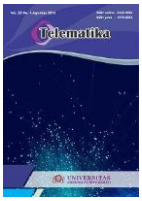

Available online at :
http://ejournal.amikompurwokerto.ac.id/index.php/telematika/

Telematika

Accreditated SINTA "2" Kemenristek/BRIN, No. 85/M/KPT/2020

UNIVERSITAS AMIKOM PURWOKERTO

Enhancing the GLANCE Framework for Line-Level Defect Prediction: An Empirical Study of Semantically-Aware Metrics and Non-Linear Classifiers

Zahid Mujaddid[1], Ema Utami[2]

[1,2] Magister of Informatics, Universitas Amikom Yogyakarta, Yogyakarta, Indonesia

ARTICLE INFO

History of the article:
Received April 10, 2025
Revised May 15, 2025
Accepted August 27, 2025

Keywords:
Software quality assurance
Line-level defect prediction
Heuristic-based models
Defect localization
GLANCE

Correspondence:
E-mail: ema.u@amikom.ac.id

ABSTRACT

Line-level defect prediction (LLDP) is critical for reducing software maintenance costs, yet its industrial adoption is often hindered by high false alarm rates that erode developer trust. While the state-of-the-art GLANCE-LR framework offers a lightweight solution, it relies on linear classifiers and purely syntactic heuristics, failing to capture the non-linear defect patterns and semantic risks associated with complex code constructs. To bridge the gap between operational efficiency and semantic awareness, this paper proposes GLANCE++, an enhanced framework that integrates a non-linear LightGBM classifier for refined file-level filtering and introduces three semantically-aware line metrics: Cognitive Complexity Score (CCS), API-Weighted Number of Function Calls (AW-NFC), and Variable-Write Count (VWC). These metrics shift the prediction paradigm from counting tokens to modeling "code risk." Empirical evaluation on 19 open-source Java projects (142 releases) reveals that while the non-linear file classifier yields marginal gains, the semantic line-level metrics achieve statistically significant improvements in precision and False Alarm Rate (FAR). However, this increased selectivity introduces a trade-off, resulting in reduced recall compared to the baseline. Our findings demonstrate that improving the semantic intelligence of heuristics yields far greater impact than increasing model complexity. This suggests that future LLDP research should prioritize theoretically grounded risk metrics over computationally expensive deep learning architectures to ensure practical deployment in real-time CI/CD pipelines.

1. INTRODUCTION

The economic and operational cost of software defects remains a persistent challenge in the software industry. Defects discovered late in the development lifecycle, or worse, by end-users in production, can be orders of magnitude more expensive to remediate than those identified early. Consequently, Software Quality Assurance (SQA) has become a cornerstone of modern software engineering, consuming a significant portion of total development effort (Pachouly, Ahirrao, Kotecha, Selvachandran, & Abraham, 2022). A primary goal of SQA is to efficiently and accurately locate buggy code. To this end, automated Software Defect Prediction (SDP) has emerged as a key mitigation strategy, employing statistical and machine learning models to identify defect-prone software artifacts before they cause significant issues (Wan et al., 2020).

Historically, SDP has operated at coarse granularities, such as the package (Kamei et al., 2010), module (Kamei, Monden, Matsumoto, Kakimoto, & Matsumoto, 2007), file (Pascarella, Palomba, & Bacchelli, 2019), or method (Hata, Mizuno, & Kikuno, 2012) level. While useful for high-level resource allocation, these predictions lack the precision required for developers to take direct action. A prediction that a 10,000-line file is "buggy" provides little guidance, especially when empirical evidence suggests that fewer than 3% of the lines within such a file may actually contain a defect (Wattanakriengkrai, Thongtanunam, Tantithamthavorn, Hata, & Matsumoto, 2022). This has motivated a critical shift in the

field toward fine-grained, Line-Level Defect Prediction (LLDP), which aims to pinpoint the exact lines of code most likely to be defective, thereby producing a ranked, fine-grained inspection list that can directly guide code review and remediation efforts.

In response to this demand, a diverse array of LLDP approaches has emerged. Deep learning such as DeepLineDP (Pornprasit & Tantithamthavorn, 2023) and BAFLineDP (Shaojian Qiu, Huang, Luo, Kuang, & Luo, 2024), and Transformer-based models such as LineVul (Fu & Tantithamthavorn, 2022) and Bugsplorer (Mahbub & Rahman, 2024), have set new performance benchmarks by automatically learning complex semantic features from code tokens. Similarly, Syntax-aware models like SyntaxLineDP (Zhu, Huang, Chen, Wang, & Zheng, 2023) and Graph-based approaches like LineFlowDP (Yang, Zhong, Zeng, Xiao, & Zheng, 2024) leverage Abstract Syntax Trees (AST) to capture structural dependencies. However, these gains come at a steep cost: these models are computationally intensive, requiring significant GPU resources for training and inference, and often operate as "black boxes," limiting their practical deployment in real-time Continuous Integration (CI) pipelines where speed and interpretability are paramount.

Addressing this efficiency gap, the GLANCE-LR framework (Guo et al., 2023) challenged the prevailing assumption that model complexity is a prerequisite for accuracy. By using a simple Logistic Regression classifier and syntactic heuristics (Number of Tokens, Number of Function Calls), GLANCE-LR achieved performance comparable to deep learning models at a fraction of the computational cost. However, this lightweight approach suffers from a fundamental "Semantic Gap." First, its reliance on a linear classifier (Logistic Regression) fails to capture the non-linear interactions between code features that characterize complex defects. Second, its line-level heuristics are purely syntactic; they treat all function calls equally and ignore the cognitive load and semantic risk associated with different code constructs. For instance, a complex nested loop modifying global state is inherently riskier than a simple print statement, yet GLANCE-LR assigns them equal weight if their token counts are similar.

In this paper, we propose GLANCE++, a framework that bridges this gap by injecting semantic intelligence into the lightweight heuristic paradigm. We argue that the limitations of current heuristic models are not due to the lack of deep neural networks, but rather the lack of theoretically grounded metrics. Consequently, GLANCE++ introduces three major methodological innovations:

We replace the linear Logistic Regression with LightGBM, a Gradient Boosting Decision Tree (GBDT) framework. Unlike Random Forests, which grow trees level-wise, LightGBM uses leaf-wise growth with Gradient-based One-Side Sampling (GOSS), allowing it to capture complex non-linear feature interactions with significantly lower memory usage and faster training times than both traditional ensemble methods and deep learning approaches (Ke et al., 2017; Mienye & Sun, 2022; Oueslati, Ouertani, Amdouni, & Manita, 2025).

We replace arbitrary syntactic counting with metrics rooted in software theory. We introduce the Cognitive Complexity Score (CCS), based on Cognitive Load Theory (Segalotto, Bolzan, & Farias, 2023), to quantify the mental effort required to understand code flow. We incorporate Variable-Write Count (VWC), derived from Data Flow Analysis principles (Shaoming Qiu, E, He, & Liu, 2025), to track state-mutation risks. Finally, we implement API-Weighted Number of Function Calls (AW-NFC), which weights invocations based on empirically established API misuse patterns (Li, Jiang, Benton, Xiong, & Zhang, 2021), distinguishing high-risk I/O operations from benign calls. We combine these enhancements into a unified two-stage pipeline that maintains the "lightweight" philosophy of the original GLANCE while significantly improving its semantic awareness.

In summary, this paper aims to redefine heuristic-based defect prediction by moving beyond simple syntactic counting toward a model of 'code risk' grounded in software theory. Accordingly, the primary contributions of this study are summarized as follows: An Enhanced File-Level Classifier (GLANCE+File): We demonstrate that upgrading from a linear solver to a gradient boosting framework (LightGBM) improves the modeling of complex defect signals while retaining high computational efficiency. Semantically-Aware Line Metrics (GLANCE+Line): We define and validate a new set of heuristic metrics (CCS, AW-NFC, VWC) that shift the focus of LLDP from "code size" to "code risk," providing a stronger theoretical foundation for heuristic defect prediction. Comprehensive Empirical Evaluation: We evaluate GLANCE++ on the BugDet dataset (19 projects, 142 releases). The results show that our semantic enhancements achieve statistically significant improvements in precision-oriented metrics (reducing False Alarm Rates), validating the hypothesis that semantic risk assessment is a more effective driver of performance than mere model complexity. To evaluate the efficacy of our contributions, we conduct a rigorous empirical study guided by the following Research Questions (RQs):

(1) RQ1: How does replacing the Logistic Regression classifier with LightGBM (GLANCE+File) impact performance?
(2) RQ2: How do semantically-aware line metrics (GLANCE+Line) improve over syntactic heuristics?

(3) RQ3: What is the combined effect (GLANCE++), and how does it advance practical LLDP?

The remainder of this paper is organized as follows. Section II describes the research methodology, including the baseline model (GLANCE-LR), the proposed enhancements for file- and line-level prediction, the dataset used, the evaluation protocol, and the performance metrics employed. Section III presents the empirical findings, offering a detailed performance comparison across models, statistical significance analysis, and a critical discussion of trade-offs, implications, and threats to validity. Section IV concludes the paper by summarizing the key contributions and highlighting potential directions for future research.

