How to Make Line-Level Defect Prediction More Effective?

Anonymous ACL submission

Abstract

Line-level defect prediction aims to precisely localize defect-prone code, yet its effectiveness is often limited by insufficient inter-line context modeling and weak coordination across granularities. To make line-level defect prediction more effective, We propose PHLDP, a **P**DG-to-sequence **H**ierarchical **L**ine-level **D**efect **P**rediction model that jointly learns defect patterns at both file and line levels. PHLDP improves effectiveness through (1) PDG-to-sequence conversion to preserve control and data dependencies, (2) hierarchical representation learning for local and global semantics, and (3) dual-level supervision to jointly optimize file-level and line-level predictions. Experiments on multiple open-source projects show that PHLDP consistently outperforms state-of-the-art baselines in both within-project and cross-project settings, particularly on effort-aware metrics, validating its practical effectiveness for line-level defect prediction. Our code is available at: https://anonymous.4open.science/r/PHLDP-7181/

1 Introduction

Software defects are a pervasive issue in software engineering, frequently causing severe economic losses (Tao et al., 2024). To mitigate these risks, software defect prediction (SDP) has emerged as an important research area, which aims to identify defective code early in the software development lifecycle, helping developers allocate testing resources more effectively and prioritize maintenance tasks (Huang et al., 2021; Zhu et al., 2023).

Recent advances in deep learning have led to significant progress in automated SDP (Hoang et al., 2019; Nong et al., 2022; Zain et al., 2023). However, most existing methods operate at coarse granularities (e.g., file-level or function-level), rendering them impractical for precise defect localization. Developers still face prohibitive inspection costs when reviewing entire files or functions to identify sparse defective lines.

Therefore, studying line-level defect prediction (LDP) is essential for improving code review efficiency. Several studies have explored this direction. For example, Wattanakriengkrai et al. (2022) apply model-agnostic interpretation to identify risky code tokens, while Pornprasit and Tantithamthavorn (2022) propose DeepLineDP, a hierarchical Bi-GRU model that performs file-level classification followed by risk-based line ranking. Yin et al. (2025) introduce LineDef, which employs graph convolutional networks to capture semantic dependencies across code lines.

Despite these advances, existing LDP methods still face two fundamental limitations: (1) **Insufficient inter-line context modeling.** As Figure 1, most methods treat code as flat sequences or independent lines, ignoring the rich semantic dependencies encoded in control flow and data flow across lines. While graph-based methods (e.g., graph neural networks) partially address this, they struggle to capture long-range dependencies critical for global code understanding. (2) **Lack of multi-granularity utilization.** State-Of-The-Art (SOTA) LDP models typically rely on coarse-grained supervision and post-hoc interpretability techniques to localize defective lines (Fu and Tantithamthavorn, 2022). This indirect supervision is typically derived from file-level labels, which may not accu-

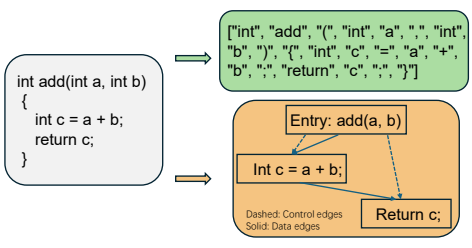


Figure 1: Illustration of Existing Sequence-Based and Graph-Based Code Modeling Paradigms

rately correspond to the true locations of defects, leading to suboptimal line-level predictions.

To address these challenges, we propose PHLDP, a **P**DG-to-sequence **H**ierarchical **L**ine-level **D**efect **P**rediction model. PHLDP performs joint file-level and line-level prediction using a hierarchical Transformer architecture based on Program Dependence Graph (PDG) sequence representations. We conduct extensive experiments on 32 releases from 9 open-source projects. PHLDP consistently outperforms SOTA baselines on both traditional file-level and effort-aware line-level metrics. Notably, it achieves substantial improvements in line-level Recall@Top20%LOC, with gains of up to 183.74% over existing methods.

In summary, our contributions are as follows:

- We adopt a novel PDG-to-sequence conversion method that effectively encodes both semantic and structural features of source code.
- We design a hierarchical Transformer-based architecture that jointly models line-level and file-level representations through a dual-granularity learning objective.
- We achieve remarkable performance on the benchmark dataset, consistently surpassing existing methods under both file-level and line-level evaluation metrics.

2 Related Work

This section reviews prior studies closely related to our work from two complementary perspectives: (1) code representation learning for SDP, and (2) prediction granularity in SDP models.

2.1 Code Representation for Defect Prediction

Learning effective code representations is central to SDP. Existing approaches can be broadly grouped into text-based, graph-based, and hybrid methods.

Text-based Representations. Text-based methods treat source code as token sequences and leverage neural sequence models to capture lexical and contextual patterns. Early work employed CNNs to identify defective code segments from token sequences (Russell et al., 2018), while more recent approaches adopt Transformer architectures for function- and line-level defect prediction (Fu and Tantithamthavorn, 2022; Nguyen and Vo, 2024). Despite their scalability and strong contextual modeling, purely text-based representations struggle to explicitly encode structural dependencies such as control and data flows.

Graph-based Representations. To model program structure explicitly, graph-based approaches construct representations based on control-flow or data-flow dependencies. Representative studies include probabilistic graphical models for fault localization (Yu et al., 2017) and GNN-based frameworks for statement-level or multi-line defect prediction (Zou et al., 2021; Cao et al., 2022). While effective at capturing structural relations, these methods often incur high computational overhead and face challenges in modeling long-range semantic dependencies.

Hybrid Representations. Hybrid approaches combine sequential and structural representations to exploit complementary strengths. Recent work integrates Transformers with graph neural networks (Hin et al., 2022) or fuses recurrent models with graph encoders (Rabheru et al., 2022; Cui et al., 2022). Although hybrid models improve representational capacity, their reliance on explicit graph construction limits scalability and deployment efficiency.

