

Catching AI-Generated Code at Scale: A Lightweight and Interpretable Detector for the LLM Era

Arnesh Batra¹ Krish Thukral² Dev Sharma^{1,*}

Aniket Khandelwal^{1,*} Arush Gumber^{1,*}

¹Indraprastha Institute of Information Technology, Delhi

²Manipal University Jaipur

arnesh23129@iiitd.ac.in, KRISH.23FE10CSE00679@mu.j.manipal.edu,
dev23189@iiitd.ac.in, aniket23087@iiitd.ac.in, arush23136@iiitd.ac.in

Abstract

As LLMs become everyday creative partners in software development, code is no longer a purely human artefact; it emerges from an increasingly hybrid human-AI process. Understanding who contributed which parts of a codebase has become a foundational question of creative attribution, touching on authorship, trust, and the integrity of co-created work. Yet existing detectors exceed 100M+ parameters, are too slow for practical deployment, and offer *no insight* into what distinguishes human from AI creative contributions. We present **NanoDroid**, a backbone-free detector with only **5.1M parameters** that runs **20×** faster than 125M-parameter UniXcoder (SOTA) and is *fully interpretable*: per-stream attribution scores explicitly expose which stylistic evidence (token statistics, transition patterns, surface style) distinguishes human from AI authorship in code. On the challenging 4-class DroidCollection benchmark (Human / Machine / Refined / Adversarial), NanoDroid achieves **90.77%** weighted F1, within 2.6% of SOTA, on the CodeT-M4 benchmark, NanoDroid achieves SOTA performance, while providing the kind of transparent decision decomposition that black-box models cannot. Beyond a detection tool, we argue NanoDroid offers a lens for studying how human and AI creativity differ stylistically, and where they blur.

1 Introduction

AI coding tools (GitHub Copilot, Cursor, ChatGPT) now generate and refactor code at scale, making provenance a practical engineering problem. Three deployment contexts make detection urgent: academic platforms must flag AI-generated submissions (Salim et al., 2024); security pipelines must audit code for silently introduced backdoors and injection flaws (Orel et al., 2025b); and competitive programming judges must enforce human-only

rules. Detection is hard because adversarial humanization actively mimics human style to evade filters, and source code’s rigid syntax compresses the statistical signals that work for natural language (Yang et al., 2023).

Existing SOTA detectors (DroidDetect, UniXcoder) require 125–149M parameters, consume significant inference budget at submission scale, and are opaque black boxes that cannot explain *why* a sample was flagged, a critical gap for any auditable deployment. We present **NanoDroid**: a backbone-free, **5.1M parameter** detector that is **21×** smaller than SOTA, runs **20×** faster, and is *intrinsically interpretable* via per-stream attribution scores. The core insight is that detection is a *statistical and structural* task; one need not understand what code does, only how it was generated, allowing us to replace the pretrained encoder with lightweight $O(L)$ primitives. Our contributions:

- **NanoDroid**: a 5.1M parameter backbone-free code origin detector achieving within 2.6% of 125M-parameter SOTA on DroidCollection across 2-, 3-, and 4-class settings.
- Three primary $O(L)$ encoding primitives: AI-Probability-Weighted Pooling (APWP), Bi-gram Transition Fingerprint (BTF), and Fibonacci TextCNN (FTC), each capturing a distinct generation signature.
- Full interpretability (Appendix D) via per-stream attribution scores and token-level AI-ness weights enabling human inspection of every prediction.
- Comprehensive ablation studies on DroidCollection, quantifying the contribution of each stream and novel component.

2 Related Work

We cover three relevant areas: **1)** AI-generated text detection, **2)** AI-generated code detection, and **3)** adversarial evasion.

* These authors contributed equally to this work.

† <https://github.com/arnesh2212/AI-Code-Det-NLP>

2.1 AI-Generated Text Detection

Detection methods fall into two paradigms. Zero-shot approaches exploit statistical signatures without labeled data: DetectGPT (Mitchell et al., 2023) leverages log-probability curvature, and Fast-DetectGPT (Bao et al., 2024) achieves similar accuracy far more efficiently via conditional probability sampling. Supervised classifiers train on paired examples and yield stronger in-domain performance, though they generalise poorly across domains (Wang et al., 2023). Large-scale benchmarks include the multilingual M4 (Wang et al., 2024), MULTITuDE (Macko et al., 2023), MAGE (Li et al., 2024), RAID (Dugan et al., 2024), and shared tasks at RuATD (Shamardina et al., 2022), COLING (Wang et al., 2025), ALTA (Mollá et al., 2024), and DagPap (Chamezopoulos et al., 2024).