2. METHODOLOGY

The primary goal of our methodology is to provide a transparent and reproducible framework for evaluating the proposed enhancements to the GLANCE-LR model. We ensure a fair comparison by building upon the original experimental setup defined by (Guo et al., 2023). This section details our baseline model, the proposed enhancements, the dataset used, and our evaluation protocol, ensuring full reproducibility.

2.1. Baseline Model: GLANCE-LR

Our work uses GLANCE-LR as the state-of-the-art baseline for heuristic-based line-level defect prediction. As illustrated in Figure 1, GLANCE-LR is a two-stage framework designed to efficiently identify and prioritize potentially defective code lines while minimizing computational overhead (Guo et al., 2023).

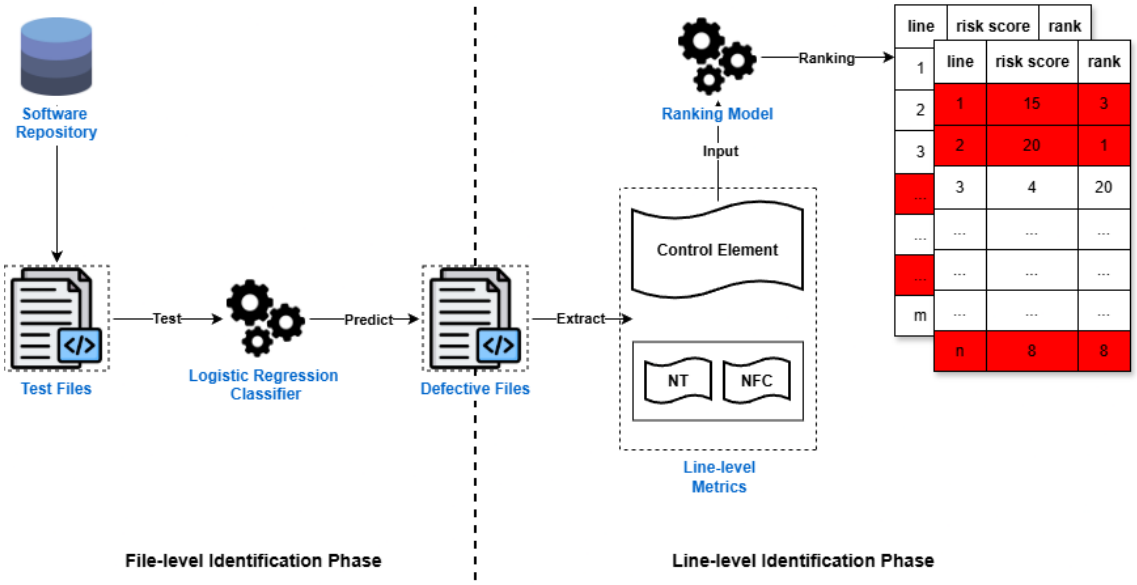

Figure 1. The two-phase architecture of GLANCE-LR framework

The first stage, File-Level Defect Prediction, aims to reduce the search space by identifying files that are likely to contain defects. Each source file is transformed into a feature vector using a Bag-of-Words (BoW) representation, where each unique code token is treated as a distinct feature. A Logistic Regression classifier is trained on historical data from a previous release to predict whether a file in a subsequent release is "defective" or "clean." This coarse-grained classification serves as an initial filter, eliminating a large proportion of clean files and focusing inspection effort on a smaller set of high-risk candidates.

The second stage, Line-Level Ranking, performs fine-grained prioritization within each predicted defective file. For every line in such a file, a DefectPronenessScore is computed using two syntactic metrics: the Number of Tokens (NT) and the Number of Function Calls (NFC). As shown in the first equation, the scoring function reflects the intuition that longer lines with more function calls are more complex and thus more likely to be error-prone. To further enhance the model's sensitivity to logical errors, any line containing a control element (e.g., `if`, `for`, `while`, `switch`, `return`) is given a final boost in ranking, effectively moving these lines to the top of the list. This rule-based prioritization ensures that developers are guided toward code constructs that are historically associated with higher defect rates

$$DefectPronenessScore = NT \cdot (NFC + 1) \quad (1)$$

By combining a lightweight machine learning classifier with simple yet effective syntactic heuristics, GLANCE-LR achieves a compelling balance between performance and efficiency, serving as a

robust benchmark against which our enhanced models, namely GLANCE+File, GLANCE+Line, and GLANCE++ are evaluated.

2.2. Proposed Enhancements: GLANCE++

Our GLANCE++ framework introduces targeted improvements to both stages of the GLANCE-LR pipeline. These enhancements can be applied independently or in combination, resulting in three model variants: GLANCE+File, GLANCE+Line, and the fully integrated GLANCE++.

2.3. GLANCE+File: An Enhanced File-Level Classifier

To overcome the limitations of the linear classifier in the baseline GLANCE-LR model, GLANCE+File replaces Logistic Regression with LightGBM (Light Gradient Boosting Machine), a gradient boosting framework based on tree-based learning algorithms. This enhancement is motivated by the fact that buggy code often results from subtle combinations of tokens and structural features that cannot be modeled effectively using linear decision boundaries.

A primary advantage of LightGBM is its ability to model complex feature interactions. As a gradient boosting decision tree (GBDT) model, it inherently captures non-linear relationships and dependencies among input features (Ke et al., 2017). In the context of defect prediction, this is critical, as defect-inducing changes often arise from the interaction of multiple code characteristics rather than isolated metrics (Assim, Obeidat, & Hammad, 2020). LightGBM also offers significant gains in efficiency and scalability. It employs two key optimizations: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) (Ke et al., 2017; Mienye & Sun, 2022; Oueslati et al., 2025). GOSS reduces computational load by focusing on data instances with large gradients, while EFB reduces dimensionality by grouping mutually exclusive sparse features.

A critical aspect of this enhancement is also the refinement in lexical feature representation. While the original GLANCE-LR uses CountVectorizer to generate a Bag-of-Words (BoW) representation, which treats all tokens equally based on their frequency, GLANCE+File employs TfidfVectorizer. This method computes the Term Frequency-Inverse Document Frequency (TF-IDF) score for each token, which not only considers how frequently a token appears in a file (term frequency) but also down-weights tokens that appear frequently across many files in the project (inverse document frequency) (Sharma & Singh, 2025).

In the GLANCE+File variant, the file-level prediction stage is executed using a LightGBM classifier trained on the TF-IDF features generated by TfidfVectorizer. We configure the vectorizer to ignore rare terms and limit the vocabulary size to focus on the most discriminative features. Furthermore, to address the inherent class imbalance in defect prediction datasets, we explicitly configure the LightGBM classifier to use weighted loss for the positive class, ensuring the model is not biased toward the majority class of clean files. The line-level ranking stage remains unchanged and continues to use the original heuristics (NT and NFC) as defined by (Guo et al., 2023).

2.3.1. GLANCE+Line: Semantically-Aware Line-Level Metric

To move beyond purely syntactic heuristics, GLANCE+Line introduces a new scoring function based on three semantically-aware metrics. These metrics are designed to better reflect the underlying causes of software defects, which are often rooted in human cognitive limitations and the misuse of complex APIs. By incorporating semantic context, we aim to create a more accurate and intuitive model of code bugginess. The new DefectPronenessScore combines these metrics into a weighted sum, enabling a nuanced assessment of each line's risk. This enhancement represents a shift from surface-level syntax to deeper, developer-centric reasoning about code quality.

The Cognitive Complexity Score (CCS) directly measures the cognitive effort required for a human to understand a line of code. This metric is grounded in Cognitive Load Theory (Segalotto et al., 2023), which posits that human working memory is limited. Code structures that break linear flow (e.g., `continue`, `break`) or increase nesting depth (e.g., nested `if` blocks) exponentially increase the mental effort required for comprehension, thereby increasing the likelihood of induced defects (Pereira, Souza, & Pinto, 2021). Based on the formulation by SonarSource, Cognitive Complexity increments for each break in the linear flow of code (e.g., `if`, `for`, `while`, `catch`, `switch`, ternary operators, and logical operators like `&&` and `||`) and adds a further penalty for each level of nesting (Lavazza, Abualkishik, Liu, & Morasca, 2023).

The API-Weighted Number of Function Calls (AW-NFC) refines the original NFC by acknowledging that not all API calls carry the same risk. Empirical studies have shown that misuses of certain types of APIs particularly those related to I/O, concurrency, reflection, and networking, are a common source of defects (Li et al., 2021). We define a set of high-risk APIs, specifically `java.io`, `java.nio`, `java.net`, `java.sql`, `java.rmi`, `java.lang.reflect`, `java.lang.Thread`, `java.util.concurrent`, and `javax.xml`, which involve non-determinism or external state mutation. Calls

to these APIs are weighted higher than standard method invocations. This allows the model to differentiate between a benign call to `Math.max()` and a potentially hazardous call to `java.io.FileInputStream`.