Overall, existing representation methods face a trade-off between structural expressiveness and computational efficiency, motivating more unified and scalable modeling paradigms.

2.2 Granularity of Defect Prediction

Another key dimension of SDP research is prediction granularity, which has evolved from coarse-grained to fine-grained levels to better support practical debugging.

Coarse-grained Prediction. Early studies focused on coarse-grained units such as files, functions, or commits. Representative approaches include file-level defect prediction using deep semantic models (Wang et al., 2020), commit-level detection pipelines (Yan et al., 2022), and function-level multi-modal frameworks (Ni et al., 2023). However, these methods provide limited support for precise fault localization.

Fine-grained Prediction. To enable actionable debugging, recent work increasingly targets fine-grained prediction, particularly at the line level. Existing methods employ post-hoc interpretation techniques (Wattanakriengkrai et al., 2022), hierarchical attention mechanisms (Pornprasit and Tantithamthavorn, 2022; Qiu et al., 2024), or structural

encoders such as GCNs (Yin et al., 2025). Despite notable progress, most line-level approaches rely on coarse-grained supervision or auxiliary localization signals, which limits their ability to accurately identify defective lines.

In contrast, our proposed PHLDP jointly models file-level and line-level defect prediction within a unified learning framework, enabling more precise and reliable defect localization.

3 Methodology

In this section, we introduce PHLDP, a **P**DG-to-sequence **H**ierarchical **L**ine-level **D**efect **P**rediction model. As illustrated in Figure 2, the proposed framework consists of three main phases: (1) source code processing and PDG sequence embedding, (2) hierarchical feature learning at both the line and file levels, and (3) dual-granularity defect prediction guided by a two-level loss function.

3.1 PDG Sequence Embedding

Prior studies have shown that a line of code is strongly influenced by its surrounding context (Kondo et al., 2020). However, modeling source code purely as sequences neglects structural and execution dependencies, while graph-based approaches incur high computational costs and limited long-range modeling capacity.

To balance efficiency and structural awareness, we adopt a graph-to-sequence conversion strategy inspired by Shi et al. (2022). As illustrated in Figure 3, we linearize a PDG sequence by traversing both control flow and data flow dependencies. Specifically, starting from the entry node a, we generate a sequential representation following execution paths: $a \xrightarrow{c} b_1 \xrightarrow{c} c \xrightarrow{c} d \rightarrow b_1 \rightarrow a \rightarrow b_2$, where dashed edges indicate control dependencies, and solid edges denote data dependencies. This serialized PDG sequence captures execution order and structural semantics, providing a rich representation for downstream learning.

After obtaining PDG sequences, we apply Byte Pair Encoding (BPE) tokenization to segment the code. Each token is mapped to a high-dimensional vector via an embedding layer. Assume that a file containing serialized PDG sequences consists of N lines of code. After tokenization, the i-th line is represented by a fixed-length token sequence of length k. The embedding of the i-th line is formulated as:

$$E_i = [e_{i,\text{CLS}}, e_{i,1}, e_{i,2}, \ldots, e_{i,k}],$$

where $e_{i,\text{CLS}}$ denotes the global contextual feature of the line.

3.2 Hierarchical Feature Learning

We introduce a hierarchical Transformer encoder (Wu et al., 2021) designed to efficiently learn line-level and file-level representations. This hierarchical architecture enhances the model's ability to capture dynamic code behaviors (e.g., data flow and control flow), which is essential for defect prediction tasks requiring comprehensive contextual understanding. The hierarchical encoder consists of three Transformer-based components.

3.2.1 Initial Line-Level Encoder

Lower layers of Transformer models tend to capture lexical and syntactic information (Zhang et al., 2023). To balance performance and efficiency, we partition the line-level Transformer, using the first six layers as the initial line-level encoder.

Assume that the j-th file contains N lines of code, with each line represented by k tokens. For the i-th line in the j-th file, the input embedding is:

$$E_{j,i} = [e_{j,i,\text{CLS}}, e_{j,i,1}, e_{j,i,2}, \ldots, e_{j,i,k}].$$

The line-level encoder processes each line independently and outputs contextualized token representations:

$$\begin{aligned} H_{j,i} &= \text{LineEncoder}(E_{j,i}), \\ H_{j,i} &= [h_{j,i,\text{CLS}}, h_{j,i,1}, \ldots, h_{j,i,k}]. \end{aligned}$$

3.2.2 File-Level Encoder

To capture inter-line dependencies, we first derive a fixed-length representation for each line by mean-pooling its contextualized token embeddings:

$$\bar{h}_{j,i} = \frac{1}{|H_{j,i}|} \sum_{t=1}^{|H_{j,i}|} h_{j,i,t},$$

The sequence of line representations $[\bar{h}_{j,1}, \bar{h}_{j,2}, \ldots, \bar{h}_{j,N}]$ is then fed into a file-level Transformer encoder:

$$h'_j = \text{FileEncoder}\big([\bar{h}_{j,1}, \bar{h}_{j,2}, \ldots, \bar{h}_{j,N}]\big),$$

where $h'_j \in \mathbb{R}^d$ encodes global contextual information for the entire file.

To improve efficiency and scalability, we incorporate sparse attention mechanisms (Yuan et al., 2025) in the file encoder, combining top-k attention and sliding window attention. This allows the model to focus on the most relevant code parts while reducing computational overhead.

