e-ISSN 3108-026X
International Multidisciplinary Research Journal Reviews (IMRJR)
A Peer-reviewed journal
Volume 3, Issue 3, March 2026 | DOI 10.17148/IMRJR.2026.030304

Quantified Explainability and Robustness Analysis of Transformer-Based Bug Detection Models

Debargha Ghosh[1], Eve Thullen Ph.D.[2], Emmanuel Udoh Ph.D.[3]

Thullen Research Lab, United States[1,2]

University of the Cumberlands, United States[2,3]

Abstract: This paper investigates how to build, systematically configure, and rigorously explains a transformer-based bug detection system. The central argument is that trustworthy explainability requires first establishing which model is worth explaining and in which configuration. We evaluate three approaches on the lrhammond/buggy-apps dataset (8,778 balanced samples): a TF-IDF baseline, DistilBERT, and GraphCodeBERT. DistilBERT exhibited mode collapse across all tested learning rates, confirming that general-purpose language model pretraining is insufficient for code defect detection — a prerequisite finding that motivates the choice of GraphCodeBERT for explainability analysis. A systematic ablation across three stride configurations (128, 256, 384 tokens) and three aggregation strategies (max, mean, majority vote) yields nine experimental conditions; stride 256 with mean aggregation is the optimal configuration (70.77% accuracy, 69.56% macro F1, $p = 5.26 \times 10^{-33}$ vs TF-IDF, McNemar's test). Explainability analysis via attention rollout and integrated gradients across ten test samples reveals that integrated gradient signal strength is 11.8× stronger for correctly detected bugs than for misclassified samples — providing a gradient-based, quantitative explanation of model failure modes. Attention rollout and integrated gradients show near-zero cross-method correlation (mean $r = -0.017$), empirically confirming they are non-redundant and complementary methods.

Keywords: Explainable AI, Bug Detection, Transformer Models, GraphCodeBERT, Mode Collapse, Attention Rollout, Integrated Gradients, Sliding Window Ablation

I. INTRODUCTION

Software bugs represent a persistent and costly challenge in modern software development. Traditional static analysis tools, while useful, frequently produce high false-positive rates and struggle with complex code patterns. Transformer-based models pretrained on large code corpora have opened new avenues for automated bug detection, but their adoption in practice depends on more than raw performance — it depends on whether developers can understand and trust the decisions these models make.

The black-box nature of deep learning models raises fundamental concerns about trustworthiness in production environments. Developers need to understand why a model flags certain code as buggy to make informed decisions about code review priorities. Explainability techniques such as attention rollout and integrated gradients have been proposed to address this, but their application to code intelligence raises an important methodological question: before analyzing what a model attends to, one must first establish that the model has learned something meaningful to attend to. This work is organized around that question. To produce trustworthy explanations, three things must be established in sequence. First, which model architecture is capable of learning the bug detection task — general-purpose language models or code-specific ones? Second, within the capable model, which configuration of the sliding window mechanism produces the best and most reliably explainable predictions? Third, what do the resulting explanations reveal about both model successes and failures? This paper answers all three questions.

The contributions of this work are fourfold. First, we demonstrate empirically that code-specific pretraining is not merely advantageous but necessary: DistilBERT failed to converge under any tested learning rate, establishing a clear boundary between models that can and cannot learn code semantics. Second, we provide a systematic ablation across nine stride–aggregation configurations, identifying stride 256 with mean aggregation as the optimal sliding window setting and characterizing the performance impact of each design choice. Third, and most centrally, we show that integrated gradient signal strength quantitatively distinguishes correctly classified from misclassified samples — with an 11.8× magnitude difference — providing a gradient-based explanation of failure modes rather than mere performance reporting. Fourth, we establish empirically that attention rollout and integrated gradients are non-redundant methods, showing near-zero cross-method correlation across ten samples, a result that empirically supports recommendations in the XAI literature to use both methods together.

II. RELATED WORK

Transformer architectures have been successfully adapted for code understanding tasks. Models such as CodeBERT [2] and GraphCodeBERT [3] demonstrate superior performance on bug detection, code summarization, and vulnerability identification by leveraging pretraining on large code corpora. GraphCodeBERT extends RoBERTa [9] with data flow graphs during pretraining, learning relationships between code variables and their usage patterns. DistilBERT [4], trained on general English text, offers 60% faster inference than BERT but lacks code-specific inductive bias — a limitation our experiments confirm empirically through observed mode collapse.

Prior work on transformer-based bug detection reports aggregate performance metrics but rarely evaluates sliding window configuration systematically. When long code sequences are processed, the choice of window stride and chunk aggregation strategy directly affects both which code patterns the model sees during training and how predictions are combined at inference time. This gap motivates the ablation study presented in Section V-B, which is, to our knowledge, the first systematic evaluation of stride and aggregation strategy for this task.

The application of explainability techniques to software engineering remains an emerging area. Attention mechanisms provide a natural avenue for explainability; however, Jain and Wallace [11] established that attention weights do not reliably indicate feature importance in NLP models — a finding our results extend empirically to the code domain. Integrated gradients, introduced by Sundararajan et al. [1], offer a principled gradient-based alternative satisfying the axioms of sensitivity and implementation invariance. Abnar and Zuidema [12] proposed attention rollout for propagating attention weights through transformer layers to obtain more faithful token importance estimates. Most prior studies in software engineering report single-sample XAI results; we extend this to ten samples with quantitative cross-method comparison and a signal-strength analysis that distinguishes correct from failed predictions.