The Variable-Write Count (VWC) quantifies the number of state-changing operations on a given line of code. Based on the principle of Data Flow Analysis (Shaoming Qiu et al., 2025), it counts all instances of assignment (`=`), compound assignment (`+=`, `-=`, etc.), and increment/decrement operators (`++`, `--`). The rationale is that lines of code that frequently modify the state of variables introduce higher dynamic complexity and increase the potential for unintended side effects, a common source of logical errors (Bagaev, Khabibrakhmanova, Sabaev, & Bugayenko, 2024). VWC captures this aspect of code behavior, which is overlooked by purely structural metrics. It reflects the operational intensity of a line, providing a signal for potential instability due to frequent state mutations.

$$DefectProneness Score = (w_1 \cdot CCS) + (w_2 \cdot AW{-}NFC) + (w_3 \cdot VWC) \quad (2)$$

These three metrics are combined into a single DefectPronenessScore using a weighted linear function as shown in the second equation. The weights (w_1, w_2, w_3) are calibrated to assign higher importance to API usage, reflecting the empirically observed high defect density in high-risk API calls (Li et al., 2021). For the GLANCE+Line model, the file-level prediction stage retains the original Logistic Regression classifier, while the line-level ranking is performed using the new semantic scoring function.

2.3.2. GLANCE++: The Integrated Model

GLANCE++ represents the full integration of the two proposed enhancements into a unified, end-to-end line-level defect prediction framework. It synergistically combines the enhanced file-level classifier (GLANCE+File) and the semantically-aware line-level ranking model (GLANCE+Line) to form a cohesive system that improves performance at both stages of the prediction pipeline. As illustrated in figure 2, the file-level prediction is performed using the LightGBM classifier trained on TF-IDF features, while the line-level ranking is conducted using the weighted DefectPronenessScore derived from the Cognitive Complexity Score (CCS), API-Weighted Number of Function Calls (AW-NFC), and Variable-Write Count (VWC).

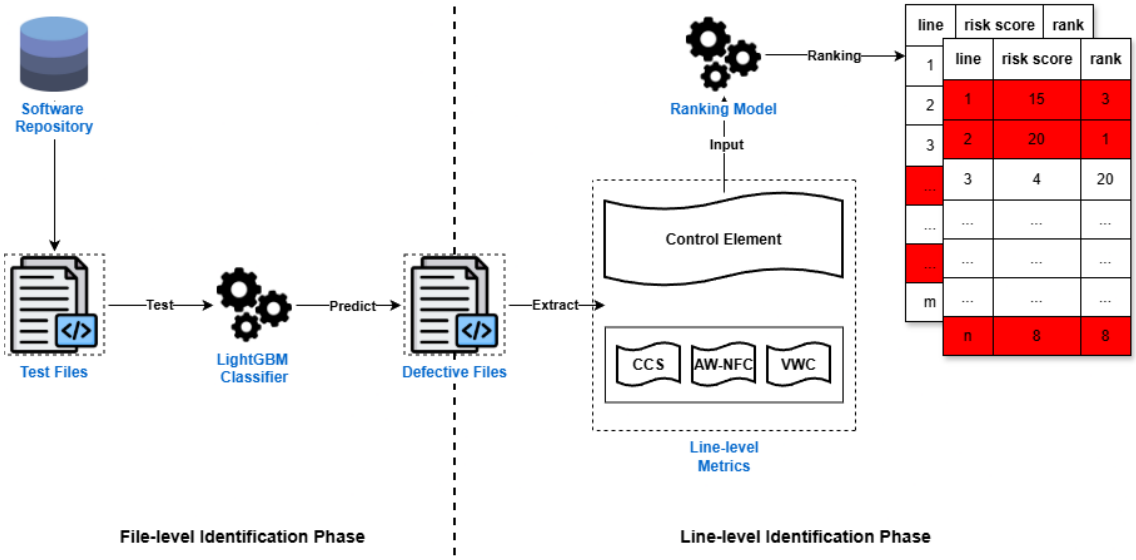

Figure 2. The two-phase architecture of GLANCE++ framework

The integration is designed to maximize predictive efficacy through a two-stage filtering process. First, the LightGBM classifier identifies defective files by capturing non-linear interactions among lexical features, thereby reducing the number of false positives and narrowing the search space more effectively than the baseline. Second, within the predicted defective files, the semantically-aware metrics prioritize lines based on their cognitive load, API risk, and state-modification intensity, enabling a more intelligent and context-sensitive ranking. The hypothesis is that the combined effect of these improvements will yield a statistically significant performance gain over both the original GLANCE-LR and its individual enhanced variants (GLANCE+File and GLANCE+Line).

2.4. Dataset

To ensure a fair and reproducible comparison with the state of the art, we adopt the BugDet dataset: a large-scale, publicly available benchmark curated by (Guo et al., 2023) for code-line-level bugginess identification (CLBI). BugDet comprises 19 open-source Java projects from the Apache Software

Foundation, spanning 142 distinct releases, and is notable for its methodological rigor in constructing ground-truth labels. Unlike prior datasets by (Wattanakriengkrai et al., 2022) that rely solely on main-branch commits, BugDet mines bug-fixing activities across multiple development branches, significantly improving label completeness and reducing false negatives. Furthermore, the dataset explicitly addresses common sources of noise, such as the mislabeling of test files as defective, and provides detailed metadata to support sensitivity analysis.

The construction of the dataset follows a rigorous and systematic pipeline designed to ensure high-quality, production-relevant labels (Guo et al., 2023). First, bug-fixing commits are identified by mining commit messages that explicitly reference resolved JIRA issues, with additional filtering to exclude commits related solely to refactoring or non-defect-related changes. Next, buggy source lines are derived using an enhanced version of the SZZ algorithm, which traces code changes across all Git branches, not only the main branch, to capture a more complete and accurate set of defect-inducing lines. To align with real-world development practices, test files are explicitly excluded from the analysis, as their purpose and failure modes differ fundamentally from production code. Furthermore, only non-empty, executable source lines are retained: blank lines, standalone comments, and files lacking any executable statements are removed during preprocessing to minimize noise and ensure that the evaluation reflects actual developer-facing logic.

Projects were selected systematically based on GitHub star count (>1,000) and resolved JIRA bug report volume (>100), ensuring diversity in size, domain, and defect density. From left to right, Table 1 columns detail each project's name and functional description, the number of releases analyzed, the range of bug-fixing commits per release, the total source code files, the codebase size measured in thousands of lines of code (KLOC), and the percentages of defective files and lines respectively.

Table 1. Statistical Information of the Studied Projects

Project	Description	#Rel	#Bugs	#Files	#KLOC	%Defect Files	%Defect Lines
Ambari	Tool for provisioning, managing, and monitoring Apache Hadoop clusters	7	62~489	2,307	366.3	13.57%	1.47%
ActiveMQ	Open source, multi-protocol, Java-based message broker	14	97~845	1,835	292.8	12.32%	1.13%
BookKeeper	Scalable, fault-tolerant, and low-latency storage service	3	31~157	240	57.7	36.67%	3.81%
Calcite	Dynamic data management framework	8	301~554	1,430	304.5	43.50%	4.20%
Cassandra	Open-source NoSQL distributed database	6	138~407	583	155.7	26.76%	1.22%
Flink	Unified stream-processing and batch-processing framework	2	55~78	3,583	564.1	2.93%	0.20%
Groovy	Flexible and extensible Java-like language for the JVM	14	36~641	914	194.1	7.99%	0.62%
HBase	Distributed, scalable, big data store	5	239~493	1,076	487	24.44%	0.94%
Hive	Distributed, fault-tolerant data warehouse system	4	56~321	3,263	953.1	6.56%	0.29%
Ignite	Distributed database and in-memory computing platform	3	824~859	2,658	612	31.08%	1.68%
Log4j2	Java-based logging framework	11	215~255	798	109	22.56%	1.56%
Mahout	Library of scalable machine learning algorithms	6	60~117	933	134.7	17.58%	1.63%
Maven	Build automation and project management tool	6	32~124	663	94.6	10.41%	0.63%
NiFi	Data flow systems automation tool	4	323~545	2,809	432.1	18.26%	1.37%
Nutch	Highly extensible and scalable web crawler	13	54~158	332	56.7	19.28%	1.06%
Storm	Distributed real-time computation system	5	5~183	997	166.4	4.41%	0.24%
Tika	Content analysis toolkit	12	105~179	393	66.2	23.41%	2.11%
Struts	Java web applications framework	15	94~227	1,087	150.8	11.04%	0.86%
ZooKeeper	Centralized service for maintaining configuration information	4	40~80	339	63.7	21.24%	1.26%
Average	-	7	217	1,507	262.5	12.67%	1.03%

These statistics reveal the dataset's substantial heterogeneity, with project sizes ranging from 56.7 KLOC for Nutch to 953.1 KLOC for Hive, and defect densities varying from 0.20% in Flink to 4.20% in Calcite. On average, each release contains 262.5 KLOC and 1.03% proportion of defective lines. This broad diversity is instrumental in mitigating threats related to inter-project variability, ensuring that our evaluation covers a wide spectrum of software scales and complexities.

2.5. Evaluation Protocol

To ensure the validity, reproducibility, and comparability of our findings, we adopt the same evaluation protocol as the original GLANCE study (Guo et al., 2023). This strict adherence enables a direct and fair comparison between the baseline model and our proposed enhancements, ensuring that any observed performance differences can be confidently attributed to the introduced modifications rather than variations in experimental setup.