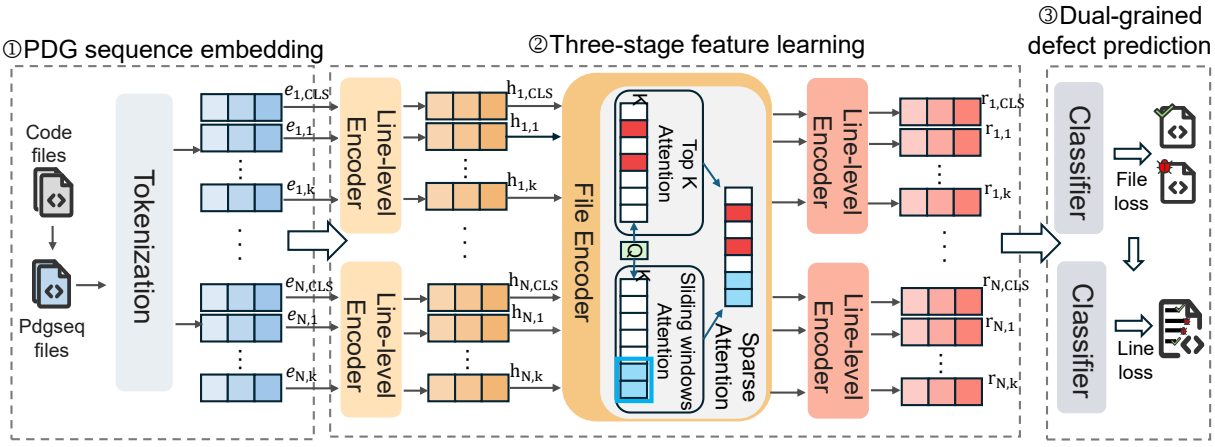


Figure 2: The architecture of the proposed PHLDP framework. It comprises three integrated components: (1) PDG Sequence Embedding, which transforms source code and Program Dependence Graphs (PDG) into tokenized representations; (2) Three-stage Feature Learning, leveraging a hierarchical structure with Line-level Encoders and a File Encoder equipped with Sparse Attention (Top-K and Sliding Window) to capture multi-scale dependencies; and (3) Dual-grained Defect Prediction, which performs joint optimization via File-level and Line-level classifiers for dual-granularity defect prediction.

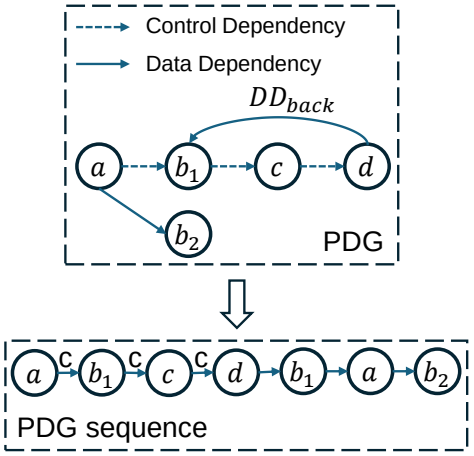


Figure 3: Illustration of PDG-to-sequence conversion capturing control and data dependencies.

3.2.3 Top Line-Level Encoder

Following Li et al. (2021), we use the last six layers of the line-level Transformer as the higher-level line encoder. To enhance line representations with global file context, we broadcast the file-level vector h'_j to each line and fuse it with the corresponding line representation $H_{j,i}$:

$$R_{j,i} = H_{j,i} \oplus h'_j,$$

where $\oplus$ denotes concatenation. The resulting sequence $R_{j,i}$ is then processed by the top layers of the line-level Transformer to produce context-enhanced token representations:

$$R_{j,i} = [r_{j,i,\text{CLS}}, r_{j,i,1}, \ldots, r_{j,i,k}],$$

which integrate both local line semantics and global file context. The final file-level representation H_j is used for file-level defect prediction, while $R_{j,i}$ is used for line-level defect prediction.

3.3 Dual-Granularity Defect Prediction

PHLDP jointly performs file-level and line-level defect prediction through a dual-granularity objective, allowing the model to exploit both coarse-grained file-level supervision and fine-grained line-level signals.

The file-level loss is defined using binary cross-entropy:

$$\mathcal{L}_{\text{file}} = \frac{1}{N} \sum_{j=1}^{N} \left[y_j^f \log P_j^f + (1 - y_j^f) \log(1 - P_j^f) \right], \tag{1}$$

where y_j^f denotes the ground-truth file label and P_j^f is the predicted probability.

Line-level supervision is available only for defective files, as non-defective files do not provide reliable annotations at the line granularity. Accordingly, the line-level loss is computed over annotated defective lines:

$$\mathcal{L}_{\text{line}} = \frac{1}{K} \sum_{i \in \mathcal{A}} [y_i \log P_i + (1 - y_i) \log(1 - P_i)] \tag{2}$$

where $\mathcal{A}$ denotes the set of annotated lines in defective files, y_i is the line label, P_i is the predicted probability, and $K = |\mathcal{A}|$.

To ensure that line-level supervision does not introduce noise from non-defective files, we adopt a

gated loss mechanism: the line-level loss is applied only when the corresponding file is defective. The overall training objective combines both losses:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{file}} + \lambda \cdot \mathbb{I}(y^f = 1) \cdot \mathcal{L}_{\text{line}}, \quad (3)$$

where λ balances the contribution of line-level supervision, and $\mathbb{I}(\cdot)$ is an indicator function.

During inference, PHLDP produces both file-level defect probabilities and line-level defect scores for all lines, enabling fine-grained defect localization while maintaining accurate file-level predictions.

4 Experimental Setup

4.1 Datasets

We utilize the publicly available line-level defect prediction dataset introduced by Pornprasit and Tantithamthavorn (2022), which contains 32 releases across 9 widely-used open-source projects. This dataset provides both file-level and line-level defect annotations, making it suitable for evaluating multi-granularity defect prediction models. Table 1 summarizes the key statistics of our dataset, while a detailed description of the dataset is provided in Appendix A.1.

Following standard practice, we adopt a temporal split strategy. For each project, the first release is used for training, the second release for validation, and the remaining releases for testing. This setup reflects realistic software evolution scenarios and avoids information leakage across time. This configuration results in 14 within-project and 112 cross-project prediction scenarios.