2.2 AI-Generated Code Detection

Code is harder to detect than natural language: strict syntactic constraints compress token distributions (Yang et al., 2023), and generic detectors like GPTZero degrade badly on code (Pan et al., 2024; JianWang et al., 2024). Shallow feature-engineering (Idialu et al., 2024; Li et al., 2023) showed that code statistics (AST depth, line count) can indicate authorship, but deep learning proved more capable (Tulchinskii et al., 2023). GPT-Sniffer (Nguyen et al., 2024) applies multi-class classification over CodeSearchNet (Husain et al., 2019) paired with ChatGPT outputs, while GPT-Sensor (Xu et al., 2025) uses contrastive training for more separable embeddings. CodeTM4 (Orel et al., 2025a), CodeMirage (Guo et al., 2025), and AICD Bench (Orel et al., 2026) provide further benchmarks. The most comprehensive resource is DroidCollection (Orel et al., 2025b), spanning over one million samples across seven languages, 43 coding models, and three real-world domains, including adversarially humanised code. DroidDetect, the detector trained on this dataset, represents the current SOTA. We use DroidCollection as our primary benchmark.

2.3 Adversarial Evasion

Supervised detectors are brittle against humanised outputs (He et al., 2024; Masrour et al., 2025). RAID (Dugan et al., 2024) studies paraphrasing and synonym substitution attacks. We extend this adversarial focus to code, training NanoDroid on DroidCollection’s adversarially humanised class to build explicit robustness.

3 NanoDroid

NanoDroid treats AI code detection as a *statistical and structural* task, not a semantic one. This allows us to replace a 125M parameter pretrained encoder with multiple lightweight $O(L)$ streams that each capture a distinct generation signature, fused via stream-level self-attention (Vaswani et al., 2017). Refer to Fig 1 for the complete model architecture.

3.1 Shared Embeddings

All token-based streams share a compact embedding layer: word embeddings $\mathbf{E} \in \mathbb{R}^{V \times 64}$ and positional embeddings $\mathbf{P} \in \mathbb{R}^{514 \times 64}$. The vocabulary ($V = 50,267$) matches the RoBERTa (Liu et al., 2019) tokenizer, enabling direct comparison with backbone-based models.

3.2 AI-Probability-Weighted Pooling (APWP)

Standard mean-pooling treats all tokens equally, and prior work on learned weighted pooling assigns attention weights conditioned on the sequence representation itself (Lin et al., 2017b). APWP instead assigns each vocabulary token t a learned scalar weight w_t that is independent of context, capturing a token’s intrinsic “AI-ness” across the corpus rather than its salience within a single sequence:

$$\text{APWP}(\mathbf{h}) = \frac{\sum_i \sigma(w_{t_i}) \mathbf{h}_i}{\sum_i \sigma(w_{t_i})} \quad (1)$$

where $\mathbf{h}_i$ is the output of a lightweight 3-layer transformer at position i . The model learns that tokens like result, helper, function, and structured docstring markers, overwhelmingly common in LLM outputs should have higher discriminative weight. APWP requires V additional parameters and $O(L)$ forward pass cost, versus $O(L^2 \cdot D \cdot 12)$ for a full transformer.

3.3 Bigram Transition Fingerprint (BTF)

LLM-generated code exhibits predictable token transitions: the model selects “expected” continuations, yielding smooth Markov chains with low burstiness, while human code carries idiosyncratic, irregular transitions whose count distributions are heavier-tailed than a uniform multinomial would predict (Mazarura et al., 2020). BTF exploits this asymmetry by:

- Hash each adjacent pair (t_i, t_{i+1}) to one of $H = 8,192$ buckets: $b = (31t_i + t_{i+1}) \bmod H$, following the feature-hashing trick of Weinberger et al. (2009). This lets BTF cover a $V^2 \approx 2.5 \times 10^9$ bigram space with a fixed H -row embedding table.

