Team_Omega at SemEval-2026 Task 13: Frozen vs. Trainable Representations for Out-of-Distribution AI-Generated Code Detection: A CodeBERT Fine-Tuning Study

Nahid Niyaz Shovon[1] **MD. Naim Parvez**[1]
[1]Rajshahi University of Engineering & Technology
{nahidniyaz185, naimparvez999}@gmail.com

Abstract

This paper describes Team_Omega's submission to SemEval-2026 Task 13 (Subtask A), which focuses on detecting AI-generated code under severe cross-language and cross-domain distribution shift. We investigate whether task-specific fine-tuning improves or harms out-of-distribution (OOD) generalization by conducting a controlled comparison between two CodeBERT-based configurations: a fully frozen encoder and a partially fine-tuned model with last-layer adaptation and a deeper residual classification head. While partial fine-tuning substantially improves in-domain performance to 0.9841 macro F1 score, it leads to severe degradation under OOD evaluation, collapsing to biased predictions and achieving only 0.3026 macro F1 score. In contrast, the frozen-backbone baseline achieves lower in-domain accuracy but more stable OOD performance, reaching macro F1 score of 0.5132. Our findings highlight a critical trade-off between in-domain discrimination and cross-language robustness, suggesting that preserving pretrained multilingual representations may be preferable to aggressive task adaptation in distribution-shifted code detection settings.

1 Introduction

Large Language Models (LLMs) have revolutionized software engineering by automating code generation, but their proliferation raises concerns regarding academic integrity and software security. To mitigate these risks, automated systems must accurately distinguish machine-generated from human-written code.

SemEval-2026 Task 13, "Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios" (Orel et al., 2026), directly addresses this challenge. Subtask A is a binary classification problem: provided a code snippet, determine whether it is fully human-written (label 0) or fully machine-generated (label 1), evaluated using Macro F1 as the primary metric. One of its distinguishing features is strict OOD evaluation protocol. Training data consists of algorithmic problems in seen languages (C++, Python, Java), while the hidden test set spans unseen languages (Go, PHP, C#, C, JavaScript) across unseen domains (research and production code). This cross-language and cross-domain evaluation design discourages reliance on language-specific syntax or generator-specific artifacts, encouraging models to learn more generalizable stylistic and semantic patterns that distinguish human from machine-generated code.

We propose a system that is built upon CodeBERT (Feng et al., 2020), conducting a systematic comparison of two configurations: (1) a frozen-backbone baseline that trains only a lightweight MLP classifier on [CLS] token representations, and (2) an improved partial-unfreeze model that strategically unfreezes only CodeBERT's final encoder layer (layer 11/12) using differential learning rates. Our enhanced classification head incorporates residual connections (He et al., 2016), GELU activations (Hendrycks and Gimpel, 2016), layer normalization (Ba et al., 2016), and class-weighted loss to address training data imbalance (48%/52% human/AI split). Our team ranked 44th out of 81 participating teams in Subtask A. Our code is publicly available.[1]

Our main contributions are: (1) a controlled comparison showing that a fully frozen CodeBERT backbone can outperform partial fine-tuning under severe OOD conditions; (2) analysis of architectural enhancements (residual connections, GELU, LayerNorm, class-weighted loss) demonstrating that increased parameterization improves in-domain performance but not OOD generalization; and (3) an OOD error analysis exposing

[1]https://github.com/TesNikk/SEMEVAL-2026-Task-13

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2034–2039

prediction collapse and highlighting the need for domain-invariant training strategies.

2 Related Work

2.1 AI-Generated Code Detection

Early work on AI-generated content (AIGC) detection focused primarily on natural language, encompassing zero-shot statistical methods such as DetectGPT (Mitchell et al., 2023) and supervised classifiers built on pretrained encoders (OpenAI, 2023). However, these methods transfer poorly to source code due to its strict syntactic constraints, non-natural-language identifiers, and language-specific structural patterns (Chen et al., 2021).

