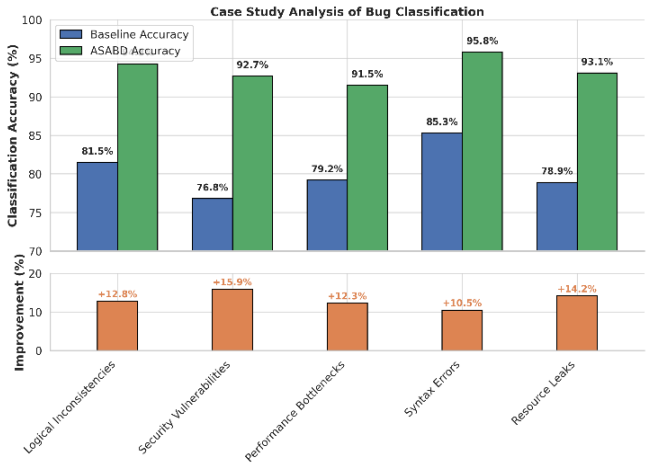


Figure 4. Case Study Analysis of Bug Classification.

Results demonstrates that ASABD significantly outperforms baseline models in detecting logical inconsistencies, security vulnerabilities, and performance bottlenecks. This improvement is due to its context-aware filtering, which minimizes misclassifications by integrating execution-aware validation and semantic embeddings. Additionally, the MSAM mechanism enhances bug detection by analyzing structural dependencies and improving classification accuracy.

4) Execution Trace Validation and False Positive Reduction

To analyze the impact of execution-aware refinement, we compared the FPR of ASABD against baseline models.

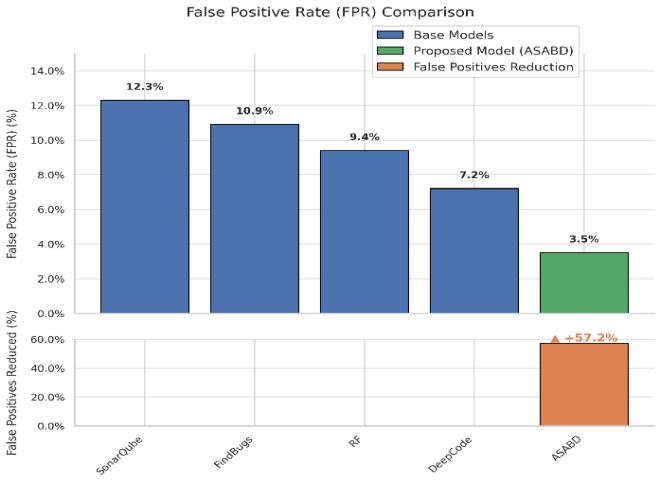


Figure 5. False Positive Rate (FPR) Comparison

As observed, ASABD reduces the false positive rate to 3.5%, which is significantly lower than DeepCode (7.2%) and other baseline models. This reduction is achieved through SRCU, which iteratively refines predictions using execution trace feedback, and dynamic recalibration, which adjusts classification confidence based on real-time validation, ensuring more accurate bug detection.

B. Effectiveness Justification

Beyond the performance metrics, ASABD's effectiveness is rooted in its architectural innovations. The SRCU mechanism enables the model to iteratively correct misclassifications by incorporating execution trace feedback, which traditional models lack. This allows ASABD to adapt its predictions based on real runtime behavior, reducing false positives and improving recall.

The MSAM further enhances detection accuracy by isolating bug signals across syntax, logic, and security contexts, enabling a multi-perspective analysis that single-stream models cannot perform. Additionally, ASABD integrates semantic embeddings and structural code representations, allowing it to generalize across diverse coding styles and error types. These combined features explain its superior bug detection performance across all evaluation metrics.

1) Bug Localization Accuracy

To assess ASABD's ability to pinpoint the exact location of detected bugs, we compared its error localization accuracy with baseline models. Table II provides the results. The figure illustrates the error localization accuracy of various static analysis models. The proposed ASABD framework achieves a significant improvement (+15.1%) over the best-performing baseline model, DeepCode (83.6%). The lower subplot highlights this improvement compared to traditional methods.