To systematically evaluate the contribution of each enhancement and to directly address our research questions (RQ1–RQ3), we conduct a comprehensive ablation study (Shaojian Qiu et al., 2024). This involves training and evaluating four distinct model configurations under identical experimental conditions. The ablation design allows us to isolate the impact of the file-level classifier, the line-level metrics, and their combination:

(1) GLANCE-LR: The original baseline, using Logistic Regression and syntactic heuristics (NT and NFC).
(2) GLANCE+File: The baseline with an enhanced file-level classifier (LightGBM), addressing RQ1.
(3) GLANCE+Line: The baseline with semantically-aware line-level metrics (CCS, AW-NFC, VWC), addressing RQ2.
(4) GLANCE++: The fully integrated model combining both enhancements, addressing RQ3.

We employ a cross-release prediction setting to emulate a realistic Software Quality Assurance (SQA) workflow. For each project with a sequence of releases, the model is trained on data from an earlier release (N) and evaluated on the immediately subsequent release (N+1). This temporal hold-out strategy reflects the practical scenario where historical defect data is used to predict bugs in newly developed or modified code (S. Wang et al., 2023; Wattanakriengkrai et al., 2022; Zhou et al., 2018). The cross-release setting is critical for ensuring external validity and preventing temporal data leakage, a common threat in defect prediction research where future information might inadvertently influence model training.

2.6. Performance Metrics

To conduct a comprehensive and rigorous evaluation of our proposed GLANCE++ framework, we evaluate performance across four distinct dimensions: classification accuracy, effort-aware ranking effectiveness, bug-level coverage, and operational efficiency. We adopt the established suite of eight performance indicators from the original GLANCE study (Guo et al., 2023), summarized in Table 2, to ensure direct baseline comparability.

Table 2. Performance Metrics for Line-Level Defect Prediction Evaluation

Scenario	Metric	Formula	Better	Range
Classification	Recall	$\frac{\lvert TP\rvert}{\lvert TP\rvert + \lvert FN\rvert}$	High	[0, 1]
	FAR	$\frac{\lvert FP\rvert}{\lvert FP\rvert + \lvert TN\rvert}$	Low	[0, 1]
	CoF	$\frac{\lvert FN\rvert}{\lvert FN\rvert + \lvert TN\rvert}$	Low	[0, 1]
	D2H	$\sqrt{\frac{(1-Recall)^2 + (0-FAR)^2}{2}}$	Low	[0, 1]
	MCC	$\frac{\lvert TP\rvert \times \lvert TN\rvert - \lvert FP\rvert \times \lvert FN\rvert}{\sqrt{(\lvert TP\rvert + \lvert FP\rvert)(\lvert TP\rvert + \lvert FN\rvert)(\lvert TN\rvert + \lvert FP\rvert)(\lvert TN\rvert + \lvert FN\rvert)}}$	High	[-1, 1]
Ranking	R@20%	$\frac{\#Defectlinesattop20\%ofrank}{\#Alldefectlines}$	High	[0, 1]
	IFA	k	Low	[0, N]
Bug-Level	HOB	$\frac{\#Hitdefects}{\#Alldefects}$	High	[0, 1]

We also introduce a new efficiency metric to rigorously assess real-time feasibility. This expanded suite of metrics ensures that our evaluation captures not only the predictive power of the model but also its practical viability in resource-constrained environments.

Classification Metrics, these metrics evaluate the model's overall ability to correctly classify individual lines of code as defective or clean, without considering the order of predictions.

(1) Recall (Wattanakriengkrai et al., 2022) measures the proportion of all actual buggy lines in a project release that the model successfully identifies. A high Recall indicates that the model is effective at finding existing defects and minimizes the number of missed bugs (False Negatives).
(2) FAR (False Alarm Rate) (Wattanakriengkrai et al., 2022) measures the proportion of all clean lines that were incorrectly flagged as buggy. A low FAR is essential for preserving developer trust and reducing wasted inspection effort.
(3) CoF (Cost of Failures) (Zhang & Cheung, 2013) measures the proportion of buggy lines that were missed (FN) relative to all lines the model predicted as clean (FN + TN). A low CoF indicates that the model's "clean" predictions are highly reliable.
(4) D2H (Distance-to-Heaven) (Agrawal, Fu, Chen, Shen, & Menzies, 2021; Wattanakriengkrai et al., 2022) is a composite metric combining Recall and FAR into a single score representing the Euclidean distance from a perfect classifier. A D2H value of 0 represents a perfect model (Recall=1 and FAR=0), while a value closer to 1 represents a worst model (Recall=0 and FAR=1).
(5) MCC (Matthews Correlation Coefficient) (Pornprasit & Tantithamthavorn, 2023; Shaojian Qiu et al., 2024; Wattanakriengkrai et al., 2022) measures the correlation coefficient between the observed and predicted classifications, ranging from -1 (total disagreement) to +1 (perfect agreement), with 0 indicating a random guess. MCC is particularly suitable for imbalanced datasets, such as line-level defect prediction task.

Ranking & Effort-Aware Metrics, these metrics evaluate how well a model prioritizes its predictions, which is critical in a resource-constrained SQA environment. They are calculated based on a single ranked list of all lines in a project release, sorted from most to least suspicious.

(1) Recall@20% (Pornprasit & Tantithamthavorn, 2023; Shaojian Qiu et al., 2024; Yang et al., 2024) measures the percentage of all buggy lines that are found within the top 20% of the most suspicious lines of code (LOC) as ranked by the model. A high Recall@20% indicates that the model is effective at ranking true defects at the top of the list.
(2) IFA (Initial False Alarms) (Pornprasit & Tantithamthavorn, 2023; Shaojian Qiu et al., 2024; Wattanakriengkrai et al., 2022; Yang et al., 2024) measures the number of clean lines a developer must inspect in the ranked list before encountering the very first true buggy line. An IFA of 0 means the top-ranked line is a true defect.

Bug-Level Metric, this metric evaluates performance from the perspective of bug reports rather than individual lines. HoB (Hit of Bugs) (Guo et al., 2023; Rahman, Khatri, Barr, & Devanbu, 2014) measures the percentage of unique bug reports (from JIRA) for which at least one associated buggy line was correctly identified by the model. A high HoB indicates that the model is not just repeatedly finding lines related to a few easy bugs, but is successfully identifying defects across a wide range of different bug reports.

Normalized Efficiency Metric, to assess practical feasibility, we introduce a normalized efficiency metric, Inference Speed (s/KLOC). While prior studies report raw inference time (Fu & Tantithamthavorn, 2022; Pornprasit & Tantithamthavorn, 2023), such measures are biased by project size (i.e., larger files naturally take longer to process). To enable a fair comparison across datasets of varying magnitude, we normalize the total inference latency (feature extraction + prediction) by the project size in thousands of lines of code (KLOC), as outlined in the third equation. This metric represents the processing time required per 1,000 lines of code, providing a project-agnostic measure of real-time deployment capability.

$$Speed = \frac{T_{extraction} + T_{prediction}}{KLOC_{total}} \quad (3)$$

We prioritize IFA (Initial False Alarms) and Recall@20%LOC as our primary indicators of practical utility, as these metrics directly reflect the effort-aware inspection behavior of developers under realistic resource constraints, often referred to as the "developer patience threshold". Critically, both metrics are also robust to extreme class imbalance, which is inherent in line-level defect prediction (where <3% of lines are buggy on average). Unlike threshold-dependent classification metrics (e.g., F1-score), IFA and Recall@20%LOC evaluate ranking quality and early bug detection without requiring balanced positive/negative samples, making them both practically meaningful and statistically reliable in highly imbalanced settings (Pornprasit & Tantithamthavorn, 2023; Shaojian Qiu et al., 2024; Wattanakriengkrai et al., 2022; Yang et al., 2024).

2.7. Statistical Analysis

To ensure that our conclusions are statistically robust and not driven by random variation, we employ non-parametric statistical tests, which are appropriate for performance data that often violates the assumption of normality (Guo et al., 2023; Wattanakriengkrai et al., 2022). For each of our proposed models, namely GLANCE+File, GLANCE+Line, and GLANCE++, we compare its performance against the GLANCE-LR baseline across the 123 cross-release prediction pairs for each of the performance metrics.

Specifically, we use the Wilcoxon signed-rank test ($p < 0.05$), a non-parametric test suitable for paired comparisons, to assess whether the median performance difference between a proposed model and the baseline is statistically significant (Ahmad, Goseva-Popstojanova, & Lutz, 2024; Parashar, Kumar Goyal, Kaushal, & Kumar Sahana, 2022). To control the False Discovery Rate (FDR) arising from multiple comparisons (i.e., multiple models and metrics), we apply the Benjamini-Hochberg (BH) correction to the raw p-values (H. Wang, Zhuang, & Zhang, 2021). A result is considered statistically significant if the BH-corrected p-value is less than the significance level of 0.05 (Guo et al., 2023).