Effective bug detection requires appropriate code representation. Traditional approaches use abstract syntax trees or control flow graphs. GraphCodeBERT incorporates data flow information during pretraining, creating semantically rich representations that capture variable relationships and program structure beyond surface-level token sequences [3]. This architectural advantage is confirmed in our experiments: GraphCodeBERT achieves 70.77% accuracy while DistilBERT collapses, demonstrating that for code intelligence tasks, the representational foundation determines whether the model learns at all.

III. METHODOLOGY

A. Datasets and Processing

We utilize the lrhammond/buggy-apps dataset [7] from Hugging Face, containing competitive programming solutions labelled as correct or buggy. After flattening the nested structure, the dataset comprises 8,778 samples: 4,389 correct and 4,389 buggies, yielding a perfectly balanced 50/50 class distribution that eliminates class imbalance as a confounding factor. Because the dataset is perfectly balanced, accuracy and macro F1 are equivalent measures of performance, and both are reported for completeness. Data are split 70/15/15 into training (6,144), validation (1,317), and test (1,317) sets using stratified sampling to preserve class balance in each split.

Token length analysis using the GraphCodeBERT tokeniser reveals substantial variation: mean 252.6 tokens, median 174 tokens, maximum 4,130 tokens, with 11.1% of samples exceeding the 512-token model limit. Figure 1 illustrates this distribution. Crucially, token length distributions are nearly identical between correct and buggy classes (mean 258 vs 247), confirming that sequence length cannot itself discriminate between classes and that length-based shortcuts are not available to the model.

e-ISSN 3108-026X
International Multidisciplinary Research Journal Reviews (IMRJR)
A Peer-reviewed journal
Volume 3, Issue 3, March 2026 | DOI 10.17148/IMRJR.2026.030304

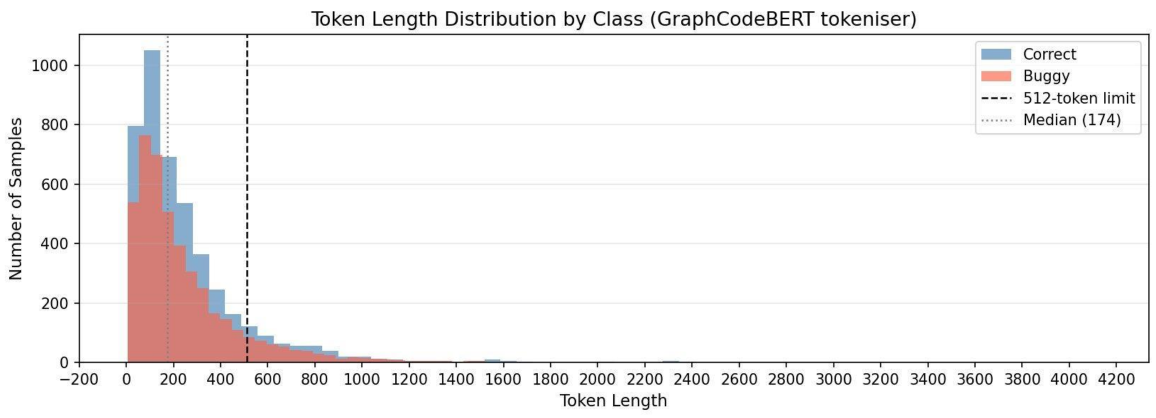


Fig. 1 Token length distribution by class (GraphCodeBERT tokeniser). Dashed line at 512 tokens marks the model's context limit; 11.1% of samples fall to the right. Near-identical distributions between correct and buggy classes confirm length is not a discriminative feature.

B. Model Architecture

TF-IDF Baseline: We implement a TF-IDF vectorizer with unigram and bigram features (max 50,000 features, sublinear TF scaling) paired with a logistic regression classifier (max 1,000 iterations). This traditional approach treats code as a bag of words and serves as the lower-bound performance reference.

DistilBERT [4] is a distilled version of BERT offering approximately 60% faster inference while retaining 97% of BERT's language understanding capabilities on general English tasks. We fine-tune it for binary sequence classification with a classification head on the [CLS] token. As detailed in Section V-A, DistilBERT exhibited mode collapse under both tested learning rates (3×10^{-5} and 1×10^{-5}), failing to learn the task entirely. This negative result is substantive: it establishes that general-purpose pretraining is insufficient for code defect detection, regardless of learning rate tuning, and motivates the focus on GraphCodeBERT for all subsequent analysis.

GraphCodeBERT [3] extends the RoBERTa architecture by incorporating data flow graphs during pretraining, learning relationships between code variables and their usage patterns. We use the Microsoft/graphcodebert-base checkpoint (124.6M parameters) and fine-tune it for binary bug detection. GraphCodeBERT is the primary model of this study, selected on the basis of its code-specific representational foundation.

C. Sliding Window Training and Ablation

Code sequences exceeding 512 tokens are handled using a sliding window mechanism. Each sample is tokenized in full, then partitioned into overlapping chunks of at most 512 tokens. At training time, each chunk is treated as an independent example with the parent sample label. At inference time, predictions from multiple chunks are aggregated into a single sample-level prediction. We conduct a systematic ablation across three stride values (128, 256, 384 tokens) and three aggregation strategies, requiring nine independent training runs. Each stride configuration trains a distinct model, as stride affects both what the model learns and how code is partitioned at inference time.

The three aggregation strategies are: Max — the window with the highest buggy-class probability determines the outcome, maximally sensitive to localized buggy patterns; Mean — average softmax probabilities across all windows, balancing evidence across the full file; Majority Vote — each window independently votes and the majority wins, robust to single noisy windows but discarding probability magnitude information.