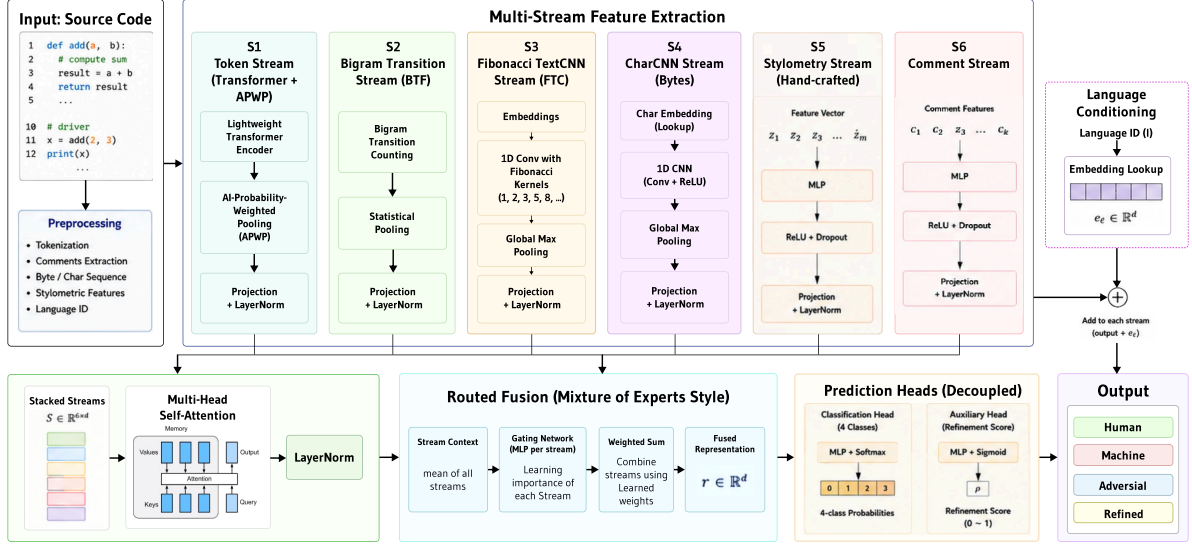


Figure 1: NanoDroid architecture. Six $O(L)$ streams capture complementary generation signatures and are fused via learned stream-level self-attention. Total parameters: 5.1M, vs. ≥ 125 M for backbone-based SOTA.

- Look up each bucket in a learned D -dimensional embedding table.
- Compute the mean and *variance* of bucket vectors over the sequence. Variance captures transition uniformity, low for predictable LLM code and high for irregular human code.
- Project $[\text{mean} || \text{var}]$ to STREAM_DIM via an MLP.

BTF is fully differentiable, $O(L)$, and length-invariant.

3.4 Fibonacci TextCNN (FTC)

We analyze token sequences at multiple scales using CNN kernels of size $[2, 3, 5, 8, 13]$, extending the multi-kernel TextCNN architecture of Kim (2014) with a Fibonacci-spaced receptive-field schedule. The ratio $\phi \approx 1.618$ provides supra-linear scale coverage that matches the hierarchical depth of code structure: token pairs (2), triples (3), short clauses (5), statements (8), and sub-function fragments (13). Standard TextCNN typically uses arithmetically spaced kernels (e.g., $[3, 4, 5]$), which over-sample short spans and under-sample statement-level structure; Fibonacci spacing allocates one filter to each natural granularity in code without redundancy. Global max-pooling per kernel (Collobert et al., 2011) yields position-invariant syntactic pattern detection. FTC is an $O(L)$ operation with ≈ 300 K parameters.

3.5 Additional Streams

CharCNN encodes raw UTF-8 bytes via three strided Conv1d layers, capturing indentation patterns, naming conventions, and operator spacing orthogonal to token semantics (≈ 225 K parameters). **Stylometry MLP** projects 14 hand-crafted lexical and structural statistics (line regularity, bigram repetition, function density, nesting depth, operator density, lexical diversity, comment ratio) to a common stream representation. Comment Stream uses an MLP over hand-crafted comment statistics, including comment-line ratio, inline-comment ratio, docstring density, average comment length, comment/code token ratio, and lexical overlap, providing complementary stylistic signals beyond code structure and syntax.