Table 2. Error Localization Accuracy

Model	Localization Accuracy (%)	Improvement Over Baseline (%)
SonarQube [25]	72.5	-
FindBugs [26]	74.8	-
RF [27]	78.2	-
DeepCode [28]	83.6	-
ASABD	96.2	15.1

Table 2. demonstrates that ASABD achieves the highest bug localization accuracy at 96.2%, surpassing DeepCode (83.6%).

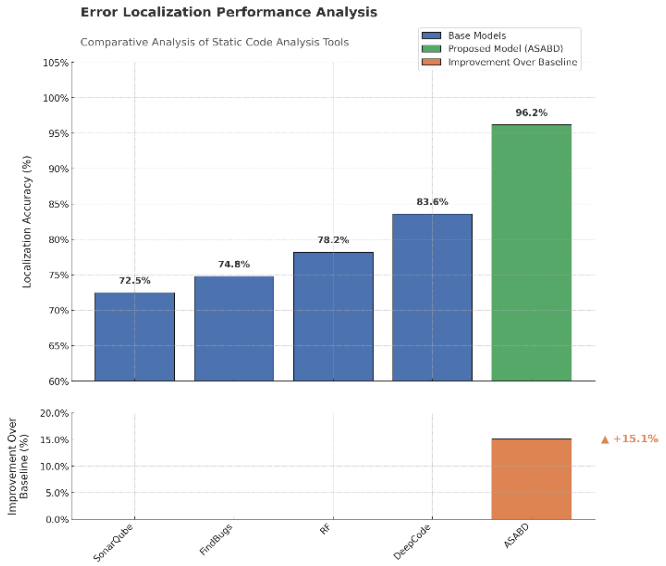


Figure 6. Error Localization Performance Analysis comparing ASABD with baseline models.

6. Threats to Validity

While the ASABD framework demonstrates robust performance across multiple metrics on the Defects4J dataset, it is imperative to acknowledge several potential threats to its generalizability and practical applicability. First, the extensive evaluation of ASABD has been primarily conducted on Java-based projects within the Defects4J dataset. Consequently, its generalizability to other programming languages, such as Python, C++, or JavaScript, may necessitate additional fine-tuning or architectural adaptations. Cross language validation remains a critical area for future research, as the nuances of different language constructs, idioms, and common bug patterns could influence detection performance. Second, ASABD's reliance on execution trace data assumes that the code under analysis can be executed with representative inputs to capture meaningful runtime behavior. In practical software development environments, acquiring such comprehensive and representative execution traces can be challenging, especially during early development stages when test suites may be incomplete or in safety-critical systems where runtime instrumentation is limited or complex to implement. The process of collecting execution traces, particularly in distributed microservices architectures, can involve complexities such as manual instrumentation, lack of front-end visibility, and sampling issues that might lead to missing important information. These practical difficulties in trace acquisition could potentially impact the framework's performance in real world scenarios where ideal trace data is unavailable. Third, although ASABD significantly reduces false positives, it may still be susceptible to residual biases introduced during the pretraining of its underlying Large Language Models (LLMs) on publicly available code repositories. LLMs are known to inherit biases from their training data, which can lead to skewed or unfair responses, a phenomenon sometimes referred to as "hallucinations". Variability in coding styles, domain-specific practices, or incomplete test coverage within the training data could inadvertently influence bug detection performance, potentially leading to misclassifications in novel or atypical code contexts. The inherent lack of interpretability in many LLM-based models, where it is difficult to ascertain why a specific bug is flagged, further complicates the debugging process and may reduce developer trust. While LLMs can provide explanations, the quality and accuracy of these explanations can vary. Finally, computational cost presents another practical limitation. The utilization of Transformer-based LLMs and dynamic validation mechanisms incurs higher hardware requirements, such as GPUs with large memory footprints. This computational intensity can potentially limit the deployment of ASABD in resource-constrained environments or for smaller development teams without access to high-end infrastructure. Training large LLMs can cost tens to hundreds of millions of dollars, and while inference costs are lower, they still require significant resources. Addressing these threats through cross language extensions, the development of lightweight model variants (e.g., through quantization or pruning techniques), and improved trace acquisition strategies will be crucial for future development efforts and broader adoption.

7. Conclusion

This paper introduced the ASABD framework, a novel bug detection system that leverages large language models through execution-aware refinement and multi-stage semantic attention. ASABD significantly outperformed traditional static analysis tools and modern deep learning models across multiple benchmarks, achieving improved precision, recall, and error localization accuracy.