Shared-task benchmarks further highlight this challenge. SemEval-2024 Task 1 (Kübler et al., 2024) demonstrated strong in-domain performance for text detectors but substantial degradation under distribution shift. Large-scale benchmarks such as CoDet-M4 (Orel et al., 2025a) and DroidCollection/DroidDetect (Orel et al., 2025b) introduced multilingual and multi-generator evaluation settings, revealing persistent cross-language and cross-domain robustness gaps. (Ahmad et al., 2024) similarly showed that OOD generalization remains a critical bottleneck in code authorship attribution, directly motivating our system's partial fine-tuning design.

2.2 Pre-trained Code Models

Encoder-only models marked a breakthrough in code understanding. CodeBERT (Feng et al., 2020) demonstrated strong cross-language transfer through joint pre-training on code–natural language pairs using masked language modeling (MLM) and replaced token detection (RTD) objectives. UniXCoder (Guo et al., 2022) unified cross-modal understanding across 12 programming languages, while CodeT5 (Wang et al., 2021) introduced encoder-decoder architectures excelling in code completion and summarization. These representations capture stylistic and semantic code properties essential for authorship attribution (Ren et al., 2023), though recent benchmarks show domain-specific overfitting (Orel et al., 2025a,b). Decoder-only models such as CodeGen (Nijkamp et al., 2023) and CodeGeeX (Zheng et al., 2023) expanded the landscape of code-generating LLMs, becoming both generators of AI code and potential feature extractors for detection systems.

Our work builds upon CodeBERT as the backbone encoder, motivated by its strong multilingual code representations and compact architecture suitable for compute-constrained fine-tuning experiments, extending partial adaptation techniques validated across recent multilingual benchmarks (Orel et al., 2025a).

Model	Venue	Key Innovation	Multilingual
CodeBERT	EMNLP'20	MLM + RTD bimodal	✓ (6 langs)
UniXCoder	ACL'22	Cross-modal unification	✓ (12 langs)
CodeT5	ACL'21	Encoder-decoder	✓ (8 langs)
CodeGen	ICLR'23	Open decoder-only	✓ (multilingual)

Table 1: Comparison of pre-trained code models.

Setting	Languages	Domains
Seen–Seen	C++, Python, Java	Algorithmic
Unseen–Seen	Go, PHP, C#, C, JS	Algorithmic
Seen–Unseen	C++, Python, Java	Research, Production
Unseen–Unseen	Go, PHP, C#, C, JS	Research, Production

Table 2: The four OOD evaluation conditions defined by the task.

3 Dataset and Task Description

3.1 Task Definition

SemEval-2026 Task 13 (Orel et al., 2026), Subtask A, poses a binary classification problem: given a code snippet of source code, a system must predict whether it is fully human-written (label 0) or fully machine-generated (label 1). The macro-averaged F1-score is the primary evaluation metric. It is computed equally over both classes. The task imposes three constraints: (i) no external datasets may be used; (ii) no models pre-trained specifically for AI-generated code detection are allowed; and (iii) the `generator` metadata column—which identifies the specific AI model that produced each sample—must not be used during either training or inference. General-purpose or code-oriented pre-trained models such as CodeBERT, UniXCoder, and CodeT5 are permitted.

3.2 Out-of-Distribution Evaluation Protocol

The task's rigorous OOD evaluation protocol tests generalization across two independent axes: *programming language* and *application domain*. Training data covers seen languages (C++, Python, Java) in a seen domain (algorithmic problems), while the test set extends to unseen languages (Go, PHP, C#, C, JavaScript) and unseen domains (research, production code). The four evaluation conditions are summarized in Table 2.

Field	Description
`code`	Source code snippet (model input)
`label`	0 = human, 1 = AI (train/val only)
`language`	Programming language of the snippet
`generator`	AI model name or "human" (*forbidden*)
`ID`	Unique identifier (test set only)

Table 3: Dataset fields and their descriptions.

3.3 Dataset

The dataset is divided into three splits. The **training set** contains 500,000 samples: 238,475 human-written and 261,525 machine-generated, results in a mild class imbalance with an approximate ratio of 1:1.1 (48%/52% human/AI). The **validation set** comprises approximately 100,000 samples drawn from the same domain distribution as the training data and is used exclusively for model selection. The **test set** is a large-scale unlabeled dataset containing 500,000 samples. Additionally, a separate sanity-check file contains 1,000 labeled examples drawn from the public test set. This split is provided solely to verify evaluation formatting and is strictly excluded from model selection and hyperparameter tuning.