D. Training Configuration

All models use the AdamW optimizer [14] with learning rate 3×10^{-5}, weight decay 0.01, gradient clipping at max norm 1.0, and linear warmup over the first 10% of training steps followed by linear decay. Batch size is 8 with mixed-precision (FP16) training on an NVIDIA L4 GPU (23.7 GB VRAM). Early stopping is applied with patience 3 epochs and a maximum of 15 epochs; the best checkpoint by validation macro F1-score is saved and reloaded for final test evaluation. All experiments use a fixed random seed of 42 for reproducibility.

E. Explainability Techniques

Attention rollout [12] propagates attention weights through all transformer layers by computing $R_l = \text{normalise}(A_l + I) \times R_{l-1}$, where A_l is the mean-head attention matrix at layer l and I is the identity matrix representing the residual connection. The CLS token's rollout scores serve as token importance estimates, mapped to line level by mean pooling over tokens within each line.

Integrated gradients [1] attribute predictions to token embeddings by integrating gradients along a straight-line path from a zero embedding baseline to the actual embeddings over 50 steps. Token-level scores are the sum over embedding dimensions of (mean gradient) × (embedding − baseline). Positive scores indicate evidence for the buggy class; negative scores indicate evidence against. Line-level scores are computed by mean pooling over tokens within each line. All models are fine-tuned locally; no external API-based inference is used, ensuring full access to model internals required for gradient computation.

Sample Selection: XAI analysis is conducted on ten test samples selected by random sampling (seed 42): five correctly classified buggy samples (true = 1, pred = 1) and five false negatives (true = 1, pred = 0). This contrast is the central comparison of the explainability analysis: it asks whether the method can distinguish between cases where the model succeeds and cases where it fails on the same ground-truth class.

IV. EXPERIMENTAL SETUP

A. Implementation Details

All experiments are conducted on Google Colab with an NVIDIA L4 GPU using PyTorch and the Hugging Face Transformers library with custom sliding window training loops. Model checkpoints persisted to Google Drive at each improvement in validation F1-score, and the best checkpoint is reloaded for final test evaluation, ensuring reported test numbers reflect the best model rather than the final epoch. The training, validation, and test splits use a fixed random seed of 42 and stratified sampling. DistilBERT mode collapse was confirmed by training loss remaining at $\ln(2) \approx 0.693$ throughout training under both tested learning rates — the expected loss of a random classifier on a balanced binary dataset.

B. Evaluation Protocol

Model performance is evaluated on the held-out test set (1,317 samples) using accuracy, macro-averaged F1-score, macro precision, and macro recall. Macro averaging weights for each class equally, appropriate for the balanced dataset. Statistical significance of performance differences is assessed using McNemar's test on test-set error patterns, comparing matched pairs of predictions between models.

V. RESULTS

A. Model Performance

Table I present the comprehensive performance comparison. GraphCodeBERT with stride 256 and mean aggregation achieves the best performance: 70.77% accuracy and 69.56% macro F1-score. The TF-IDF baseline achieves 46.92% accuracy, near chance on a balanced dataset, confirming that bag-of-words representations are insufficient for code semantics. DistilBERT achieved 49.96% accuracy with macro F1 of 33.32%, reflecting mode collapse: the model predicted every test sample as buggy, yielding recall 1.0 and precision 0.50 for the buggy class. Figure 2 shows the DistilBERT training curves confirming non-convergence.

TABLE I Model Performance Comparison on Test Set (1,317 Samples)

Model	**Stride**	**Agg.**	**Accuracy**	**Macro F1**	**Precision**	**Recall**	**Epochs**
TF-IDF + Logistic Regression	N/A	N/A	0.4692	0.4690	0.4692	0.4693	N/A
DistilBERT (mode collapse)	256	max	0.4996	0.3332	0.2498	0.5000	6
GraphCodeBERT (canonical)	256	max	0.7054	0.6939	0.7413	0.7052	5
GraphCodeBERT (best ablation)	**256**	**mean**	**0.7077**	**0.6956**	**0.7464**	**0.7075**	**5**

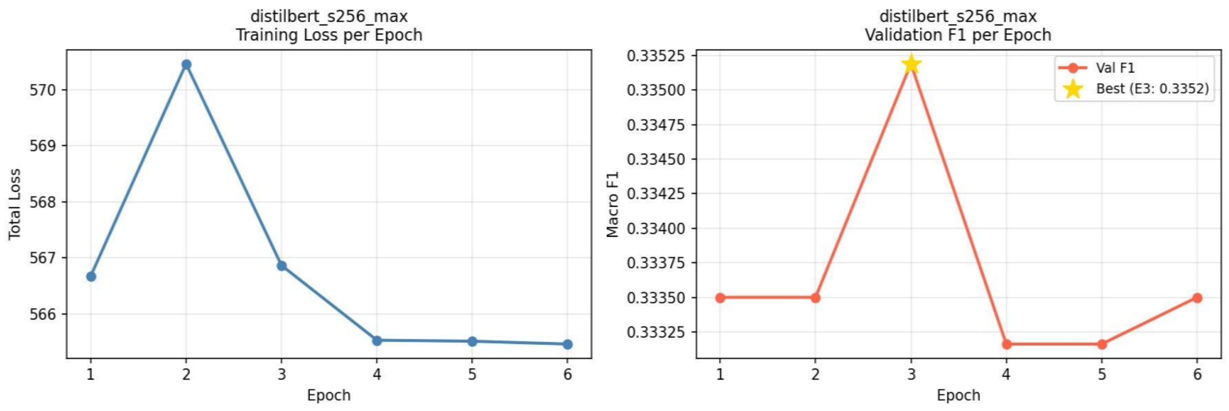


Fig. 2 DistilBERT training curves (stride 256, max aggregation). Training loss decreases marginally while validation F1 oscillates near 0.335 — the value expected from an all-one-class predictor on a balanced dataset — confirming model collapse across all six epochs.