4.2 Evaluation Metrics

Following Pornprasit and Tantithamthavorn (2022), we evaluate file-level predictions with standard classification metrics and line-level predictions with effort-aware metrics.

Area Under the Curve (AUC). Quantifies the model's discriminative capability between defective and non-defective files and reflects the probability that a randomly chosen defective file is ranked higher than a randomly chosen clean file. Higher values correspond to better performance.

Balanced Accuracy (BA). BA is calculated as the average of sensitivity (true positive rate) and specificity (true negative rate), offering a balanced evaluation of classification performance, especially useful for imbalanced datasets. A higher value reflects better overall accuracy.

Matthews Correlation Coefficient (MCC). MCC assesses the overall quality of binary classifications by incorporating all four categories of the confusion matrix. It is particularly effective in scenarios with class imbalance. A value closer to 1 denotes stronger predictive consistency, and larger values indicate better performance.

Recall@Top20%LOC (Recall@20). The proportion of defective lines ranked within the top 20% of lines of code by predicted defect probability; higher values indicate more effective prioritization.

Effort@Top20%Recall (Effort@20). The proportion of code that must be inspected to identify 20% of defective lines; lower values indicate higher cost-efficiency.

Initial False Alarm (IFA). The number of clean lines inspected before the first defective line; lower values indicate fewer false positives and reduced inspection overhead.

4.3 Baselines

To comprehensively evaluate the effectiveness of PHLDP, we compare it against a broad range of defect prediction approaches, with particular emphasis on recent SOTA line-level models.

These SOTA methods represent the most competitive neural architectures for fine-grained defect localization, leveraging advanced contextual modeling, attention mechanisms, or graph-based representations. Specifically, LineVul (Fu and Tantithamthavorn, 2022) employs a Transformer architecture to perform function- and line-level defect prediction by modeling long-range token dependencies. DeepLineDP (Pornprasit and Tantithamthavorn, 2022) adopts a hierarchical Bi-GRU structure to capture both token-level semantics and inter-line contextual information. BAFLineDP (Qiu et al., 2024) further enhances line representations through bilinear attention fusion of semantic and contextual features, while LineFlowDP (Yang et al., 2024) leverages program dependence graphs and centrality measures to identify defect-prone lines from a control- and data-flow perspective. More recently, LineDef (Yin et al., 2025) applies graph convolutional networks to learn rich line-level representations from program graphs.

In addition to these SOTA approaches, we include representative file-level and line-level baselines that are widely adopted in prior studies. File-level methods such as Bi-LSTM, CNN, DBN, and Bag-of-Words focus on learning semantic or statistical representations of entire files for defect

Project	Files	Lines	Defect File Ratio	Defect Line Ratio	Releases
ActiveMQ	1884–3420	142k–299k	2%–7%	0.08%–0.44%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	1515–8846	75k–485k	2%–8%	0.09%–0.25%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	1963–2705	412k–533k	6%–28%	0.10%–0.63%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	757–884	74k–93k	2%–4%	0.10%–0.17%	1.5.7, 1.6.0-Beta1, 1.6.0-Beta2
HBase	1059–1834	246k–537k	7%–11%	0.17%–1.02%	0.94.0, 0.95.0, 0.95.2
Hive	1416–2662	290k–567k	6%–19%	0.31%–2.90%	0.9.0, 0.10.0, 0.12.0
JRuby	731–1614	106k–240k	2%–13%	0.03%–0.09%	1.1, 1.4, 1.5, 1.7
Lucene	805–2806	101k–342k	2%–8%	0.07%–0.39%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	1672–2578	106k–165k	2%–16%	0.05%–0.46%	1.3.0-beta1, 1.3.0-beta2, 1.5.3

Table 1: Overview of the experimental datasets used in our study.

classification. Line-level baselines, including ErrorProne, N-gram language models, and Random Forest classifiers with handcrafted features, serve as conventional approaches for fine-grained defect prediction. Together, these baselines provide a comprehensive comparison across different modeling granularities and methodological paradigms.

4.4 Training Details

PHLDP is implemented in PyTorch and initialized with the CodeT5+ pre-trained model (Wang et al., 2023). All experiments are conducted on a machine equipped with an Intel Xeon E5-2603 v4 CPU, 64 GB of RAM, and an NVIDIA RTX A5000 GPU with 24 GB of memory. The model is optimized using the AdamW optimizer with a learning rate of 1×10^{-5} and a batch size of 8, and trained for 10 epochs. To mitigate class imbalance, we employ a weighted binary cross-entropy loss. Model selection is based on the highest MCC score achieved on the validation set, as MCC is robust to imbalanced data distributions commonly found in defect prediction tasks.

4.5 Statistical Test

To ensure statistically robust performance comparisons, we apply the Scott-Knott Effect Size Difference (SK-ESD) test (Herbold, 2017). This hierarchical clustering technique groups model performances based on statistical significance and effect size. This method hierarchically clusters models based on both statistical significance and effect size (using Cohen's d (Cohen, 2013)), grouping models with insignificant differences or negligible effect sizes into the same rank.