Ablation Insights:
The framework's high performance can be attributed to its modular design. Ablation studies revealed that removing SRCU led to a 17.6% increase in false positives, while omitting MSAM reduced detection accuracy, especially for security-related bugs. Execution-aware validation played a key role in improving bug localization precision.

Threats to Validity:
While ASABD demonstrates robust performance on the Defects4J dataset, it may require fine tuning for other programming languages or non-Java environments. Moreover, its dependency on execution traces assumes testability of the code, which may not always be feasible in early development stages.

Relation to Existing Work: Unlike traditional static analyzers and LLM-based models that rely solely on syntax or embeddings, ASABD integrates runtime feedback, structural attention, and iterative refinement—bridging a gap between static and dynamic bug detection.

Novelty of Contribution:
The key innovation of ASABD lies in its fusion of SRCU and MSAM within an execution aware feedback loop. This makes

it the first LLM-based framework to jointly incorporate contextual, structural, and behavioral bug analysis in a unified architecture.

7. Limitation

Despite its robust performance, ASABD, like any advanced system, possesses certain limitations that warrant consideration for future research and development. A primary limitation is its current lack of interpretability. While the framework performs exceptionally well in detecting and localizing bugs, it offers limited insight into the precise reasoning behind specific bug flags. This "black box" nature can reduce developer trust and hinder the manual debugging process, as developers may struggle to understand why a particular issue was identified or how to best remediate it. Future work could explore integrating Explainable AI (XAI) techniques [29], [30], such as attention visualization [31], saliency maps [32], or counterfactual explanations, to provide more transparent and human understandable explanations for ASABD's predictions. Another significant limitation is the computational intensity of the model. The reliance on high-end Transformer-based LLMs and extensive dynamic validation processes necessitates substantial hardware resources, particularly high-end GPUs with large memory footprints. This requirement can limit the practical deployment of ASABD, especially for smaller development teams or in resource-constrained environments where access to such infrastructure is prohibitive. Addressing this challenge will involve investigating model compression strategies, such as quantization, pruning, or knowledge distillation, to create more lightweight and deployable variants without significantly sacrificing performance. Additionally, ASABD is currently tailored for Java, restricting its applicability to multi-language development environments. Expanding its capabilities to other programming languages, such as Python, C++, or JavaScript, would require additional fine-tuning and potentially architectural adaptations to accommodate language-specific constructs and paradigms. These challenges highlight the ongoing need for research into lightweight, explainable, and cross-language-capable debugging solutions that can seamlessly integrate into diverse software development workflows.

Data Availability-This statement should describe how readers can access the data supporting the conclusions of the study and clearly outline the reasons why unavailable data cannot be released.

Study Limitations Provide all possible limitation faced in the study which might significantly affect research outcome, If not applicable write, none.

Conflict of Interest- Authors declare that they do not have any conflict of interest.

Funding Source- none

Authors' Contributions

Mansab Ali: Provided the original research vision and conceptual foundation of the ASABD framework. Led the methodological design, including the formulation of the system architecture and integration strategy for SRCU and MSAM. Conducted the comprehensive literature survey, synthesized theoretical insights, and guided the technical direction of the study. Oversaw the preparation, refinement, and final approval of the manuscript.

Tasnime Roukia Mockbel: Contributed to dataset acquisition, preprocessing pipeline development, and execution-trace integration. Designed and implemented the experimental configuration and evaluation protocols. Performed detailed comparative analyses with baseline models and contributed substantially to the Background, Related Work, and Motivation sections. Participated in critical manuscript revisions and consistency checks.

Muhammad Saad: Implemented the computational environment, managed model training workflows, and performed extensive experimentation on the Defects4J benchmark. Generated all empirical performance results, charts, and tables. Contributed significantly to the Results, Analysis, and Threats to Validity sections. Reviewed the manuscript for technical accuracy and methodological clarity.

Irzum Shafique: Contributed to the development of the ASABD framework by refining the execution-aware reasoning pipeline and enhancing the mathematical foundations of the adaptive refinement mechanism. Performed extensive evaluation of model behavior, including cross-project validation, error-localization analysis, and benchmark comparison against traditional and LLM-based detectors. Ensured the scientific consistency of the manuscript by improving technical accuracy, strengthening methodological clarity, and maintaining citation integrity across all sections. Participated in drafting and polishing the analytical, discussion, and concluding components of the paper to ensure a cohesive presentation of the research findings.