The model collapse observed in DistilBERT is mechanistically informative. The absence of code-specific token co-occurrence patterns in the model's pretraining distribution likely prevents it from escaping the symmetric loss basin that characterizes random classification on balanced data. When the model cannot distinguish code semantics from general English, the path of least resistance is to assign all probability mass to one class, yielding the lowest achievable loss without learning discriminative features. This interpretation is consistent with the loss plateau at exactly ln (2) and with the failure persisting across both tested learning rates.

McNemar's test comparing TF-IDF against GraphCodeBERT (stride 256, max) yields $\chi^2 = 143.22$, $p = 5.26 \times 10^{-33}$ ($b = 180$, $c = 491$), confirming that GraphCodeBERT makes significantly fewer errors. The GraphCodeBERT canonical run (stride 256, max) produced notable asymmetry: TN = 592, FP = 67, FN = 321, TP = 337. Buggy-class precision is 0.834 while recall is 0.512, indicating a highly conservative decision boundary — when the model predicts buggy, it is correct 83% of the time, but it misses 49% of actual bugs. This asymmetry suggests the decision boundary is skewed toward minimizing false positives, possibly reflecting the pretraining bias toward syntactically correct code patterns in GraphCodeBERT's training corpus.

B. Sliding Window Ablation

Table II presents the full 3 × 3 ablation. Of the 1,317 test samples, 132 (10.0%) exceed 512 tokens and require windowing; all performance differences in Table II are driven entirely by these samples. Because only 10% of samples require windowing, performance differences are modest in absolute terms but consistent in direction across all nine conditions, indicating that window configuration is a relevant design choice even for a minority subset. Figures 3 through 6 show training dynamics across stride configurations, and Figures 7 and 8 visualize the ablation results.

TABLE II Sliding Window Ablation: GraphCodeBERT Performance Across All Conditions

Stride	**Aggregation**	**Accuracy**	**Macro F1**	**Precision**	**Recall**
128	Max	0.6856	0.6817	0.6952	0.6856
128	Mean	0.6872	0.6822	0.6995	0.6871
128	Majority Vote	0.6856	0.6803	0.6989	0.6856
256	Max	0.7054	0.6939	0.7413	0.7052
256	**Mean**	**0.7077**	**0.6956**	**0.7464**	**0.7075**
256	Majority Vote	0.6963	0.6816	0.7404	0.6961
384	Max	0.6879	0.6845	0.6964	0.6878
384	Mean	0.6887	0.6848	0.6984	0.6886
384	Majority Vote	0.6743	0.6683	0.6875	0.6742

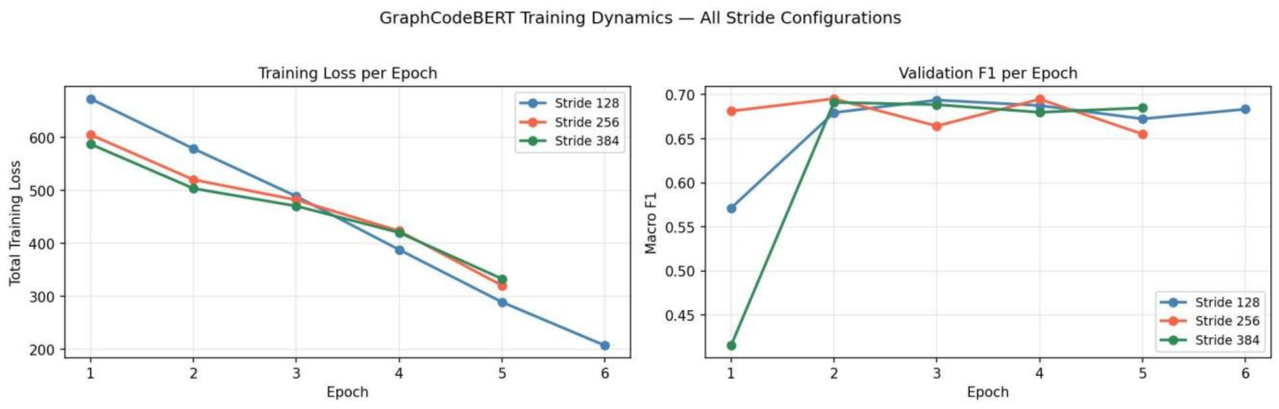


Fig. 3 GraphCodeBERT training dynamics across all stride configurations. Left: total training loss decreases steadily for all strides. Right: validation F1 converges rapidly by epoch 2 for strides 256 and 384, while stride 128 requires more epochs due to the larger training chunk count generated by its higher overlap.