Each sample in the dataset contains the fields described in Table 3.

4 Methodology

We formulate Subtask A as binary classification: given a code snippet x, predict $y \in \{0, 1\}$ (human vs. AI-generated), evaluated via macro-averaged F1.

4.1 Model Architecture

CodeBERT (`microsoft/codebert-base`) (Feng et al., 2020) is used as the backbone encoder. CodeBERT follows the RoBERTa-base architecture, consisting of 12 transformer layers with a hidden dimensionality of 768 and approximately 125 million parameters. It was pre-trained on a bimodal corpus of code and natural language using masked language modeling and replaced token detection objectives, making it suitable as a general-purpose code representation model. Input code snippets are tokenized using the CodeBERT tokenizer with a maximum sequence length of 512 tokens; sequences longer than this length are truncated, and shorter sequences are padded to the maximum length.

We investigate two experimental configurations that differ in the degree of encoder adaptation.

Configuration A: Frozen Backbone Baseline. Here, all parameters of the CodeBERT encoder are frozen, and only a lightweight classification head is trained. The classification head operates on the `[CLS]` token representation and consists of three linear layers with decreasing dimensionality ($768 \to 256 \to 64 \to 2$), interleaved with ReLU activations and dropout regularization (rate 0.3). All linear layers are initialized with Xavier uniform initialization (Glorot and Bengio, 2010) and zero biases. This configuration yields approximately 214K trainable parameters and serves as a frozen reference model.

Configuration B: Partial Unfreezing with Residual Head. Here, only the final transformer layer (layer 11) of CodeBERT is unfrozen, while layers 0-10, embeddings, and the pooler remain frozen. This enables limited task adaptation while preserving lower-level pretrained representations. A differential learning rate scheme is applied, assigning a $10\times$ lower learning rate to the unfrozen encoder layer relative to the classification head to mitigate catastrophic forgetting (Kirkpatrick et al., 2017).

The classification head has a residual pre-classifier followed by a deeper projection network. The pre-classifier applies a linear transformation ($768 \to 768$), LayerNorm, GELU, and dropout (0.3), with a residual connection adding the original `[CLS]` embedding. The resulting representation is passed through successive linear layers ($768 \to 512 \to 256 \to 64 \to 2$), each followed by LayerNorm, GELU, and dropout (0.3), except for the penultimate layer where dropout is reduced to 0.15. This configuration introduces approximately 8.2M trainable parameters ($\sim$7% of total).

4.2 Optimization Strategy

Both configurations use AdamW (Loshchilov and Hutter, 2019) (weight decay 0.01) with OneCycleLR (Smith, 2018) (cosine annealing, 10% linear warmup), batch size 32, and gradient clipping (max norm 1.0).

Configuration A (frozen backbone) uses a single learning rate of 2×10^{-4} and standard cross-entropy loss.

Configuration B uses discriminative learning rates: 2×10^{-5} for the unfrozen encoder layer and 2×10^{-4} for the classification head. Class

weights are applied to weighted cross-entropy.

$$w_c = \frac{N}{C \cdot n_c}, \tag{1}$$

where N is the total number of samples, $C = 2$, and n_c is the count for class c.

Both models are trained for a single epoch of 500K samples (15,625 steps), reflecting a compute-constrained design that balances convergence and overfitting risk. No extensive hyperparameter search or ablation was conducted; the reported configurations represent manually selected settings. Validation metrics reported throughout correspond to end-of-epoch evaluation on the in-domain validation split.

5 Experimental Setup

The primary metric is macro-averaged F1: $F1_{\text{macro}} = \frac{1}{2}(F1_{\text{human}} + F1_{\text{AI}})$, assigning equal weight to both classes regardless of prevalence.

5.1 Training Protocol

Both models are trained in strict compliance with the task constraints: no external datasets are used, and the `generator` metadata column is excluded from both training and inference. The provided test sample (1,000 labeled examples) is reserved exclusively as a post-hoc formatting and sanity check, and is not used for model selection or hyperparameter tuning.