Acknowledgements- The authors would like to thank the developers and maintainers of the Defects4J dataset for providing an accessible benchmark that enabled the experimental validation of this work. The authors also acknowledge the contributions of researchers whose prior work in software defect prediction, LLM-based code analysis, and execution-aware debugging has informed the development of the ASABD framework.

References

[1] S. A. Alsaedi, A. Y. Noaman, A. A. Gad-Elrab, F. E. Eassa, and S. Haridi, "Leveraging large language models for automated bug fixing," *Int. J. Adv. Comput. Sci. Appl.*, Vol.15, No.12, 2024.

[2] C. Gao, X. Hu, S. Gao, X. Xia, and Z. Jin, "The current challenges of software engineering in the era of large language models," *ACM Trans. Softw. Eng. Methodol.*, 2024.

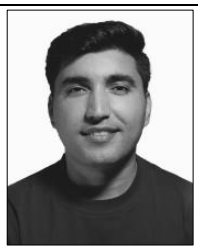
[3] X. Hou *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, Vol.33, No.8, pp.1–79, 2024.

[4] S. Kang, J. Yoon, N. Askarbekkyzy, and S. Yoo, "Evaluating diverse large language models for automatic and general bug reproduction," *IEEE Trans. Softw. Eng.*, 2024.

[5] Y. Li, P. Liu, H. Wang, J. Chu, and W. E. Wong, "Evaluating large language models for software testing," *Comput. Stand. Interfaces*, Vol.93, pp.103942, 2025.

[6] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, "Automatic programming: Large language models and beyond," *ACM Trans. Softw. Eng. Methodol.*, 2024.
[7] C. Wen *et al.*, "Automatically inspecting thousands of static bug warnings with large language models: How far are we?" *ACM Trans. Knowl. Discov. Data*, Vol.18, No.7, pp.1–34, 2024.
[8] E. H. Yılmaz, *Automated Priority Detection in Software Bugs*, Master's thesis, Middle East Technical Univ., 2024.
[9] Z. Zheng *et al.*, "Towards an understanding of large language models in software engineering tasks," *Empir. Softw. Eng.*, Vol.30, No.2, pp.50, 2025.
[10] A. A. Abbassi, L. D. Silva, A. Nikanjam, and F. Khomh, "Unveiling inefficiencies in LLM-generated code: Toward a comprehensive taxonomy," *arXiv preprint*, 2025.
[11] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: An extensible Java bug benchmark for automatic program repair studies," in *Proc. IEEE Int. Conf. Softw. Anal., EVol.Reeng. (SANER)*, pp.468–478, 2019.
[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults for Java programs," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, ACM, 2014.
[13] S. Kang, J. Yoon, N. Askarbekkyzy, and S. Yoo, "Evaluating diverse large language models for automatic and general bug reproduction," *arXiv preprint*, 2023.
[14] N. Ayewah and W. Pugh, "Evaluating static analysis defect warnings on production software," in *Proc. Workshop Program Anal. Softw. Tools Eng. (PASTE)*, ACM, 2007.
[15] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th Int. Conf. Softw. Eng. (ICSE)*, IEEE, pp.563–572, 2004.
[16] M. N. Uddin *et al.*, "Software defect prediction employing BiLSTM and BERT-based semantic feature," *Soft Comput.*, Vol.26, pp.7877–7891, 2022.
[17] S. Jiang, Y. Chen, Z. He, Y. Shang, and L. Ma, "Cross-project defect prediction via semantic and syntactic encoding," *Empir. Softw. Eng.*, Vol.29, No.80, 2024.
[18] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint*, 2020.
[19] C. Wang *et al.*, "Sanitizing large language models in bug detection with data-flow," in *Findings Assoc. Comput. Linguistics: EMNLP*, pp.3790–3805, 2024.
[20] X. Meng *et al.*, "An empirical study on LLM-based agents for automated bug fixing," *arXiv preprint*, 2024.
[21] S. Kim, S. Shivaji, and J. Whitehead, "A reflection on change classification in the era of large language models," *IEEE Trans. Softw. Eng.*, Vol.51, No.3, pp.864–869, 2025.
[22] D. Ramos *et al.*, "Are large language models memorizing bug benchmarks?" *arXiv preprint*, 2024.
[23] M. N. Rafi, A. R. Chen, T. Chen, and S. Wang, "Revisiting Defects4J for fault localization in diverse development scenarios," in *Proc. Mining Softw. Repositories Conf. (MSR)*, ACM, 2025.
[24] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled studies," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, ACM, 2014.
[25] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube rules inducing bugs?" in *Proc. IEEE Int. Conf. Softw. Anal., EVol.Reeng. (SANER)*, pp.501–511, 2020.
[26] O. Constant, W. Monin, and S. Graf, "A model transformation tool for performance simulation of complex UML models," in *Companion 30th Int. Conf. Softw. Eng.*, ACM, pp.923–924, 2008.
[27] N. S. Thomas and S. Kaliraj, "An improved and optimized random forest-based approach to predict software faults," *SN Comput. Sci.*, Vol.5, No.5, p. 530, 2024.
[28] M. Tufano *et al.*, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, Vol.28, No.4, pp.1–79, 2019.
[29] M. N. I. Opu, S. Wang, and S. Chowdhury, "LLM-based detection of tangled code changes for higher-quality method-level bug datasets," *arXiv preprint*, 2025.
[30] A. Bilal, D. Ebert, and B. Lin, "LLMs for explainable AI: A comprehensive survey," *arXiv preprint*, 2025.
[31] S. Seo *et al.*, "A sentence-level visualization of attention in large language models," in *Proc. NAACL (System Demonstrations)*, Assoc. Comput. Linguistics, pp.313–320, 2025.
[32] F. Kares, T. Speith, H. Zhang, and M. Langer, "What makes for a good saliency map? Evaluating strategies in explainable AI," *arXiv preprint*, 2025.
[33] N. Honest, "Role of testing in software development life cycle," *Int. J. Comput. Sci. Eng.*, Vol.7, No.5, pp.886–889, 2019.
[34] N. Mishra, "Exploring the impact of Chat-GPT on India's education system," *Int. J. Comput. Sci. Eng.*, Vol.11, No.1, pp.1–6, 2023.