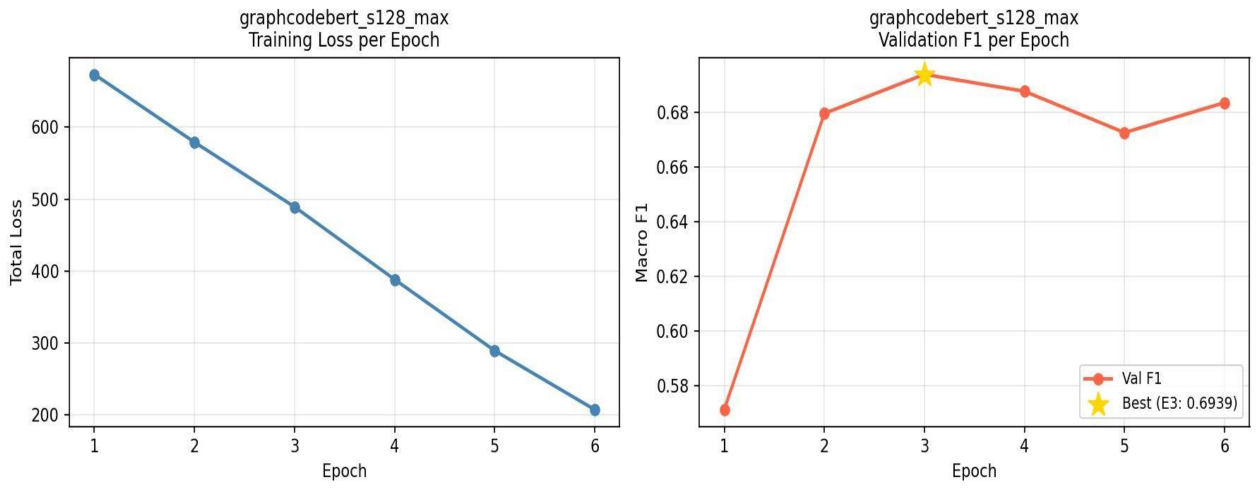


Fig. 4 Stride 128 individual training curves. Loss decreases linearly across 6 epochs; best validation F1 = 0.6939 at epoch 3. The slower convergence relative to strides 256 and 384 reflects the larger number of training chunks produced by 75% overlap.

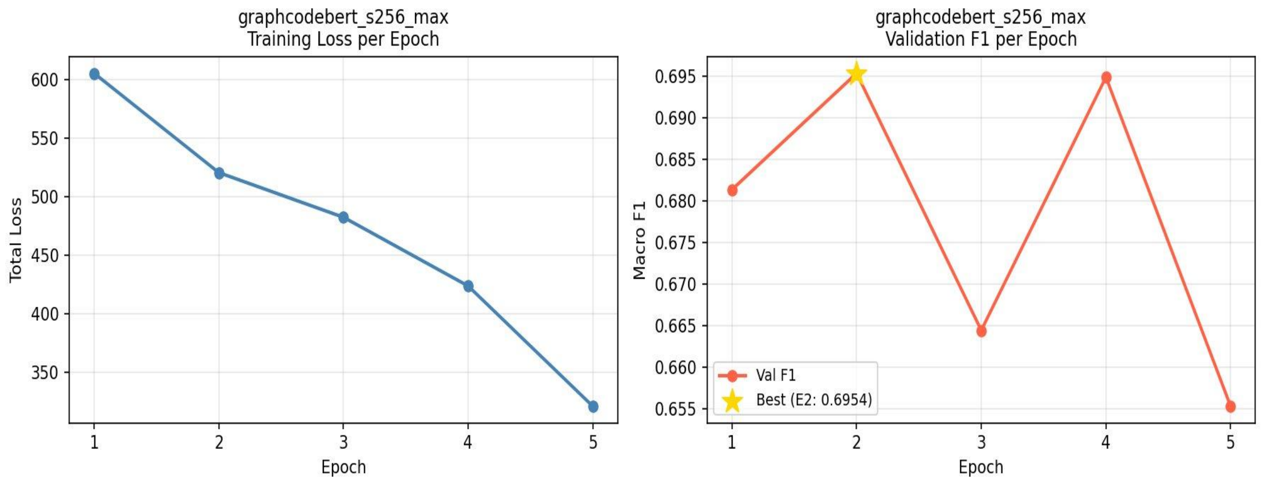


Fig. 5 Stride 256 individual training curves. Best validation F1 = 0.6954 at epoch 2; early stopping triggered at epoch 5. Loss decreases smoothly throughout training, consistent with stable gradient signals.