To quantify the practical significance of any observed differences, we compute Cliff's delta (δ), a non-parametric effect size measure that indicates the degree of separation between two distributions (Chen et al., 2024; Zhong, Song, Lv, & He, 2021). Unlike p-values, which can be influenced by sample size, Cliff's delta provides a standardized, interpretable measure of the magnitude of the difference. We interpret the effect size using the following thresholds: Negligible ($|\delta| < 0.147$), Small ($0.147 \leq |\delta| < 0.33$), Medium ($0.33 \leq |\delta| < 0.474$), and Large ($|\delta| \geq 0.474$) (Guo et al., 2023; Pornprasit & Tantithamthavorn, 2023). A large effect size indicates a meaningful and practically relevant improvement, even if the p-value is not significant, and vice versa.

2.8. Implementation Details

The implementation of our GLANCE++ framework is built on Python 3.11, utilizing Scikit-learn for feature extraction and the LightGBM framework (Version 4.6.0) for the enhanced file-level classifier. All experiments are conducted on a commodity laptop equipped with an Intel Core i7-12700H Processor (14 Cores) @ 2.3 GHz and 16 GB of RAM, running on Windows 11. Unlike deep learning approaches that require heavy GPU acceleration (e.g., NVIDIA RTX 3090), our framework operates efficiently on a standard CPU. The average model training time is approximately 2.65 seconds per release. Below, we describe the hyperparameter settings used for our approach.

File-Level Classifier settings. For the lexical feature extraction, we use the TfidfVectorizer with a vocabulary size of 5,000 (`max_features=5000`) and a minimum document frequency of 3 (`min_df=3`). This configuration ensures that we capture the most discriminative code tokens while filtering out rare identifiers that often lead to overfitting (Stradowski & Madeyski, 2023; Zhao, Damevski, & Chen, 2023). For the LightGBM classifier, we use the following settings:

(1) Learning Rate: Set to 0.05 with 1,000 estimators (`n_estimators=1000`) to ensure gradual and robust convergence.
(2) Tree Structure: We utilize the leaf-wise growth strategy with `num_leaves=31`, `max_depth=-1` (unlimited), and `min_child_samples=20`. This allows the model to capture complex non-linear feature interactions more effectively than level-wise growth algorithms (e.g., Random Forest) (Ke et al., 2017).
(3) Optimization: We use the binary log loss objective (`objective="binary"`) and the Area Under the Curve (`metric="auc"`) for evaluation during training.
(4) Class Imbalance: We explicitly set `is_unbalance=True`. Instead of using synthetic oversampling techniques like SMOTE (which can introduce noise and increase training time), this parameter automatically adjusts weights for the positive class (defective files) within the loss function, ensuring the model is not biased toward the majority class (clean files) (Roy, Pramanik, & Sarkar, 2024).

Line-Level Metric settings. For the line-level ranking, we apply a weighted scoring function to combine our semantic metrics. We set the weights as follows: $w_{CCS} = 1.0$, $w_{AW-NFC} = 1.5$, and $w_{VWC} = 1.0$. The higher weight for AW-NFC (1.5) was chosen based on empirical observations that high-risk API calls (e.g., `java.io`, `java.sql`) are stronger indicators of defect proneness than purely structural complexity alone (Li et al., 2021).

3. RESULTS AND DISCUSSION

This section presents the evaluation of our proposed enhancements to the GLANCE framework for code-line-level defect prediction. We analyze the performance of three model variants: GLANCE+File, GLANCE+Line, and GLANCE++, in comparison to the original baseline model GLANCE-LR proposed by (Guo et al., 2023). These experiments were conducted on a large-scale dataset composed of 19 open-

source Java projects spanning 142 releases, following the same cross-release prediction protocol and performance metrics as the original study.

3.1. Performance Comparison Across Metrics

Figure 3 presents the distribution of performance across eight evaluation metrics (Recall, FAR, CoF, D2H, MCC, IFA, Recall@20%LOC, and HoB) for four model variants: the baseline GLANCE-LR (marker blue), GLANCE+File (marker red), GLANCE+Line (marker light blue), and GLANCE++ (marker orange). Each boxplot represents the performance across 123 cross-release prediction pairs, enabling a comprehensive comparison of central tendency, variability, and extreme values.

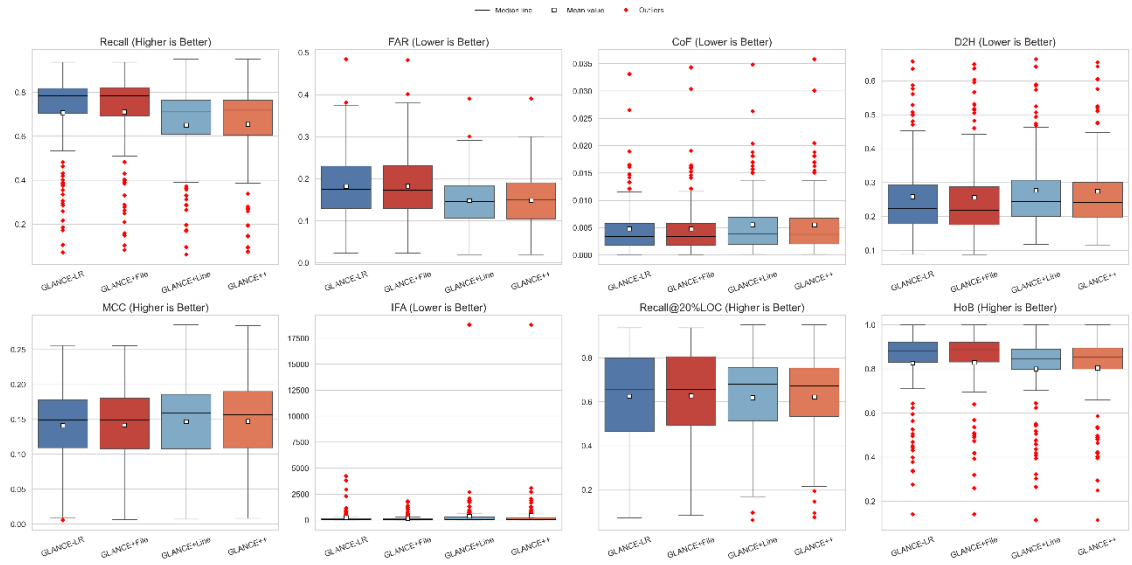

Figure 3. Performance Distribution Comparison between GLANCE-LR, GLANCE+File, GLANCE+Line, and GLANCE++

A visual inspection of the boxplots highlights several notable trends across the models. GLANCE+File maintains a median Recall comparable to GLANCE-LR while reducing median IFA, suggesting improved efficiency without sacrificing defect detection. By contrast, GLANCE+Line and GLANCE++ emphasize precision-oriented metrics: they achieve higher FAR and lower MCC, but this comes at the expense of reduced Recall. The distribution of results further illustrates differences in robustness. GLANCE-LR and GLANCE+File show relatively tight interquartile ranges (IQR) for Recall, CoF, and MCC, indicating stable performance across projects. In comparison, GLANCE+Line and GLANCE++ display wider variability, especially in effort-aware metrics such as IFA, where extreme outliers indicate cases of disproportionately high inspection effort.

Table 3 summarizes the mean, median, and standard deviation of each metric across the four models. This quantitative analysis confirms the visual observations and provides a clearer picture of performance centrality and variability:

(1) *GLANCE-LR (Baseline)*. Demonstrates high sensitivity with the highest Median Recall (0.783) and HoB (88%). However, it lacks precision, evidenced by a high False Alarm Rate (FAR) of 0.175 and a substantial Median Initial False Alarm (IFA) count of 36, indicating that developers often encounter numerous false positives before finding the real bug (Guo et al., 2023).

(2) *GLANCE+File*. The substitution of Logistic Regression with LightGBM primarily impacts early filtering. While its general classification metrics (Recall, FAR, MCC) remain nearly identical to the baseline, it achieves a 42% reduction in Median IFA (from 36 to 21). This suggests that the non-linear classifier effectively filters out obvious clean files, even though it does not fundamentally alter the ranking of lines within the remaining defective files.

(3) *GLANCE+Line*. This semantically-enhanced variant drives a major shift toward precision, achieving the lowest Median FAR (0.146) and highest Median MCC (0.159). This confirms that metrics like CCS and AW-NFC successfully filter out benign code structures that syntactic heuristics would otherwise flag. However, this selectivity comes at a cost: Median Recall drops to 0.711, indicating the model becomes too conservative and misses simple defects lacking semantic complexity.

(4) *GLANCE++*. The fully integrated model largely mirrors the performance profile of GLANCE+Line, with a comparable Median Recall (0.718) and FAR (0.149). Crucially, both semantic models exhibit extreme variance in inspection effort; while their median IFA is moderate (~54-57), the standard deviation spikes to over 1,700. This volatility suggests that in specific "outlier" cases (e.g., robustly

tested "god classes"), the semantic models heavily penalize complex-but-correct code, forcing developers to inspect thousands of lines in those specific files.