3.6 Language-Stream Fusion

Each stream is enriched with a per-programming-language embedding (32-dim, 21 languages) projected to stream dimension. The six streams are stacked as a sequence and passed through a Stream-Level Self-Attention (SLSA) module (More detailed in the Appendix), allowing each stream to attend to all others. NanoDroid (our full model) additionally includes a comment stream and a router that learns per-sample stream weights, plus a disagreement probe that explicitly models mixed-authorship signals.

3.7 Training Objectives

NanoDroid is trained with a composite loss:

$$\begin{aligned}\mathcal{L} = & \mathcal{L}_{\text{focal}} + \lambda_{\text{sc}}\mathcal{L}_{\text{XL-SupCon}} \\ & + 0.1\mathcal{L}_{\text{aux}} + 0.05\mathcal{L}_{\text{R-Drop}} \\ & + \lambda_u\mathcal{L}_{\text{uniform}} + \lambda_m\mathcal{L}_{\text{mix}}\end{aligned}\quad (2)$$

Focal Loss (Lin et al., 2017a) with label smoothing ($\gamma = 2$, $\epsilon = 0.1$) focuses gradient on hard Machine-Refined boundary examples. **XL-SupCon** (Khosla et al., 2020) pulls same-class embeddings together across different languages, directly improving cross-lingual generalization. **R-Drop** (Wu et al., 2021) regularizes by minimizing KL divergence between two forward passes with different dropout masks. A **language-level prototype bank** (EMA centroids per class per language) provides per-language decision boundaries at inference, directly targeting cross-language failure modes. **Fast Gradient Method** (Miyato et al., 2016) (FGM, $\epsilon = 0.25$) adversarial training augments the token embedding layer to build robustness to adversarial code samples.

Table 1: Loss ablation on the 3-class dev set (5 epochs each). Full config: $\lambda_{\text{sc}}=0.10$, $\lambda_u=0.02$, $\lambda_m=0.08$.

Configuration	λ_{sc}	λ_u	λ_m	Wt. F1
Full NanoDroid	0.10	0.02	0.08	0.881
w/o XL-SupCon	—	0.02	0.08	0.863
w/o Uniformity	0.10	—	0.08	0.874
w/o Mix Loss	0.10	0.02	—	0.869
w/o R-Drop	0.10	0.02	0.08	0.872
w/o FGM adv. training	0.10	0.02	0.08	0.866
Focal only (baseline)	—	—	—	0.847

4 Experiments and Results

4.1 Baselines

We compare against the following models, using their **original hyperparameters and results** as reported in the respective papers:

Zero-shot baselines (no fine-tuning). **Fast-DetectGPT** (Bao et al., 2024) performs conditional probability curvature analysis on LLM token likelihoods. **GPTZero** is a commercial zero-shot detector. **CoDet-M4** (Orel et al., 2025a) zero-shot uses a ModernBERT-based (Warner et al., 2024) architecture.

Fully Trained Baselines The **Full Training** block reports all detectors trained end-to-end on DroidCollection using AdamW, learning rate 5×10^{-5} , 3 epochs, linear warmup (ratio 0.1) with cosine decay, and batch size 64. **M4_{FT}** (Wang et al.,

2024) uses a RoBERTa-base (Liu et al., 2019) backbone. **GPT-Sniffer_{FT}** (Nguyen et al., 2024) uses CodeBERT (Feng et al., 2020) ($\approx 125\text{M}$). **CoDet-M4_{FT}** (Orel et al., 2025a) uses ModernBERT-base. **UniXcoder** (Guo et al., 2022) (125M) is a cross-modal code encoder we train under the same protocol. **DroidDetect_{CLS-Base}** (Orel et al., 2025b) (ModernBERT-base, 149M) is one of the SOTA models. **NanoDroid** (ours, 5.1M) matches competitive baselines at $\sim 20\times$ fewer parameters.

4.2 NanoDroid Hyperparameters

NanoDroid is trained for 15 epochs with AdamW ($\text{lr} = 5 \times 10^{-5}$, weight decay = 0.01), batch sizes 64 / 128 (3-class / 4-class), linear warmup (ratio 0.1), gradient clipping 1.0, projection dim 128, max sequence length 512. Loss weights: $\lambda_{\text{sc}} = 0.1$, prototype weight = 0.3, $\lambda_u = 0.02$, $\lambda_m = 0.08$, router temperature = 0.85, FGM $\epsilon = 0.25$. All experiments use seed 42.