5.2 Implementation Details

Experiments are conducted on the Kaggle platform using a single NVIDIA Tesla P100 GPU. A fixed random seed (42) is set across all relevant libraries to ensure deterministic training behavior. The maximum input sequence length is 512 tokens, and uses a batch size of 32.

5.3 Experimental Comparison Setup

The two configurations form a controlled experimental setup to isolate the impact of encoder adaptation. Table 4 summarizes the key differences between the two configurations.

Aspect	Config. A	Config. B
Encoder	Fully frozen	Last layer trainable
Trainable params	∼214K	∼8.2M
Head depth	3 Linear	4 Linear + pre-clf
Activation	ReLU	GELU
Normalization	None	LayerNorm
Residual conn.	No	Yes
Class weighting	No	Yes (balanced)
Differential LR	No	Yes
Loss	CrossEntropy	Weighted CE

Table 4: Comparison of the two experimental configurations.

6 Results

6.1 In-Domain Performance

Table 5 summarizes the training and validation results for both configurations.

Configuration	Train Loss	Train F1	Val Loss	Val F1
A (Frozen)	0.2800	0.8862	0.1853	0.9216
B (Partial Unfreeze)	0.1007	0.9640	0.0569	0.9841

Table 5: In-domain training and validation results. Both models are trained for one epoch on the same data.

Configuration B outperforms Configuration A across all in-domain metrics, with validation macro F1 improving from 0.9216 to 0.9841 and notably lower validation loss (0.0569 vs. 0.1853). This confirms that partial encoder adaptation with differential learning rates enables more discriminative representations than the frozen baseline.

6.2 OOD Generalization

Table 6 reports the macro F1 scores from our official submissions to the SemEval-2026 Task 13 leaderboard, evaluated on the hidden test set spanning unseen languages and domains. The generalization gap between validation and leaderboard performance is the primary quantity of interest.

Both configurations suffer substantial OOD drops, but the severity differs markedly. Configuration A drops from 0.9216 to 0.5132, whereas Configuration B drops from 0.9841 to 0.3026. Counterintuitively, the stronger in-domain model generalizes worse. This suggests that Configuration B's larger parameter budget (∼8.2M vs. ∼214K) overfits to language-specific syntax and algorithmic patterns that do not transfer to unseen languages and domains. However, we note that this effect may also be partially influenced by limited training duration: with only a single epoch over 500K samples, the partially fine-tuned model may not have reached stable representations. Additional training (e.g., 2–3 epochs) could potentially alter this behavior, although this was not explored due to compute constraints.

Configuration	Val F1	Test F1
A (Frozen)	0.9216	0.5132
B (Partial Unfreeze)	0.9841	0.3026

Table 6: Official leaderboard results (OOD evaluation on the test set).

Config.	Class	Precision	Recall	F1
A (Frozen)	Human	0.8565	0.4916	0.6247
	AI	0.2870	0.7130	0.4093
B (Partial Unfreeze)	Human	0.9615	0.1287	0.2270
	AI	0.2444	0.9821	0.3914

Table 7: Per-class performance on the 1,000 labeled test sample.

6.3 Error Analysis

We evaluate both configurations on the 1,000-example labeled public test sample (777 human, 223 AI) to analyze model behavior. This split is separate from the hidden leaderboard test set; therefore, the reported per-class scores are diagnostic and do not reflect official leaderboard results. Table 7 presents precision, recall, and F1 per class.

The results reveal distinct failure modes across configurations (Table 7). Configuration A shows balanced but mediocre performance, whereas Configuration B exhibits *prediction collapse*: it adopts a degenerate prediction strategy, classifying the vast majority of samples as "AI-generated" (AI recall $= 0.9821$, human recall $= 0.1287$). This near-constant majority-class prediction under OOD conditions indicates that partial fine-tuning has overridden generalizable representations with distribution-specific decision boundaries. Notably, the diagnostic sample contains 777 human vs. 223 AI examples ($\approx$3.5:1 imbalance), which may inflate AI-class precision and further amplify the apparent severity of Configuration B's majority-class bias.