Table 3. The Mean Value, Median Value, and Standard Deviation Value of GLANCE Models (The Best Value is Marked in Bold)

	Model	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
Mean	GLANCE-LR	0.707	0.182	0.005	0.259	0.141	235	0.625	83%
	GLANCE+File	0.711	0.183	0.005	0.257	0.142	147	0.627	83%
	GLANCE+Line	0.650	0.148	0.006	0.277	0.147	381	0.619	80%
	GLANCE++	0.655	0.148	0.006	0.275	0.147	431	0.622	80%
Median	GLANCE-LR	0.783	0.175	0.003	0.223	0.149	36	0.655	88%
	GLANCE+File	0.783	0.173	0.003	0.218	0.150	21	0.655	89%
	GLANCE+Line	0.711	0.146	0.004	0.245	0.159	54	0.679	85%
	GLANCE++	0.718	0.149	0.004	0.242	0.157	57	0.671	85%
Std Dev	GLANCE-LR	0.191	0.083	0.005	0.119	0.052	626	0.198	16%
	GLANCE+File	0.191	0.085	0.005	0.120	0.052	322	0.199	17%
	GLANCE+Line	0.177	0.065	0.005	0.114	0.055	1727	0.186	16%
	GLANCE++	0.177	0.067	0.006	0.115	0.055	1751	0.186	16%

In summary, the performance data highlights a fundamental divergence in model behavior. While the file-level enhancement (GLANCE+File) offers a safe optimization, improving efficiency without altering classification dynamics, the line-level semantic enhancements (GLANCE+Line and GLANCE++) fundamentally reshape the prediction landscape. They effectively trade sensitivity for significantly higher precision and reduced median inspection effort. However, the emergence of extreme volatility in IFA warrants caution, suggesting that semantic metrics may behave unpredictably in complex edge cases.

3.2. Statistical Significance and Effect Size Analysis

To rigorously assess the performance differences between our enhanced models and the baseline, we applied non-parametric Wilcoxon signed-rank test ($p < 0.05$) with Benjamini-Hochberg (BH) correction to control for multiple comparisons, and quantified the magnitude of difference using Cliff's delta (δ). Table 4 presents the statistical analysis of eight evaluation metrics, comparing GLANCE+File, GLANCE+Line, and GLANCE++ against the GLANCE-LR baseline across 123 cross-release prediction pairs.

Table 4. The Results from Wilcoxon-signed Test and Cliff's Delta for Comparison between Baseline GLANCE-LR and Enhanced Variants (Abbr. N: Negligible; S: Small; M: Medium; L: Large)

	Model	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
p-value	GLANCE+File	6.1e-01	8.9e-01	6.9e-01	8.8e-01	4.7e-01	2.2e-01	3.6e-01	6.5e-01
	GLANCE+Line	2.7e-18	6.4e-22	9.2e-15	1.1e-07	3.1e-07	6.5e-02	2.1e-01	1e-16
	GLANCE++	6.8e-14	4.7e-21	4.6e-12	5.6e-05	1.6e-05	6.3e-02	5e-01	8e-09
BH-corrected p-value	GLANCE+File	0.614	0.889	0.692	0.881	0.468	0.219	0.365	0.647
	GLANCE+Line	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	0.065	0.206	< 0.001
	GLANCE++	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	0.063	0.5	< 0.001
Cliff's Delta	GLANCE+File	0.027 (N)	-0.003 (N)	-0.005 (N)	-0.011 (N)	0.005 (N)	-0.095 (N)	0.011 (N)	0.023 (N)
	GLANCE+Line	-0.353 (M)	-0.250 (S)	0.103 (N)	0.142 (N)	0.060 (N)	0.143 (N)	-0.054 (N)	-0.227 (S)
	GLANCE++	-0.347 (M)	-0.250 (S)	0.098 (N)	0.130 (N)	0.064 (N)	0.146 (N)	-0.041 (N)	-0.194 (S)

GLANCE+File vs. GLANCE-LR. The statistical analysis reveals that the model does not achieve a statistically significant improvement over the baseline on any metric (BH-corrected $p > 0.05$). Despite the 42% reduction in median IFA, the high variability across projects renders this improvement statistically negligible ($\delta = -0.095$). This suggests that for file-level prediction in this dataset, the linear separability assumption of Logistic Regression holds sufficiently well, and the non-linear capacity of LightGBM offers diminishing returns.

GLANCE+Line vs. GLANCE-LR. In contrast, the model demonstrates statistically significant differences across six out of eight metrics (BH-corrected $p < 0.001$). The effect sizes reveal a clear trade-off between sensitivity and precision:

(1) Recall shows a medium negative effect ($\delta = -0.353$), indicating a meaningful reduction in the proportion of bugs detected. This suggests that the semantically-aware metrics may be overly conservative in some contexts.
(2) FAR shows a small negative effect ($\delta = -0.250$), reflecting a significant gain in precision by reducing false alarms.
(3) HoB shows a small negative effect ($\delta = -0.227$), indicating a moderate decline in the ability to hit diverse bug reports.

The direction of the Cliff's delta values confirms that GLANCE+Line semantic metrics successfully enforce a stricter decision boundary, significantly reducing noise (False Alarms) but systematically filtering out "low-complexity" bugs.

GLANCE++ vs. GLANCE-LR. The model exhibits a statistical profile nearly identical to GLANCE+Line, with significant improvements in the same six metrics (BH-corrected $p < 0.001$) and similar effect sizes. This indicates that the addition of the file-level classifier does not mitigate the recall loss nor significantly boost the precision gains of the line-level metrics. The performance is dominated by the line-level ranking characteristics.

3.3. Runtime Efficiency and Scalability Analysis

To validate the "lightweight" claim of GLANCE++, we conducted a rigorous runtime analysis across all 19 projects (123 releases) using a standard commodity CPU (Intel Core i7-12700H). We instrumented the framework to isolate inference time, which is defined as the sum of feature extraction (syntactic and semantic) and model prediction, from training time. To ensure fair comparison across projects of vastly different sizes, we normalized execution time by calculating the Speed (seconds per 1,000 Lines of Code, or KLOC). Table 5 aggregates the efficiency results categorized by project scale. The results reveal three critical findings regarding the operational characteristics of GLANCE++.

(1) *High-Speed Inference*. The global average inference speed is 0.0044 seconds per KLOC. This translates to a throughput of approximately 230,000 lines of code per second. For a massive industrial project of 1 million lines, the estimated analysis time is under 4.5 seconds. This confirms that the addition of semantic metrics (CCS, AW-NFC), while computationally heavier than simple token counting, does not compromise the real-time nature of the framework.
(2) *Positive Scalability*. Contrary to many static analysis tools that suffer from non-linear performance degradation, GLANCE++ exhibits positive scalability. As shown in Table 5, the normalized speed actually improves as project size increases (from 0.0050 s/KLOC for small projects to 0.0034 s/KLOC for large projects). This indicates that the fixed overheads of the framework (e.g., model loading, vectorizer initialization) are effectively amortized over larger codebases.
(3) *Rapid Retraining*. The average training time is just 2.65 seconds, with even the largest projects taking only ~6 seconds. This ultra-fast training capability allows the model to be retrained frequently, potentially on every commit, without becoming a bottleneck in Continuous Integration (CI) pipelines.

Table 5. Runtime Efficiency Analysis of GLANCE++ Across Project Scales

Project Scale	Avg. Size (KLOC)	Avg. Training Time (s)	Avg. Inference Time (s)	Avg. Speed (s/KLOC)
Small (< 100 KLOC)	68.3	0.84	0.33	0.0050
Medium (100–500 KLOC)	240.0	3.18	1.03	0.0041
Large (> 500 KLOC)	737.9	6.03	2.39	0.0034
Global Average	224.6	2.65	0.91	0.0044

Collectively, these metrics provide definitive empirical evidence supporting the "lightweight" designation of GLANCE++. Unlike deep learning models that often require specialized hardware and minutes of processing time, our framework delivers high-fidelity semantic analysis on standard CPUs with negligible latency. This operational efficiency removes the traditional performance barrier to adopting advanced defect prediction, demonstrating that semantically-aware tools can be seamlessly integrated into high-velocity development environments without disrupting developer workflow.

3.4. Practical Trade-Offs and Implications

This study reveals several fundamental trade-offs inherent in enhancing the GLANCE framework. These trade-offs highlight the delicate balance between precision, sensitivity, and operational efficiency, with significant implications for practical deployment.

The Precision-Recall Trade-off. The most distinct trade-off observed is between false alarm reduction and defect coverage. GLANCE+Line and GLANCE++ achieve statistically significant reductions in False Alarm Rate (FAR) and improvements in MCC. This indicates a stronger ability to distinguish true defects from noise. However, these gains come at the cost of significantly reduced Recall (approx. -0.35 effect size). This suggests that while semantically-aware metrics improve prediction confidence by targeting high-risk code (e.g., complex APIs), they may be overly conservative, missing "simpler" defects that lack semantic complexity.

Operational Efficiency vs. Semantic Depth. A common concern with semantic analysis is the computational cost. However, our runtime analysis demonstrates that this trade-off is minimal. While GLANCE++ requires AST traversal, making it theoretically more expensive than the purely lexical GLANCE-LR, the actual inference time remains in the range of milliseconds per file. The "cost" of semantic depth is therefore not time, but rather the complexity of implementation (maintaining AST parsers). Given the significant gain in precision, this maintenance cost is likely justifiable for industrial applications where developer time is the scarcest resource.