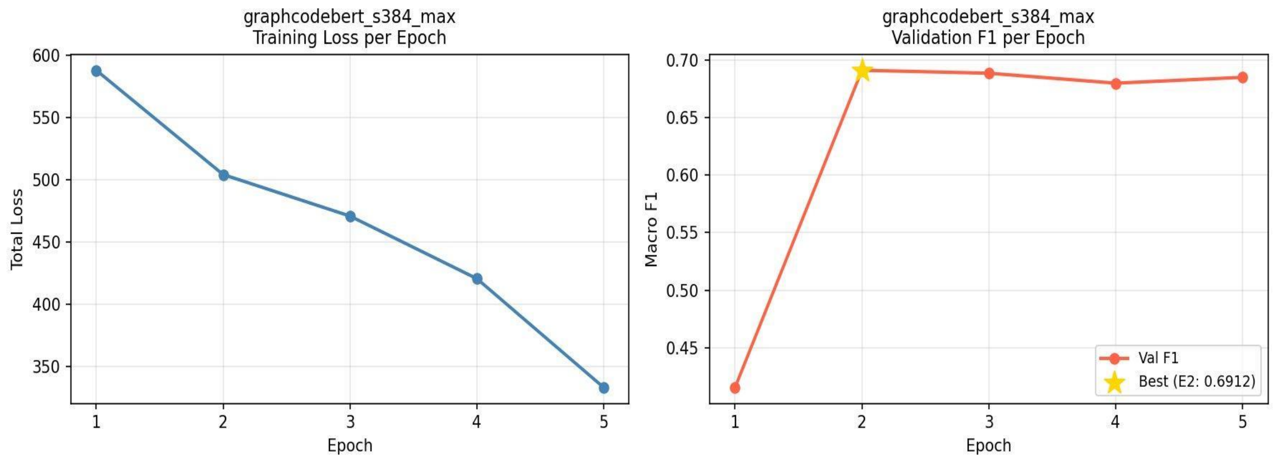
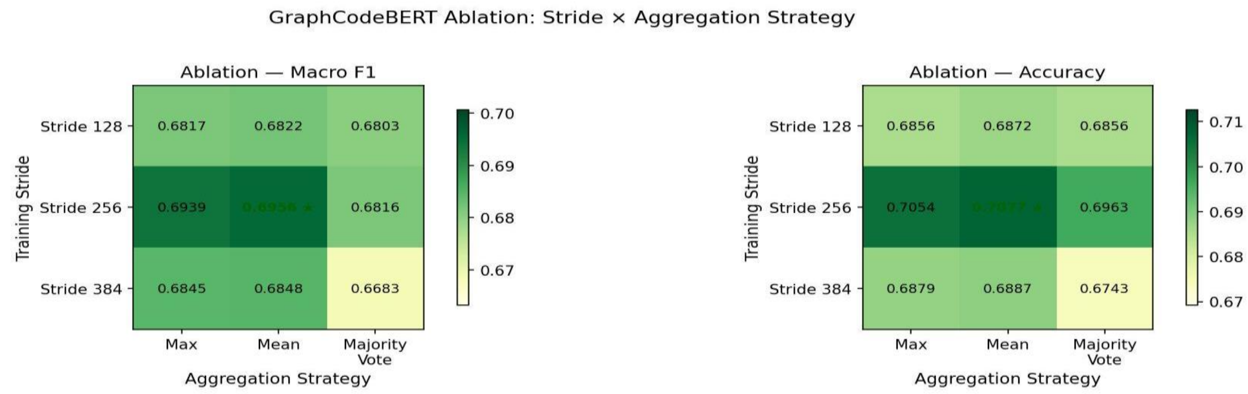


Fig. 6 Stride 384 individual training curves. Best validation F1 = 0.6912 at epoch 2; early stopping triggered at epoch 5. Rapid convergence to a slightly lower ceiling than stride 256, consistent with reduced boundary coverage.



Macro F1	Max	Mean	Majority Vote
Stride 128	0.6817	0.6822	0.6803
Stride 256	0.6939	0.6958 ★	0.6816
Stride 384	0.6845	0.6848	0.6683

Accuracy	Max	Mean	Majority Vote
Stride 128	0.6856	0.6872	0.6856
Stride 256	0.7054	0.7077 ★	0.6963
Stride 384	0.6879	0.6887	0.6743

Fig. 7 Ablation heatmap showing macro F1 (left) and accuracy (right) across all nine stride–aggregation combinations. Stride 256 with mean aggregation (starred) achieves the best performance in both metrics. Majority vote is consistently the weakest aggregation strategy, most severely at stride 384.

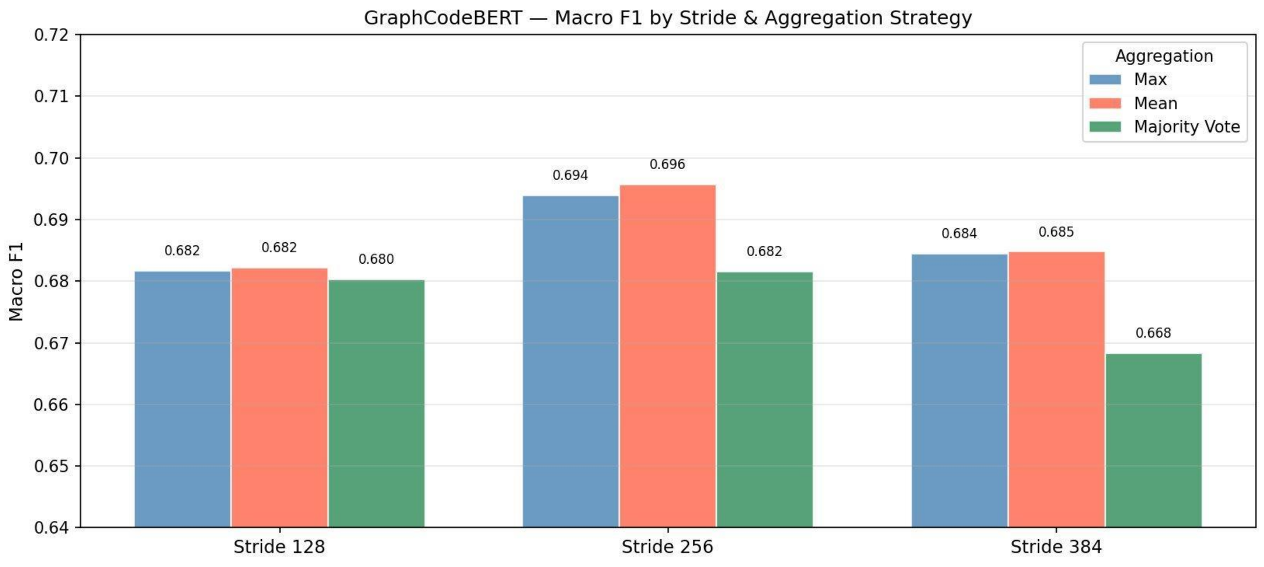


Fig. 8 Macro F1 by stride and aggregation strategy. Mean aggregation consistently outperforms max and majority vote at every stride, confirming that averaging probability scores across windows retains more information than discrete voting or single-window selection.