6.4 Discussion

The two configurations highlight a clear trade-off between in-domain accuracy and OOD robustness.

Frozen representations are more domain-agnostic. Configuration A's frozen encoder was never adapted to the training distribution, paradoxically making it more robust though still inadequate on OOD data. The pre-trained features capture general code properties (lexical patterns, structural regularities) that transfer modestly across languages.

Fine-tuning amplifies distribution-specific features. Configuration B achieves substantially higher in-domain performance but overfits to patterns specific to the seen languages (C++, Python, Java) and the algorithmic domain. Under language and domain shift, these learned features fail to generalize, leading to severe performance degradation and biased predictions. Additionally, Configuration B uses class-weighted loss while Configuration A does not, introducing a confound that may independently affect the observed prediction bias under distribution shift.

Inference efficiency. Configuration A is also more efficient at inference, with $\sim$214K vs. $\sim$8.2M trainable parameters, yielding faster forward passes.

Implications for future work. These results suggest that standard fine-tuning is insufficient for robust cross-language generalization. Explicit domain-invariant strategies such as contrastive learning, which pulls same-class representations together while pushing apart different-class embeddings, or adversarial domain adaptation, which trains an auxiliary discriminator to make learned features domain-agnostic may be necessary to achieve stable OOD performance.

Limitations

The 512-token input limit truncates longer programs, potentially removing structural information. Training was limited to a single epoch due to the Kaggle platform's 12-hour GPU runtime constraint; with 500K training samples and $\sim$8.2M trainable parameters, Configuration B may be undertrained rather than solely overfitting, and additional epochs (e.g., 2–3) could potentially alter the observed generalization dynamics. Neither configuration explicitly models domain invariance, relying instead on pre-trained representations for transfer. No extensive ablation studies over architectural choices or loss configurations were performed. Finally, inference uses simple argmax prediction without ensembling or calibration, leaving room for improvement.

References

Wasi U Ahmad, Snigdha Chakraborty, and 1 others. 2024. Out-of-distribution generalization for code authorship attribution. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. Association for Computational Linguistics.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Mark Chen, Jerry Tworek, Heewoon Jun, Qiming Yuan, Henrique Peneder Pinto, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*.

Daya Guo, Shuo Ren, Shuai Lu, Shengdong Li, Duyu Feng, Dazhao Liu, Ming Li, Ming Zhou, Nan Duan, and Alex Zhou. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, and 1 others. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*.

Clara Kübler, Pasquale Minervini, and Sebastian Riedel. 2024. Semeval-2024 task 1: Track A (human vs. AI text detection). In *Proceedings of the 18th International Workshop on Semantic Evaluation (SemEval-2024)*. Association for Computational Linguistics.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Eric Mitchell, Jingfeng Peng, Timo Dettmers, Daniel Krasnopolsky, Elaine A. Chi, Christopher Potts, and Eric Horvitz. 2023. DetectGPT: Zero-shot machine-generated text detection using probability curvature. *Transactions on Machine Learning Research*.

Erik Nijkamp, Bo Li, Mark Rabe, and 1 others. 2023. Codegen: Open large language models for code. In *International Conference on Learning Representations*.

OpenAI. 2023. OpenAI classifier. `https://openai.com/blog/new-ai-classifier-for-indicating-ai-written-text`. Accessed: 2026-03-01.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025a. CoDet-M4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026. SemEval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. In *Proceedings of the 20th International Workshop on Semantic Evaluation (SemEval-2026)*, San Diego, USA. Association for Computational Linguistics.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025b. `Droid`: A resource suite for AI-generated code detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31263–31289.

Shuo Ren, Daya Guo, Shuai Lu, Shengdong Li, Duyu Feng, Dazhao Liu, Ming Li, Ming Zhou, Nan Duan, and Alex Zhou. 2023. Cross-lingual code representation learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

Leslie N. Smith. 2018. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*.

Yue Wang, Weishi Li, Marek Mishra, Shomir Goldwasser, Ding Wang, and Fei Xia. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *arXiv preprint arXiv:2303.17568*.