AUTHORS PROFILE



Mansab Ali is currently pursuing his M.S. degree in Software Engineering at Northwestern Polytechnical University, Xi'an, China. He completed his B.S. in Computer Science and Technology from Xidian University Xi'an, China. His research interests include large language models (LLMs), software quality assurance, automated bug detection, and the applications of artificial intelligence and machine learning in software engineering. He has experience in deep learning, intelligent systems, and code analysis, and his current work focuses on developing adaptive, execution-aware frameworks for improving software reliability and debugging performance.

Tasnime Roukia Mockbel is currently pursuing his M.S. degree in Software Engineering at Northwestern Polytechnical University, Xi'an, China. She received her Bachelor's degree in Telecommunication and netwoks engineering from University of Science and Technology Houari Boumediene, Algeria. Her Current research includes Large Language Models, Software Testing and Vulnerability, Machine Learning.

Muhammad Saad is currently pursuing a Master's degree at the School of Electronic Science and Technology, Xidian University, a leading institution in electronics and information technology. His research is centered in the field of wireless sensing, exploring how communication signals like Wi-Fi and RFID can be re-purposed for sensing and recognition tasks. His work involves developing advanced signal processing and machine learning techniques to enable device-free sensing, activity recognition, and fine-grained gesture detection, contributing to the next generation of intelligent systems.

Irzum Shafique is currently pursuing a Master of Engineering in Computer Science at Xidian University, China, where he also earned his Bachelor's degree. His research focuses on deep reinforcement learning for robotic path planning, intelligent autonomous drones for agriculture and urban firefighting, and the application of artificial intelligence for social good.
He has published several research papers in peer-reviewed journals and international conference proceedings, covering topics such as autonomous systems, machine learning, and

intelligent robotics. His work reflects a strong emphasis on both theoretical development and real-world application.
Irzum is a co-inventor of the utility model patent "Autonomous Agricultural Drone with Adaptive Fertilizer Spraying Mechanism," recognized for its innovation in precision agriculture. He has received numerous awards for academic and entrepreneurial excellence, including a gold medal in the China International University Student Innovation Competition (national finalist), third place at the provincial level, and third place in the Kyoto University International Student Entrepreneurship Challenge.