Stride 256 is consistently the best-performing configuration, with mean F1 0.6904 averaged across aggregation strategies, compared to 0.6814 for stride 128 and 0.6792 for stride 384. Mean aggregation outperforms max and majority vote at every stride; majority vote is consistently worst. The best overall combination is stride 256 with mean aggregation (accuracy 0.7077, macro F1 0.6956, bold row in Table II).

The underperformance of stride 128 despite greater overlap is attributable to boundary token overrepresentation: with 75% overlap, boundary tokens appear in up to four windows each, creating a training bias toward recurring boundary patterns over semantically distinctive interior content. Stride 384 underperforms because boundary regions receive less redundant coverage, increasing the risk of missing localised bugs that span window edges. Stride 256 (50% overlap) provides the optimal balance between redundancy and coverage.

C. Explainability Analysis

1) Sample Selection and Overview:

Table III summarises the ten XAI samples. The five correctly classified samples (S01–S05) all have p(buggy) ≥ 0.995, indicating very high model confidence. The five false negatives (S06–S10) have p(buggy) ≤ 0.389, well below the 0.5 decision threshold. None of the ten samples required sliding window chunking, providing clean single-window attribution maps for both methods.

TABLE III XAI Sample Summary: Top Lines by Attention Rollout and Integrated Gradients

ID	**Group**	**p(buggy)**	**Pred.**	**Attention Top Line**	**IG Top Line**
S01	Correct buggy	0.9957	✓	import sys	if(b[k][j] == 0):
S02	Correct buggy	0.9957	✓	q = int(input())	while v < r:
S03	Correct buggy	0.9956	✓	print(ans)	if counts_jobs[...] >= 1:
S04	Correct buggy	0.9951	✓	val=int(input())	b=p.count('10')+...
S05	Correct buggy	0.9956	✓	def possible(a):	if lmax == i+1 ...:
S06	False negative	0.3691	✗	for nt in range(...):	else:
S07	False negative	0.3892	✗	def main():	write(main())
S08	False negative	0.2756	✗	print(t)	t+=(n//k)
S09	False negative	0.3411	✗	import string	else:
S10	False negative	0.2873	✗	import sys	print(val)

2) Attention Rollout Analysis:

Attention rollout scores are uniformly low (line-level maxima 0.004–0.009) and broadly distributed across lines in both groups. The top-attended lines for both correctly classified and false negative samples are structurally prominent: import statements, function definitions, and loop headers. This reflects the known behaviour of attention rollout, which captures information routing through transformer layers and tends to highlight structurally central tokens regardless of their diagnostic relevance. No meaningful separation is observed between the correctly classified and false negative groups in terms of attention patterns, consistent with the view that attention weights are not reliable indicators of feature importance [11].

3) Integrated Gradients Analysis:

Integrated gradient scores reveal a clear separation between groups, summarised in Table IV. Correctly classified samples exhibit token-level maximum |IG| scores between 0.518 and 1.008 (mean 0.789), while false negatives show scores between 0.016 and 0.131 (mean 0.067). While this analysis is based on ten samples and the findings should be interpreted accordingly, the magnitude of the separation — 11.8× at the token level and 10.7× at the line level — suggests a meaningful signal rather than random fluctuation. For correctly classified samples, top IG-highlighted lines are semantically meaningful: conditional expressions (if(b[k][j] == 0):), loop conditions (while v < r:), and arithmetic computations — precisely the locations where off-by-one errors and wrong comparison operators manifest. For false negatives, IG scores are weak and mixed in sign, indicating the model found no sufficiently distinctive pattern to shift its prediction toward the buggy class.

e-ISSN 3108-026X
International Multidisciplinary Research Journal Reviews (IMRJR)
A Peer-reviewed journal
Volume 3, Issue 3, March 2026 | DOI 10.17148/IMRJR.2026.030304

TABLE IV IG Signal Strength by Classification Outcome

Metric	**Correctly Detected**	**False Negatives**
Mean max |IG| (token level)	0.7885	0.0669
Mean top-line |IG|	0.1006	0.0094
Ratio (correct / missed)	11.8× (token level)	10.7× (line level)
Mean Pearson r (attn vs IG)	−0.0097	−0.0233

This finding offers a quantitative, gradient-based explanation for false negatives that goes beyond reporting that the model failed. The weakness of the gradient signal in misclassified samples suggests these bugs are genuinely difficult to detect from static code patterns alone — consistent with the competitive programming domain, where bugs often manifest only for specific input values and may be syntactically indistinguishable from correct code.

4) Cross-Method Correlation:
The Pearson correlation between attention rollout and IG line-level scores is near-zero across all ten samples: mean token-level $r = -0.017$ (std 0.027), mean line-level $r = -0.078$ (std 0.160). No sample from either group exceeds $|r| = 0.24$ at the line level, and all p-values exceed 0.20. Furthermore, the top-scoring line identified by each method never coincides across all ten samples (0/10 agreement). This empirically supports the claim that attention weights cannot substitute for gradient-based attribution in code intelligence tasks, aligning with prior findings in NLP literature [11] and extending them to the code domain. The two methods are genuinely complementary: attention rollout reveals structural information routing while integrated gradients identify causal token sensitivity — a dual perspective that neither method provides alone.