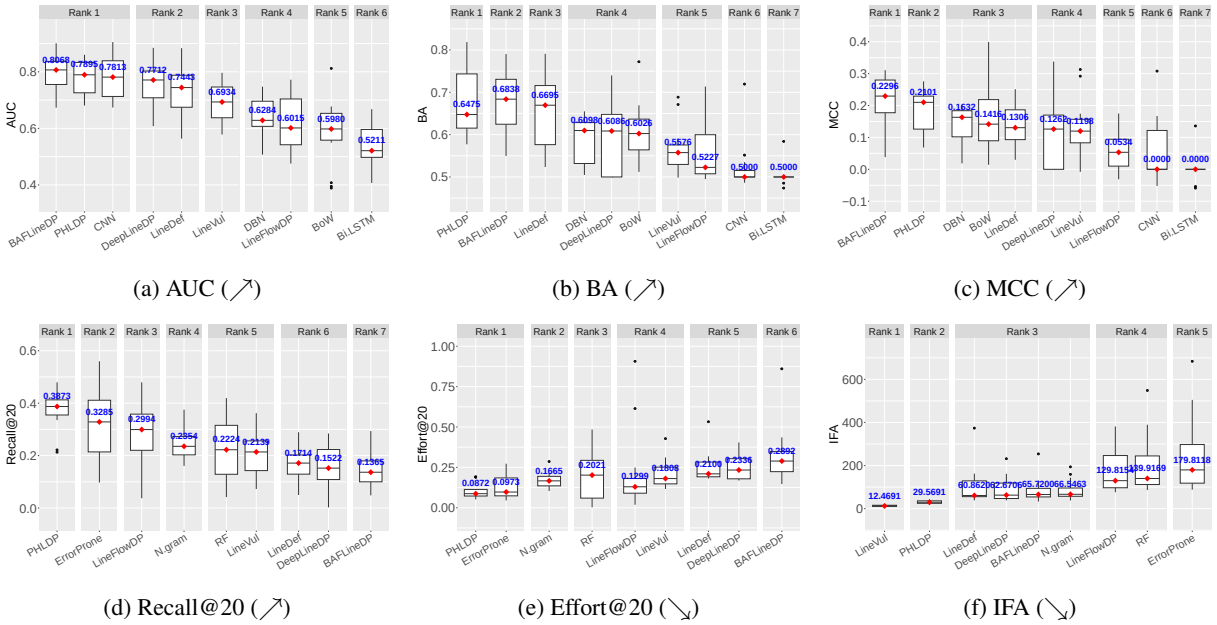
5 Experimental Results

5.1 Within-Project Results

Within-Project Defect Prediction (WPDP) leverages historical data from the same project to identify defects, which is particularly valuable during the early stages of software development. While most SOTA methods primarily focus on file-level prediction, line-level defect prediction remains relatively underexplored. To bridge this gap, we evaluate PHLDP under the WPDP setting at both file and line granularities.

Figure 4 reports the rankings and median metric distributions of SK-ESD under WPDP, covering three file-level metrics and three line-level metrics. PHLDP ranks first on four metrics and second on the remaining two, demonstrating statistically significant improvements with substantial effect sizes. At the file level, in terms of median performance, PHLDP achieved an AUC of 0.7895, a BA of 0.6475, and an MCC of 0.2101. Compared with the weakest baseline, PHLDP improved AUC by 0.2684 (50.51%), BA by 0.1475 (29.50%), and MCC by 0.2101. More substantial improvements are observed at the line level. PHLDP achieves a Recall@20 of 0.3873, an increase of 0.2508 (183.74%) compared to the weakest baseline. Effort@20 is reduced to 0.0072, corresponding to a 0.2020 (97.51%) decrease compared to the weakest baseline. In addition, IFA drops from 179.81 to 29.57, yielding an 83.56% reduction. Importantly, the SK-ESD test demonstrates that PHLDP consistently achieves the top ranks in both Recall@20 and Effort@20, with substantial effect sizes.

Although PHLDP does not achieve the highest MCC, it remains competitive with most baseline methods. Likewise, while its IFA performance is weaker than that of LineVul, PHLDP is expected to be more practically useful. This is because LineVul was originally designed as a function-level approach that processes at most 512 tokens, truncating any excess input. Consequently, a substantial portion of files is only partially processed, which can lead to inflated false positives in IFA.



(a) AUC (↗) (b) BA (↗) (c) MCC (↗)

(d) Recall@20 (↗) (e) Effort@20 (↘) (f) IFA (↘)

Figure 4: SK-ESD results under the within-project defect prediction (WPDP) setting.

5.2 Cross-Project Results

Cross-Project Defect Prediction (CPDP) is considerably more challenging than WPDP due to pronounced distributional and structural discrepancies between source and target projects. Such domain shifts often lead to severe performance degradation for conventional SOTA methods.

As shown in Figure 5, PHLDP further demonstrates strong generalization capability under domain shift in the CPDP setting, as evidenced by its favorable rankings and median metric distributions. At the file level, based on median values, PHLDP achieves an AUC of 0.7549, improving by 0.2636 (53.65%) over the weakest baseline. It also attains a median BA of 0.6442, corresponding to a gain of 0.1442 (28.84%). For MCC, PHLDP reaches 0.1542, yielding a 107.80% relative improvement the weakest baseline. At the line level, PHLDP attains a median Recall@20 of 0.2616, representing a 108.11% increase over the lowest baseline value. Meanwhile, Effort@20 is reduced to 0.1514, and IFA drops to 31.0235, marking an 81.44% reduction compared to the worst-performing baseline. Overall, PHLDP ranks first in three out of the six evaluated metrics under CPDP.

Although BAFLineDP slightly surpasses PHLDP in file-level AUC and MCC, this advantage primarily stems from its LSTM-based encoder, which excels at capturing long-range sequential dependencies at the file level. However, this architectural strength does not translate to line-level prediction tasks and fails to deliver comparable inspection efficiency. In contrast, PHLDP's dual-granularity design effectively integrates both file-level and line-level information, resulting in more balanced and robust performance across evaluation metrics. Furthermore, similar to WPDP, LineVul exhibits incorrect IFA measurements, highlighting limitations in its ability to prioritize defective lines accurately.