4.3 Results

Table 2 compares NanoDroid against all baselines across classification granularities on DroidCollection.

Zero-shot and fine-tuned baselines. Zero-shot methods perform poorly on code detection. Fast-DetectGPT achieves the best zero-shot performance with 76.58% binary weighted F1, but drops to 70.98% in the 3-class setting, indicating limited transfer of probability-curvature signals to code. M4, GPTSniffer, and CoDet-M4 remain below 55% across both settings.

Fine-tuning improves performance substantially. CatBoost reaches 90.02% binary weighted F1, providing a strong non-neural baseline, while GCN ($\sim 2\text{M}$ parameters) achieves 77.32%, indicating limited capacity for modeling code semantics.

Fully trained baselines. Fully trained transformer-based models significantly outperform zero-shot and shallow methods. CoDet-M4_{FT} achieves the best binary performance at 99.08%, while UniXcoder achieves the best 3-class and 4-class results at 94.18% and 93.35%, respectively.

Our reproduced DroidDetect_{CLS-Base} achieves 98.02% binary, 93.81% 3-class, and 93.16% 4-class weighted F1, confirming the strength of large pretrained encoders in this setting.

NanoDroid. Despite having only 5.1M parameters, NanoDroid achieves competitive performance across all settings. It reaches 97.22% binary weighted F1, matching GPT-Sniffer_{FT} while using approximately $25\times$ fewer parameters.

Table 2: Weighted F1 (%) on DroidCollection across classification settings. **Bold**: best per column overall. Underline: best within ≤ 10 M parameters.

Group	Model	Params	2-class	3-class	4-class	Interp.
<i>Zero-Shot</i>	Fast-DetectGPT (Bao et al., 2024)	—	76.58	70.98	—	✓
	M4 (Wang et al., 2024)	125M	52.81	57.92	—	✗
	GPTSniffer (Nguyen et al., 2024)	125M	52.40	49.96	—	✗
	CoDet-M4 (Orel et al., 2025a)	149M	47.97	41.28	—	✗
	GPTZero	—	54.81	51.48	—	✗
<i>Fine-Tuned</i>	GCN (Orel et al., 2025b)	~2M	77.32	59.88	—	✗
	CatBoost (Prokhorenkova et al., 2018)	—	90.02	78.83	—	✗
<i>Full Training</i>	M4 _{FT} (Wang et al., 2024)	125M	91.63	76.57	—	✗
	GPT-Sniffer _{FT} (Nguyen et al., 2024)	125M	97.22	85.16	—	✗
	CoDet-M4 _{FT} (Orel et al., 2025a)	149M	99.08	89.60	—	✗
	DroidDetect _{CLS-Base} (Orel et al., 2025b)	149M	98.02	93.81	93.16	✗
	UniXcoder (Guo et al., 2022)	125M	98.11	94.18	93.35	✗
<i>Ours</i>	NanoDroid	5.1M	<u>97.22</u>	<u>91.52</u>	<u>90.77</u>	✓

Table 3: Per-class precision/recall/F1 on the 4-class DroidCollection test set. **Bold**: best metric across all three models. NanoDroid achieves the highest Refined recall and matches DroidDetect_{CLS-Base} on Adversarial recall at $29\times$ lower parameter cost.

Model	Params	Human ($n=49,932$)		Machine ($n=26,524$)		Refined ($n=13,230$)		Adversarial ($n=16,140$)		Wt. F1
		P/R	F1	P/R	F1	P/R	F1	P/R	F1	
DroidDetect _{CLS-Base} (Orel et al., 2025b)	149M	.9761 / .9858	.9809	.9001 / .8967	.8984	.8263 / .8164	.8213	.9312 / .9173	.9242	93.16
UniXcoder (Guo et al., 2022)	125M	.9800 / .9796	.9798	.9020 / .9042	.9031	.8309 / .8285	.8297	.9253 / .9252	.9252	93.35
NanoDroid (ours)	5.1M	.9747 / .9633	.9690	.9096 / .8203	.8627	.7250 / .8505	<u>.7828</u>	.8708 / .9196	<u>.8946</u>	<u>90.77</u>

Table 4: Recall on human-written and adversarial code in the binary (2-class) setting. Data for published baselines from Table 5; NanoDroid computed on the full DroidCollection test split. **Bold**: best per row (excluding the removed Large variant).