VI. DISCUSSION

A. Implications for Practice

The DistilBERT mode collapse finding has a clear practical implication: deploying general-purpose language models for code defect detection without code-specific pretraining is ineffective regardless of learning rate tuning. The marginal cost of using GraphCodeBERT over DistilBERT is modest (499 MB versus 268 MB checkpoint), while the performance difference is the distinction between a functional and a non-functional detection system.

The high precision, low recall characteristic of GraphCodeBERT (83% precision, 51% recall on the buggy class) suggests this model is best suited as a high-confidence pre-screening tool in code review workflows, where flagged samples are subsequently reviewed by a human developer. Threshold calibration could shift the operating point toward higher recall at the cost of more false alarms, depending on deployment context.

The IG signal strength finding has practical value beyond model performance reporting. A monitoring system could compute IG signal magnitude at inference time and use it as a confidence proxy — flagging low-signal predictions for additional scrutiny. This transforms the explainability analysis from a post-hoc research tool into a deployable component of the detection pipeline.

B. Limitations

The buggy-apps dataset is limited to competitive programming solutions, which differ from production software in style, structure, and bug type. The 10.0% of test samples requiring windowing is a small subset; ablation findings should be validated on datasets with higher proportions of long files. The XAI analysis covers ten samples due to the computational cost of integrated gradients with 50 interpolation steps; while the signal-strength separation is large, larger-scale validation would strengthen the quantitative claims.

C. Future Directions

Promising directions include threshold calibration experiments to characterize the precision-recall trade-off empirically, evaluation on production code datasets to assess generalization beyond competitive programming, and automated extraction of IG-highlighted patterns across hundreds of samples to identify systematic bug signatures. Human evaluation studies could validate whether IG-highlighted lines align with developer intuitions about bug locations. Extending multi-class bug categorization would provide more actionable output, and incorporating execution trace information alongside static analysis could address the fundamental limitation of static models on execution-dependent bugs.

D. Threats to Validity

Results may be sensitive to hyperparameter choices beyond those ablated, particularly learning rate and batch size. Early stopping with patience 3 may terminate training before the optimal checkpoint for some configurations; however, validation of F1 curves shows stable convergence well before stopping in all reported runs. External validity: the dataset is drawn exclusively from competitive programming solutions, a domain with characteristic short functions and specific bug patterns (off-by-one errors, wrong comparison operators) that may not represent the full diversity of production software bugs. Generalization of other languages, codebases, or bug taxonomies requires further validation. Construct validity: the binary correct/buggy label masks variation in bug severity, detectability, and type. A sample labelled buggy that contains a subtle input-dependent error is structurally different from one with an obvious syntactic mistake, yet both receive the same training signal. This construct limitation is inherent to the dataset and affects all models evaluated.

VII. CONCLUSION

This work presents a structured investigation of transformer-based bug detection, organized around the argument that trustworthy explainability requires first establishing which model and configuration is worth explaining. Three principal findings emerge. First, domain-specific pretraining is necessary, not merely beneficial: DistilBERT failed to learn the task under any tested configuration, while GraphCodeBERT achieved 70.77% accuracy and 69.56% macro F1-score, establishing a clear boundary between models that can and cannot represent code semantics. Second, sliding window configuration meaningfully affects performance even when only 10% of samples require windowing: stride 256 with mean aggregation is optimal, outperforming alternatives by up to 2.7 percentage points in macro F1. Third, integrated gradient signal strength provides a quantitative, gradient-based explanation of model failure modes: misclassified samples exhibit 11.8× weaker IG signal than correctly classified samples, suggesting false negatives arise from genuinely ambiguous code rather than systematic model error.

The complementarity of attention rollout and integrated gradients is confirmed quantitatively: near-zero cross-method correlation and zero top-line agreement across ten samples establish that both methods are necessary for comprehensive model interpretation. Together they provide a dual-lens view of model behavior that demonstrates the value of quantitative cross-sample explainability analysis as a tool for understanding, not just reporting, transformer behavior in code intelligence tasks.

ACKNOWLEDGMENT

The authors acknowledge the use of Google Colab for computational resources and the Hugging Face community for providing pretrained models and datasets. We thank the contributors to the lrhammond/buggy-apps dataset for enabling this research.

REFERENCES

[1]. M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in Proc. 34th Int. Conf. Machine Learning (ICML), 2017, pp. 3319–3328.
[2]. Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP, 2020, pp. 1536–1547.
[3]. D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learning Representations (ICLR), 2021.
[4]. V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," arXiv:1910.01108, 2019.
[5]. A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 5998–6008.
[6]. N. Kokhlikyan et al., "Captum: A unified and generic model interpretability library for PyTorch," arXiv:2009.07896, 2020.
[7]. L. R. Hammond, "Buggy-apps: A dataset for automated bug detection," Hugging Face Datasets, 2023. [Online]. Available: https://huggingface.co/datasets/lrhammond/buggy-apps
[8]. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, 2019, pp. 4171–4186.
[9]. Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," arXiv:1907.11692, 2019.
[10]. S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 4765–4774.
[11]. S. Jain and B. C. Wallace, "Attention is not explanation," in Proc. NAACL-HLT, 2019, pp. 3543–3556.
[12]. S. Abnar and W. Zuidema, "Quantifying attention flow in transformers," in Proc. 58th Annual Meeting of the ACL, 2020, pp. 4190–4197.
[13]. M. Abdar et al., "A review of uncertainty quantification in deep learning: Techniques, applications and challenges," Information Fusion, vol. 76, pp. 243–297, 2021.
[14]. I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in Proc. Int. Conf. Learning Representations (ICLR), 2019.