5.3 Ablation Study

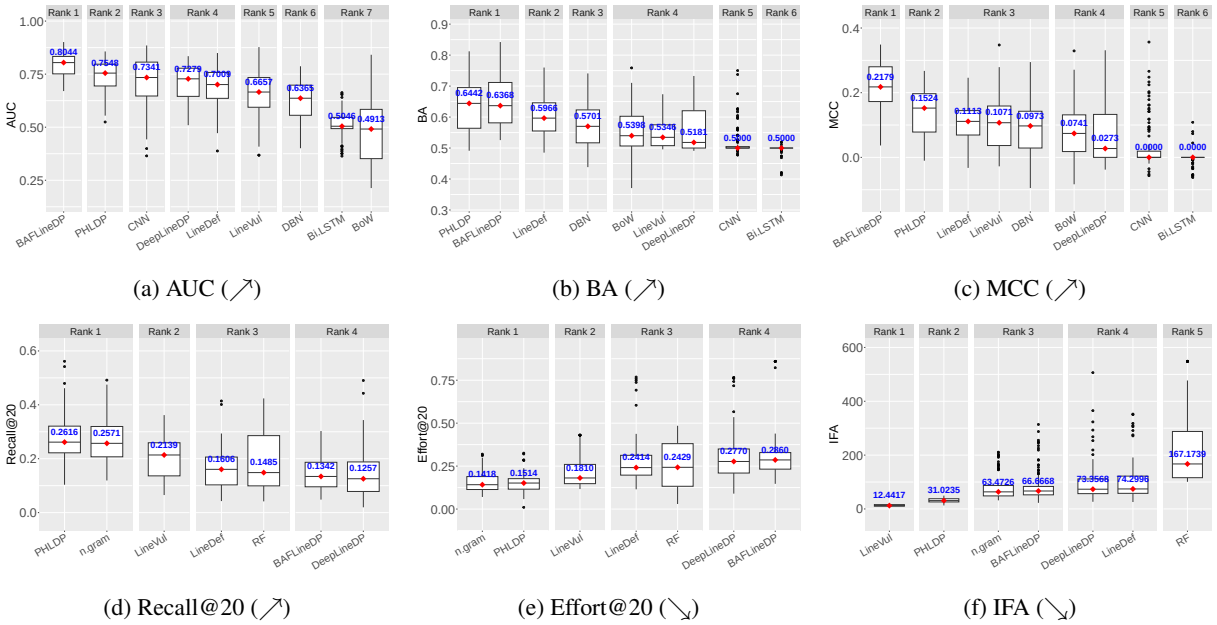
We conduct an ablation study under the WPDP setting to quantify the contributions of three key components: PDG-based input sequences, sparse attention, and dual-granularity modeling. All reported results are averaged over all projects under the WPDP setting.

Effect of PDG-based Sequences.

We evaluate the impact of PDG-based sequences by replacing them with original token-based code representations. As shown in Table 2, removing PDG-based inputs leads to clear degradation in line-level performance, with Recall@20 and Effort@20 being most affected. This suggests that PDG-based sequences provide richer structural and semantic cues that are beneficial for fine-grained defect localization.

Effect of Sparse Attention.

To assess the role of sparse attention, we ablate the sparse attention mechanism while keeping other components unchanged. As shown in Table 2, this



(a) AUC (↗) (b) BA (↗) (c) MCC (↗)

(d) Recall@20 (↗) (e) Effort@20 (↘) (f) IFA (↘)

Figure 5: SK-ESD results under the cross-project defect prediction (CPDP) setting.

Method	AUC	BA	MCC	Recall@20	Effort@20	IFA
PHLDP (full model)	0.7793	0.6797	0.1850	**0.3749**	**0.0993**	**30.3201**
w/o pdgseq (only original code sequence)	0.7793	0.7128	0.2029	0.3215	0.1286	30.4318
w/o sparse attention	0.7755	0.6998	0.1921	0.2946	0.1952	37.5683
w/o line-level loss	**0.7965**	**0.7161**	**0.2121**	0.1898	0.2717	52.9507

Table 2: Ablation study under the within-project setting. The best values for each metric are highlighted in bold.

variant exhibits noticeably worse line-level effectiveness, including lower Recall@20 and higher Effort@20 and IFA. These results indicate that sparse attention helps suppress irrelevant context in long code sequences and improves localization precision.

Effect of Dual-Granularity Modeling.

We further examine the contribution of dual-granularity modeling by training a variant with only file-level supervision. As reported in Table 2, removing line-level supervision substantially weakens line-level performance. This confirms that jointly optimizing file-level and line-level objectives is critical for accurate defect localization.

Overall, the ablation study confirms that all components are essential to the effectiveness of PHLDP.

6 Conclusion

This paper addressed the question of how to make line-level defect prediction more effective by introducing PHLDP, a hierarchical framework that jointly models file-level and line-level supervision. By integrating structurally-aware PDG representations with multi-granularity learning, PHLDP enhances line-level prediction effectiveness through improved contextual modeling and better alignment between global and local defect signals. Extensive experiments show that PHLDP consistently outperforms state-of-the-art methods across diverse project settings, with particularly strong gains on effort-aware metrics that reflect practical inspection costs. These results demonstrate that effectiveness in line-level defect prediction crucially depends on jointly modeling structural dependencies and supervision across granularities. Future work will explore richer program semantics and further improve generalization in cross-project scenarios.

7 Limitations

While PHLDP demonstrates effectiveness, this study has limitations.

Dataset Scope. Our evaluation is based on a benchmark of Java projects. The generalizability to other programming languages, domains (e.g., proprietary systems), or development practices remains to be verified. Future work should assess the approach on more diverse and industrial-scale

systems.

Evaluation Metrics. Our evaluation employs standard accuracy and cost-effectiveness metrics, but they do not capture all practical aspects, such as defect severity or developer usability. Incorporating additional measures like precision or conducting user studies would offer a more holistic assessment of practical utility.

References

Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*, pages 1456–1468.

Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. routledge.

Ningning Cui, Liwei Chen, Gewangzi Du, Tongshuai Wu, Chenguang Zhu, and Gang Shi. 2022. Bhmvd: binary code-based hybrid neural network for multiclass vulnerability detection. In *2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 238–245. IEEE.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620.

Steffen Herbold. 2017. Comments on scottknottesd in response to" an empirical comparison of model validation techniques for defect prediction models". *IEEE Transactions on Software Engineering*, 43(11):1091–1094.