Model	Params	Human R.	Adv. R.
Fast-DetectGPT	—	0.84	0.48
M4	125M	0.40	0.73
CoDet-M4	149M	0.38	0.63
GPTZero	—	0.53	0.10
GPTSniffer	125M	0.65	0.49
M4 _{FT}	125M	0.91	0.67
GPT-Sniffer _{FT}	125M	0.97	0.55
CoDet-M4 _{FT}	149M	0.96	0.51
DroidDetect _{CLS-Base}	149M	0.93	0.92
NanoDroid (ours)	5.1M	0.96	0.92

On the 3-class task, NanoDroid achieves 91.52%, outperforming GPT-Sniffer_{FT} (85.16%) and CoDet-M4_{FT} (89.60%). On the most challenging 4-class setting, it reaches 90.77%, remaining within 2.6 points of UniXcoder (93.35%) and 2.4 points of DroidDetect_{CLS-Base} (93.16%).

NanoDroid is also the only model providing token-level interpretability, enabling direct inspec-

tion of decision signals without additional explanation methods.

4.4 Inference Efficiency

To substantiate the efficiency claims motivating NanoDroid’s design, we benchmark inference latency and memory usage against UniXcoder, the strongest-performing baseline in Table 2, under matched conditions: same hardware (NVIDIA RTX A4500), same software stack (PyTorch 2.11, CUDA 13.0), and identical input sequence length (512 tokens). NanoDroid has 5.09M parameters versus UniXcoder’s 125.93M, a $24.7\times$ reduction. Table 5 reports wall-clock inference speedup and peak GPU memory across batch sizes. NanoDroid’s speedup grows with batch size, from $7.9\times$ at batch size 1 to $37.2\times$ at batch size 32, reflecting the lower per-sample compute cost of its $O(L)$ streams relative to UniXcoder’s quadratic self-attention. Memory savings follow the same trend: at batch size 8, NanoDroid uses 118 MB versus UniXcoder’s 622 MB ($5.3\times$ less), and at batch size 32, 329 MB versus 1018 MB ($3.1\times$ less). These gains directly support the deployment scenario motivating this work: a submission-scale

Batch size	Speedup	NanoDroid mem.	UniXcoder mem.
1	7.9×	42 MB	187 MB
8	26.9×	118 MB	622 MB
32	37.2×	329 MB	1018 MB

Table 5: Inference speedup and peak GPU memory of NanoDroid vs. UniXcoder at sequence length 512, measured on an NVIDIA RTX A4500 (PyTorch 2.11, CUDA 13.0). NanoDroid has 5.09M parameters vs. UniXcoder’s 125.93M (24.7× smaller).

platform scoring code inline can batch requests and realize substantially larger throughput gains than a single-sample estimate would suggest.

4.5 Per-Class Analysis

Table 3 reports per-class precision, recall, and F1 for the 4-class setting; the confusion matrix is shown in Fig. 6.

Refined class. NanoDroid achieves the highest Refined recall at **0.8505**, exceeding UniXcoder (0.8285) and DroidDetect_{CLS-Base} (0.8164). This advantage is consistent with the overall 3-class improvement observed in Table 2. The gain is driven by the FTC stream, which captures over-uniformity in token-frequency distributions introduced during LLM refinement. Larger transformer models appear to distribute this signal across attention layers, while NanoDroid isolates it explicitly through a dedicated component. Also note that Refined samples form the most ambiguous decision boundary for NanoDroid (Fig. 6), while Human samples are consistently identified with high reliability.

Adversarial class. NanoDroid achieves an Adversarial recall of 0.9196, slightly exceeding DroidDetect_{CLS-Base} (0.9173) and approaching UniXcoder (0.9252). This is notable given the 29× smaller parameter count.

This performance is supported by FGM adversarial training and the BTF burstiness stream, which captures residual statistical artifacts that persist after human-style edits.

4.6 Adversarial Robustness

Table 4 evaluates recall on human-written and adversarial samples in the binary setting. A consistent issue in prior work is the trade-off between human and adversarial recall. Models optimized for adversarial detection often degrade on human samples, as seen in GPT-Sniffer_{FT} (0.97 / 0.55) and CoDet-M4_{FT} (0.96 / 0.51).