To summarize, model selection in defect prediction is not a one-size-fits-all decision, but a strategic choice that must align with organizational goals, risk tolerance, and development workflows. For Gatekeeping (CI/CD), GLANCE+Line is the optimal choice. Its high precision and low False Alarm Rate minimize developer disruption, and its sub-second inference time (see Table 5) ensures it does not slow down the build pipeline. For deep audits, the baseline GLANCE-LR may still be preferable. In scenarios where the goal is to find every potential bug regardless of false positives (e.g., prior to a major release), the higher Recall of the baseline model is more valuable than the precision of the enhanced variants.

3.5. Main Findings and Research Question Implications

This section synthesizes the empirical results to directly address our three research questions, linking empirical patterns to software theory.

RQ1: Can a more powerful file-level classifier improve GLANCE? Answer: Marginally, but not significantly. While LightGBM reduces the median inspection effort (IFA) by better modeling non-linear feature interactions, the improvement is not statistically significant. This aligns with the Pareto Principle in defect prediction (Szymański & Ochodek, 2023): the majority of defect signals at the file level are likely captured by simple linear patterns (e.g., presence of specific keywords), leaving little room for complex classifiers to add value without overfitting.

RQ2: Can more semantically-aware line metrics improve line ranking? Answer: Yes, by significantly improving precision and trust. GLANCE+Line significantly outperforms the baseline in FAR and MCC. This validates the Cognitive Load Theory in the context of LLDP (Fakhoury, Roy, Ma, Arnaoudova, & Adesope, 2020): code that is difficult for humans to process (high CCS) or involves external state mutation (high AW-NFC) is indeed more defect-prone. However, this comes with an inherent limitation: it creates a "blind spot" for low-complexity defects, necessitating a strategic choice by the user between high coverage (Baseline) and high trust/low noise (GLANCE+Line).

RQ3: What is the combined effect of both enhancements? Answer: The performance is dominated by line-level metrics. GLANCE++ offers no statistical advantage over GLANCE+Line. This confirms that the bottleneck in Line-Level Defect Prediction is local ranking, not global file filtering. Future optimization efforts should focus on refining line-level heuristics (e.g., dampening the penalty for well-tested complex code) rather than further increasing model complexity at the file level.

3.6. Threats to Validity

Following established guidelines for empirical software engineering research (Wohlin et al., 2024), we identify and discuss the primary threats to validity in our study, categorized as construct, internal, and external validity.

Construct Validity. A key threat to construct validity lies in the operationalization of defect-proneness. Our proposed line-level metrics (CCS, AW-NFC, and VWC) are designed as proxies for code bugginess based on established software engineering principles (e.g., cognitive load, API risk, and state complexity). However, these metrics may not fully capture all dimensions of code quality or defect-inducing patterns. For instance, certain subtle logical errors or concurrency bugs may not be reflected in

our scoring function. Furthermore, the ground-truth labels in the BugDet dataset are derived using the SZZ algorithm, which, while widely used, is known to be susceptible to noise from tangled commits (i.e., single commits that fix multiple bugs or are incorrectly linked to bug reports) (Wattanakriengkrai et al., 2022). This labeling noise may affect the accuracy of our evaluation, particularly for lines that are indirectly related to a bug fix. Despite efforts to use high-quality datasets, this remains a common limitation in defect prediction research.

Internal Validity. Internal validity concerns potential biases in our experimental design and implementation. One potential threat is implementation bias, which refers to errors in our code that could skew the results. To mitigate this, we built our framework directly on the publicly available CLBI implementation by (Guo et al., 2023), reusing their data loading, preprocessing, and evaluation scripts. This ensures a direct and fair comparison between GLANCE-LR and our enhanced variants. We also performed extensive unit testing and cross-verification of key components (e.g., metric computation, model training). Another potential threat is hyperparameter sensitivity; however, we used default or commonly accepted configurations for LightGBM and determined the weights for the DefectPronenessScore through cross-validation on a held-out subset, minimizing arbitrary choices.

External Validity. The generalizability of our findings is limited to the 19 open-source Java projects in the BugDet dataset. While these projects span diverse domains and sizes, they may not represent the characteristics of commercial software systems, which often have different development processes, code review practices, and architectural constraints. Moreover, our model relies on Java-specific syntactic and semantic patterns (e.g., API package structures, control flow constructs), which may not translate directly to other programming languages such as Python, C++, or JavaScript. For example, the risk profile of APIs in Python's requests or pandas libraries may differ significantly from Java's java.io or java.net. Therefore, while our results are robust within the studied context, further evaluation is needed to assess the cross-language and cross-domain applicability of GLANCE++.

4. CONCLUSION AND FUTURE WORK

This study addresses the "Semantic Gap" in heuristic-based Line-Level Defect Prediction (LLDP) by proposing GLANCE++, a framework that shifts the paradigm from simple syntactic counting to theoretically grounded Semantic Risk Assessment. While the original GLANCE-LR established that lightweight models could perform competitively against state-of-the-art deep learning approaches, it lacked the ability to distinguish between benign code and high-risk constructs driven by cognitive complexity or state mutation.

Our empirical evaluation, conducted on 19 open-source Java projects spanning 142 releases, reveals several key findings. First, the introduction of semantically-aware metrics, grounded in Cognitive Load Theory (CCS), API Misuse Patterns (AW-NFC), and Data Flow Analysis principles (VWC), resulted in statistically significant reductions in False Alarm Rates (FAR). This validates our hypothesis that defect proneness is better predicted by "code risk" than "code size". Second, we crucially demonstrated that this semantic depth does not come at the cost of scalability. Our runtime analysis confirms that GLANCE++ maintains an ultra-low inference latency of 0.0044 seconds per KLOC, making it orders of magnitude faster than Transformer-based alternatives and perfectly suited for real-time CI/CD gatekeeping. Third, the marginal performance gain from replacing Logistic Regression with LightGBM (GLANCE+File) suggests that the bottleneck in LLDP is not file-level classification power, but rather line-level ranking intelligence.

From a practical standpoint, our findings suggest that GLANCE+Line is the most impactful enhancement. It is particularly well-suited for precision-critical environments such as industrial static analysis tools or safety-critical systems, where minimizing false positives and developer distraction is paramount. In contrast, GLANCE+File may be preferable in QA settings where maximizing bug coverage (high recall) is the primary goal, despite a higher false alarm rate.

We identify three high-priority directions to further advance this "Semantic Heuristic" paradigm:

(1) Deep Semantic Heuristics: Beyond CCS and AW-NFC, future work should explore metrics derived from Control Flow Graph (CFG) analysis, such as "loop taint analysis," to capture complex data-flow anomalies without full symbolic execution.
(2) Cross-Language Generalization: Extending the semantic risk profiles (e.g., high-risk APIs) to other languages like Python or C++ would test the robustness and generalizability of our proposed risk theories across diverse development ecosystems.
(3) Explainability and Developer Trust: Incorporating explainable AI (XAI) techniques, such as attention visualization or rule-based explanations, could enhance the interpretability of predictions, thereby increasing developer trust and adoption in real-world CI/CD pipelines.

In summary, this work demonstrates that refining the semantic intelligence of heuristic-based models can yield meaningful improvements in practical defect prediction. By building upon the foundation laid by (Guo et al., 2023), we reinforce the idea that simple, interpretable, and efficient models can not only compete with but also guide the development of more advanced systems.

The replication package containing the GLANCE++ implementation and all experimental artifacts supporting the findings of this study is publicly available at: https://github.com/archazid/GLANCE (Mujaddid, 2025).

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to Zhaoqiang Guo, Shiran Liu, Xutong Liu, Wei Lai, Mingliang Ma, Xu Zhang, Chao Ni, Yibiao Yang, Yanhui Li, Lin Chen, Guoqiang Zhou, and Yuming Zhou for their seminal work and for making their complete replication kit publicly available. The dataset and implementation resources provided by the authors were instrumental in ensuring the reproducibility and validity of our experimental framework.

REFERENCES

Agrawal, A., Fu, W., Chen, D., Shen, X., & Menzies, T. (2021). How to "DODGE" complex software analytics. *IEEE Transactions on Software Engineering*, *47*(10), 2182–2194.

Ahmad, M. J., Goseva-Popstojanova, K., & Lutz, R. R. (2024). The untold impact of learning approaches on software fault-proneness predictions: An analysis of temporal aspects. *Empirical Software Engineering*, *29*(4), 87.

Assim, M., Obeidat, Q., & Hammad, M. (2020). Software defects prediction using machine learning algorithms. *2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI)* (pp. 1–6). Presented at the 2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI), Sakheer, Bahrain: IEEE. Retrieved August 6, 2025, from https://ieeexplore.ieee.org/document/9325677/

Bagaev, M., Khabibrakhmanova, A., Sabaev, G., & Bugayenko, Y. (2024). The impact of mutability on cyclomatic complexity in java. arXiv. Retrieved December 8, 2025, from https://arxiv.org/abs/2410.10425

Chen, J., Xu, J., Cai, S., Wang, X., Chen, H., & Li, Z. (2024). Software defect prediction approach based on a diversity ensemble combined with neural network. *IEEE Transactions on Reliability*, *73*(3), 1487–1501.

Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., & Adesope, O. (2020). Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, *25*(3), 2140–2178.

Fu, M., & Tantithamthavorn, C. (2022). LineVul: A transformer-based line-level vulnerability prediction. *Proceedings of the 19th International Conference on Mining Software Repositories* (pp. 608–620). Presented at the MSR '22: 19th International Conference on Mining Software Repositories, Pittsburgh Pennsylvania: ACM. Retrieved September 28, 2024, from https://dl.acm.org/doi/10.1145/3524842.3528452

Guo, Z., Liu, S., Liu, X., Lai, W., Ma, M., Zhang, X., Ni, C., et al. (2023). Code-line-level bugginess identification: How far have we come, and how far have we yet to go? *ACM Transactions on Software Engineering and Methodology*, *32*(4), 1–55.

Hata, H., Mizuno, O., & Kikuno, T. (2012). Bug prediction based on fine-grained module histories. *2012 34th International Conference on Software Engineering (ICSE)* (pp. 200–210). Presented at the 2012 34th International Conference on Software Engineering (ICSE 2012), Zurich: IEEE. Retrieved August 6, 2025, from http://ieeexplore.ieee.org/document/6227193/

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K., Adams, B., & Hassan, A. E. (2010). Revisiting common bug prediction findings using effort-aware models. *2010 IEEE International Conference on Software Maintenance* (pp. 1–10). Presented at the 2010 IEEE 26th International Conference on Software Maintenance (ICSM), Timi oara, Romania: IEEE. Retrieved August 5, 2025, from http://ieeexplore.ieee.org/document/5609530/

Kamei, Y., Monden, A., Matsumoto, S., Kakimoto, T., & Matsumoto, K. (2007). The effects of over and under sampling on fault-prone module detection. *First International Symposium on Empirical*

Software Engineering and Measurement (ESEM 2007) (pp. 196–204). Presented at the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain: IEEE. Retrieved August 5, 2025, from http://ieeexplore.ieee.org/document/4343747/

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., et al. (2017). LightGBM: a highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems* (Vol. 30). Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

Lavazza, L., Abualkishik, A. Z., Liu, G., & Morasca, S. (2023). An empirical evaluation of the "Cognitive Complexity" measure as a predictor of code understandability. *Journal of Systems and Software*, *197*, 111561.

Li, X., Jiang, J., Benton, S., Xiong, Y., & Zhang, L. (2021). A large-scale study on API misuses in the wild. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 241–252). Presented at the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), Porto de Galinhas, Brazil: IEEE. Retrieved August 8, 2025, from https://ieeexplore.ieee.org/document/9438601/

Mahbub, P., & Rahman, M. M. (2024). Predicting line-level defects by capturing code contexts with hierarchical transformers. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 308–319). Presented at the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland: IEEE. Retrieved September 28, 2024, from https://ieeexplore.ieee.org/document/10589817/

Mienye, I. D., & Sun, Y. (2022). A survey of ensemble learning: Concepts, algorithms, applications, and prospects. *IEEE Access*, *10*, 99129–99149.

Mujaddid, Z. (2025, December 8). archazid/GLANCE: V1.0.0. Zenodo. Retrieved December 9, 2025, from https://zenodo.org/doi/10.5281/zenodo.17858503

Oueslati, R., Ouertani, M. W., Amdouni, A., & Manita, G. (2025). MS-FSOA-LightGBM: Multi-strategy starfish optimization algorithm with LightGBM for software defect prediction. *Procedia Computer Science*, 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025), *270*, 4054–4063.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., & Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, *111*, 104773.

Parashar, A., Kumar Goyal, R., Kaushal, S., & Kumar Sahana, S. (2022). Machine learning approach for software defect prediction using multi-core parallel computing. *Automated Software Engineering*, *29*(2), 44.

Pascarella, L., Palomba, F., & Bacchelli, A. (2019). Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, *150*, 22–36.

Pereira, J. H. A., Souza, A. L. O. T. D., & Pinto, V. H. S. C. (2021). Cognitive load analyzer: A support tool for cognitive-driven development. *Brazilian Symposium on Software Engineering* (pp. 468–473). Presented at the SBES '21: Brazilian Symposium on Software Engineering, Joinville Brazil: ACM. Retrieved December 8, 2025, from https://dl.acm.org/doi/10.1145/3474624.3476011

Pornprasit, C., & Tantithamthavorn, C. K. (2023). DeepLineDP: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*, *49*(1), 84–98.

Qiu, Shaojian, Huang, H., Luo, J., Kuang, Y., & Luo, H. (2024). BAFLineDP: Code bilinear attention fusion framework for line-level defect prediction. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 1–12). Presented at the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland: IEEE. Retrieved October 12, 2024, from https://ieeexplore.ieee.org/document/10589784/

Qiu, Shaoming, E, B., He, J., & Liu, L. (2025). Survey of software defect prediction features. *Neural Computing and Applications*, *37*(4), 2113–2144.

Rahman, F., Khatri, S., Barr, E. T., & Devanbu, P. (2014). Comparing static bug finders and statistical prediction. *Proceedings of the 36th International Conference on Software Engineering* (pp. 424–434). Presented at the ICSE '14: 36th International Conference on Software Engineering, Hyderabad India: ACM. Retrieved August 8, 2025, from https://dl.acm.org/doi/10.1145/2568225.2568269

Roy, D., Pramanik, R., & Sarkar, R. (2024). Margin-aware adaptive-weighted-loss for deep learning based imbalanced data classification. *IEEE Transactions on Artificial Intelligence*, *5*(2), 776–785.

Segalotto, M., Bolzan, W., & Farias, K. (2023). Effects of modularization on developers' cognitive effort in code comprehension tasks: A controlled experiment. *Proceedings of the XXXVII Brazilian Symposium on Software Engineering* (pp. 206–215). Presented at the SBES 2023: XXXVII Brazilian Symposium on Software Engineering, Campo Grande Brazil: ACM. Retrieved December 8, 2025, from https://dl.acm.org/doi/10.1145/3613372.3613387

Sharma, G., & Singh, P. (2025). Comparative study: Word2Vec versus TF-IDF in software defect predictions. In S. Namasudra, N. Kar, S. K. Patra, & D. Taniar (Eds.), *Data Science and Network Engineering*, Lecture Notes in Networks and Systems (Vol. 1165, pp. 95–107). Singapore: Springer Nature Singapore. Retrieved August 18, 2025, from https://link.springer.com/10.1007/978-981-97-8336-6_8

Stradowski, S., & Madeyski, L. (2023). Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review. *Information and Software Technology*, *159*, 107192.

Szymański, K., & Ochodek, M. (2023). On the applicability of the pareto principle to source-code growth in open source projects (pp. 781–789). Presented at the 18th Conference on Computer Science and Intelligence Systems. Retrieved December 10, 2025, from https://annals-csis.org/Volume_35/drp/5221.html

Wan, Z., Xia, X., Hassan, A. E., Lo, D., Yin, J., & Yang, X. (2020). Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, *46*(11), 1241–1266.

Wang, H., Zhuang, W., & Zhang, X. (2021). Software defect prediction based on gated hierarchical LSTMs. *IEEE Transactions on Reliability*, *70*(2), 711–727.

Wang, S., Huang, L., Gao, A., Ge, J., Zhang, T., Feng, H., Satyarth, I., et al. (2023). Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering*, *49*(3), 1188–1231.

Wattanakriengkrai, S., Thongtanunam, P., Tantithamthavorn, C., Hata, H., & Matsumoto, K. (2022). Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, *48*(5), 1480–1496.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2024). *Experimentation in software engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved August 18, 2025, from https://link.springer.com/10.1007/978-3-662-69306-3

Yang, F., Zhong, F., Zeng, G., Xiao, P., & Zheng, W. (2024). LineFlowDP: A deep learning-based two-phase approach for line-level defect prediction. *Empirical Software Engineering*, *29*(2), 50.

Zhang, H., & Cheung, S. C. (2013). A cost-effectiveness criterion for applying software defect prediction models. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 643–646). Presented at the ESEC/FSE'13: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Saint Petersburg Russia: ACM. Retrieved August 8, 2025, from https://dl.acm.org/doi/10.1145/2491411.2494581

Zhao, Y., Damevski, K., & Chen, H. (2023). A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, *55*(10), 1–35.

Zhong, Y., Song, K., Lv, S., & He, P. (2021). An empirical study of software metrics diversity for cross-project defect prediction. (C. Chai, Ed.)*Mathematical Problems in Engineering*, *2021*, 1–11.

Zhou, Y., Yang, Y., Lu, H., Chen, L., Li, Y., Zhao, Y., Qian, J., et al. (2018). How far we have progressed in the journey? An examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology*, *27*(1), 1–51.

Zhu, J., Huang, Y., Chen, X., Wang, R., & Zheng, Z. (2023). SyntaxLineDP: A line-level software defect prediction model based on extended syntax information. *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)* (pp. 83–94). Presented at the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), Chiang Mai, Thailand: IEEE. Retrieved October 10, 2024, from https://ieeexplore.ieee.org/document/10366602/