David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607.

Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE.

Yuan Huang, Jinyu Jiang, Xiapu Luo, Xiangping Chen, Zibin Zheng, Nan Jia, and Gang Huang. 2021. Change-patterns mapping: A boosting way for change impact analysis. *IEEE transactions on software engineering*, 48(7):2376–2398.

Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical software engineering*, 25:890–939.

Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. 2021. Align before fuse: Vision and language representation learning with momentum distillation. *Advances in neural information processing systems*, 34:9694–9705.

Thu-Trang Nguyen and Hieu Dinh Vo. 2024. Context-based statement-level vulnerability localization. *Information and Software Technology*, 169:107406.

Chao Ni, Xinrong Guo, Yan Zhu, Xiaodan Xu, and Xiaohu Yang. 2023. Function-level vulnerability detection through fusing multi-modal knowledge. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1911–1918. IEEE.

Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering*, 49(4):1983–2005.

Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2022. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*, 49(1):84–98.

Shaojian Qiu, Huihao Huang, Jianxiang Luo, Yingjie Kuang, and Haoyu Luo. 2024. Baflinedp: Code bilinear attention fusion framework for line-level defect prediction. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE.

Rishi Rabheru, Hazim Hanif, and Sergio Maffeis. 2022. A hybrid graph neural network approach for detecting php vulnerabilities. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–9. IEEE.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE.

Yucen Shi, Ying Yin, Zhengkui Wang, David Lo, Tao Zhang, Xin Xia, Yuhai Zhao, and Bowen Xu. 2022. How to better utilize code graphs in semantic code search? In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 722–733.

Hongwei Tao, Xiaoxu Niu, Lang Xu, Lianyou Fu, Qiaoling Cao, Haoran Chen, Songtao Shang, and Yang

Xian. 2024. A comparative study of software defect binomial classification prediction models based on machine learning. *Software Quality Journal*, 32(3):1203–1237.

Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2020. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 48(5):1480–1496.

Chuhan Wu, Fangzhao Wu, Tao Qi, and Yongfeng Huang. 2021. Hi-transformer: Hierarchical interactive transformer for efficient and effective long document modeling. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 848–853. Association for Computational Linguistics.

Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. 2022. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, 48(1):82–101.

Fengyu Yang, Fa Zhong, Guangdong Zeng, Peng Xiao, and Wei Zheng. 2024. Lineflowdp: A deep learning-based two-phase approach for line-level defect prediction. *Empirical Software Engineering*, 29(2):50.

Shouyu Yin, Shikai Guo, Hui Li, Chenchen Li, Rong Chen, Xiaochen Li, and He Jiang. 2025. Line-level defect prediction by capturing code contexts with graph convolutional networks. *IEEE Transactions on Software Engineering*, 51(1):172–191.

Xiao Yu, Jin Liu, Zijiang Yang, and Xiao Liu. 2017. The bayesian network based program dependence graph and its application to fault localization. *Journal of Systems and Software*, 134:44–53.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, and 1 others. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*.

Zuhaira Muhammad Zain, Sapiah Sakri, and Nurul Halimatul Asmak Ismail. 2023. Application of deep learning in software defect prediction: Systematic literature review and meta-analysis. *Information and Software Technology*, 158:107175.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*.

Jianzhong Zhu, Yuan Huang, Xiangping Chen, Ruomei Wang, and Zibin Zheng. 2023. Syntaxlinedp: a line-level software defect prediction model based on extended syntax information. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 83–94. IEEE.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236.

A Appendix

A.1 Dataset Details

Table 1 summarizes the datasets used in this study. The dataset comprises 32 releases from 9 Java-based open-source systems, including ActiveMQ, Camel, Derby, Groovy, HBase, Hive, JRuby, Lucene, and Wicket. For each project, we report the range of source code files and lines of code across releases, together with the corresponding defect file ratios and defect line ratios.

Across all projects, the number of files per release ranges from 731 to 8,846, while the total lines of code vary from approximately 74K to 567K, reflecting substantial diversity in project scale. Defective files account for 2% to 28% of all files, whereas defective lines are much rarer, ranging from only 0.03% to 2.90% of the total lines of code. This highlights the severe class imbalance at the line level. The final column lists the specific releases used for each project. Overall, the datasets cover a wide range of code sizes and defect distributions, providing a realistic and challenging testbed for evaluating line-level defect prediction models.

A.2 File-level Average Results Table

Table 3 reports the average file-level defect prediction performance under both within-project and cross-project settings, evaluated using AUC, BA, and MCC.

Under the within-project setting, PHLDP achieves the highest BA (0.6797), indicating a better balance between defective and non-defective files under class imbalance. It also attains the second-highest AUC (0.7793) and MCC (0.1850),

Model	Within-project			Cross-project		
	AUC (↗)	BA (↗)	MCC (↗)	AUC (↗)	BA (↘)	MCC (↘)
BOW	0.5831	0.6074	0.1633	0.4747	0.5558	0.0798
Bi-LSTM	0.5436	0.5031	0.0017	0.5133	0.4963	-0.0025
DBN	0.6380	0.5901	0.1332	0.6244	0.5762	0.0955
CNN	0.7859	0.5218	0.0552	0.7094	0.5186	0.0337
DeepLineDP	0.7503	0.5864	0.1083	0.7047	0.5607	0.0707
BAFLineDP	**0.7938**	0.6734	**0.2203**	**0.7929**	**0.6500**	**0.2138**
LineVul	0.6912	0.5645	0.1289	0.6550	0.5467	0.1052
LineFlowDP	0.6188	0.5575	0.0565	–	–	–
LineDef	0.7349	0.6481	0.1368	0.6946	0.6053	0.1104
PHLDP (ours)	0.7793	**0.6797**	0.1850	0.7446	0.6361	0.1420

Table 3: File-level defect prediction performance under within-project and cross-project settings. **Bold** and underline indicate the best and second-best results, respectively.

demonstrating competitive discrimination ability and stable correlation with ground-truth labels. Compared with representative learning-based baselines such as DeepLineDP, PHLDP improves BA by 0.0933 and MCC by 0.0767, suggesting that its gains are not limited to threshold-independent metrics.

In the cross-project setting, performance degradation is observed across all models due to distribution shift. Nevertheless, PHLDP consistently ranks second on all three metrics, achieving an AUC of 0.7446, a BA of 0.6361, and an MCC of 0.1420. Relative to traditional baselines such as BOW and Bi-LSTM, PHLDP yields notable improvements, particularly in BA and MCC, indicating stronger robustness under unseen project distributions.

Overall, PHLDP demonstrates stable and well-balanced performance across both evaluation settings. While it is not always the top-performing method in terms of AUC alone, its consistently strong BA and MCC suggest more reliable decision boundaries under class imbalance and cross-project variability.

A.3 Line-level Average Results Table

Table 4 reports the average line-level defect prediction performance under within-project and cross-project settings, evaluated using Recall@20, Effort@20, and IFA.

In the within-project setting, PHLDP achieves the best overall performance across all three metrics. It attains the highest Recall@20 (0.3749), indicating that a large proportion of defective lines can be identified by inspecting only the top-ranked lines. At the same time, PHLDP yields the lowest Effort@20 (0.0993), reflecting reduced inspection cost. PHLDP also achieves the second-lowest IFA, substantially reducing the number of non-defective lines examined before encountering the first defect.

The advantage of PHLDP persists in the more challenging cross-project setting. Despite the overall performance drop caused by distribution shift, PHLDP attains the highest Recall@20 (0.2785) and the lowest Effort@20 (0.1530), while maintaining the second-best IFA. These results indicate that PHLDP generalizes more robustly across projects while preserving inspection efficiency.

In summary, PHLDP achieves a favorable trade-off among defect recall, inspection effort, and early defect discovery at the line level. Its consistent performance across both evaluation settings suggests its suitability for practical defect localization scenarios.

A.4 Line-level Defect Prediction Case Study

Table 5 presents a qualitative case study of line-level defect prediction on a real-world bug from ActiveMQ 5.3.0 (`cursors/AbstractStoreCursor.java`). The reported values denote the normalized likelihood that a given line is defective.

The defect is related to duplicated message handling in the store cursor logic and becomes observable in the debug logging branch (Lines 88–89), where duplicated messages are explicitly reported. Although the logging statements themselves do not introduce the defect, they are strongly correlated with the faulty execution path and thus provide informative localization signals.

As shown in Table 5, DEEPLINEDP and LINEDEF assign high defect scores to a contiguous block of statements preceding the conditional branch (Lines 83–86), and also yield elevated scores for the guard at Line 88. This pattern indicates a tendency to propagate defect likelihood across nearby state updates and control-flow-

Model	Within-project			Cross-project		
	Recall@20 (↗)	Effort@20 (↘)	IFA (↘)	Recall@20 (↗)	Effort@20 (↘)	IFA (↘)
ErrorProne	<u>0.3233</u>	<u>0.1261</u>	247.8348	–	–	–
N-gram	0.2468	0.1695	86.2584	<u>0.2684</u>	<u>0.1605</u>	82.3539
RF	0.2274	0.2038	196.8168	0.1803	0.2479	224.7291
DeepLineDP	0.1572	0.2515	87.8985	0.1432	0.3013	96.8802
BAFLineDP	0.1472	0.3148	83.5999	0.1498	0.3145	86.6579
LineVul	0.2067	0.2058	**12.8684**	0.2069	0.2062	**12.8710**
LineFlowDP	0.2843	0.2089	173.7681	–	–	–
LineDef	0.1649	0.2517	105.1442	0.1605	0.2718	100.7546
PHLDP (ours)	**0.3749**	**0.0993**	<u>30.3201</u>	**0.2785**	**0.1530**	<u>31.4503</u>

Table 4: Line-level defect prediction performance under within-project and cross-project settings. **Bold** and <u>underline</u> indicate the best and second-best results, respectively.

Line	Code Snippet	PHLDP	BAFLineDP	DeepLineDP	LineDef
83	`message.incrementReferenceCount();`	0.0041	0.1816	**0.9990**	**0.9999**
84	`batchList.put(message.getMessageId(), message);`	0.0870	0.5437	**0.9967**	**1.0000**
85	`clearIterator(true);`	0.0507	0.3990	**0.9909**	**0.9995**
86	`recovered = true;`	0.0000	0.0193	**0.9946**	**0.9925**
87	`} else {`	–	0.3053	0.9431	0.9409
88	**`if (LOG.isDebugEnabled()) {`**	0.0011	0.1998	**1.0000**	0.9142
89	**`LOG.debug(... "cursor got duplicate: " + message);`**	**1.0000**	**1.0000**	0.9935	0.6446
91	`storeHasMessages = true;`	0.0493	0.5713	**0.9956**	0.1432
93	`return recovered;`	0.0076	0.0000	0.0000	0.0000

Table 5: Line-level defect prediction scores for a real bug in ActiveMQ 5.3.0 (`cursors/AbstractStoreCursor.java`). Higher values indicate higher defect likelihood.

related statements, resulting in relatively coarse-grained localization.

In contrast, PHLDP and BAFLINEDP produce more localized predictions, assigning peak scores primarily to the logging statement at Line 89, while substantially down-weighting surrounding state-modifying lines. The conditional guard at Line 88 also receives comparatively lower scores, suggesting a stronger reliance on semantic cues associated with anomalous behavior rather than positional proximity.

Overall, this case study illustrates how different modeling assumptions lead to distinct interpretations of defect locality at the line level. Qualitative analysis of such cases complements aggregate metrics by providing insights into the practical behavior of line-level defect prediction models.