DroidDetect_{CLS-Base} balances both objectives, achieving 0.93 / 0.92. NanoDroid matches this

adversarial recall at 0.92 while maintaining strong human recall at 0.96. This demonstrates that robust adversarial detection can be achieved without sacrificing performance on human-written code, even at significantly lower model scale.

4.7 Ablation Study

Table 6 reports leave-one-out stream ablation on the 3-class test set. Each row removes one stream and replaces it with a zero vector.

Table 6: Leave-one-out stream ablation on 3-class test set. Drop = decrease in Weighted F1 vs. full model (91.52%).

Removed Stream	Wt. F1 (%)	Δ
<i>None (full model)</i>	91.52	—
FTC (Fibonacci TextCNN)	87.11	−4.41
APWP (AI-prob. pooling)	89.80	−1.72
BTF (Bigram transition)	91.31	−0.21
Stylometry MLP	91.40	−0.12
CharCNN (byte-level)	91.46	−0.06
Comment stream	91.47	−0.05

The FTC stream’s dominant contribution (4.41%) confirms that Fibonacci-spaced multi-scale convolution captures syntactic patterns that no other stream replicates. The APWP contribution (1.72%) validates the learned token-level AI-ness weights as a genuine discriminative signal. BTF (0.21%) and Stylometry (0.12%) provide small but consistent improvements. CharCNN and Comment streams contribute marginally on average but are more impactful for the Adversarial class specifically, where byte-level surface statistics and comment-density ratios differ from machine-generated norms.

CoDet-M4 benchmark. Table 7 reports binary classification results on the CoDet-M4 dataset (Orel et al., 2025a). NanoDroid achieves **99.03%** F1 and **99.00%** accuracy (mean ± std over five seeds), surpassing all baselines including UniXcoder (98.65%) and CodeT5 (98.35%)—both of which are substantially larger models. This confirms that NanoDroid’s $O(L)$ primitives capture sufficient discriminative signal for binary detection without a pretrained backbone, and that its advantage on DroidCollection generalises to an independent benchmark.

5 Discussion and Conclusion

NanoDroid achieves 90.77% weighted F1 on the 4-class DroidCollection benchmark, within 2.6 points of UniXcoder (93.35%) and 2.4 points of

Table 7: Binary classification on CoDet-M4. NanoDroid is mean $\pm$ std over five seeds.

Model	P	R	F1	Acc.
Baseline	71.09	65.14	62.03	65.17
SVM	72.41	72.35	72.19	72.19
CatBoost	88.71	88.81	88.78	88.79
CodeBERT	95.70	95.72	95.70	95.71
CodeT5	98.36	98.35	98.35	98.35
UniXCoder	98.65	98.66	98.65	98.65
NanoDroid	98.80$\pm$0.04	99.25$\pm$0.05	99.03$\pm$0.04	99.00$\pm$0.04

DroidDetect_{CLS-Base} (93.16%), at 25–29 $\times$ lower parameter cost. On the independent CoDet-M4 binary benchmark it achieves **SOTA at 99.03% F1**, surpassing UniXcoder (98.65%) and CodeT5 (98.35%), confirming that its gains generalise beyond DroidCollection. At just 21 MB and with 20 $\times$ faster inference than 125M-parameter baselines, NanoDroid is the first detector realistically deployable at submission scale: a platform processing millions of code submissions daily can run NanoDroid inline without dedicated GPU clusters, a barrier that has historically blocked adoption of transformer-based detectors in production.

Per-class results reveal a clean difficulty ordering. NanoDroid is strongest on **Human** (F1 96.90) and **Adversarial** (F1 89.46, recall 91.96), confirming robustness without collapsing toward overpredicting machine authorship, a critical property for fair deployment where false positives carry real consequences for students and developers. **Machine** is recoverable (F1 86.27), while **Refined** remains the clear bottleneck (F1 78.28), occupying an intermediate regime where human and model cues coexist. Three $O(L)$ primitives (APWP, BTF, FTC) replace the pretrained encoder entirely, and per-stream attribution with per-token AI-ness scores make every prediction auditable: a reviewer can see exactly which stylistic evidence drove the decision, something a 90% confidence score from a black-box backbone cannot provide. This transparency is not cosmetic; it is a prerequisite for deployment in high-stakes settings such as academic integrity enforcement and secure code review. Future work will pursue distillation from larger detectors to close the remaining accuracy gap, targeted augmentation for the Refined boundary, and broader language generalisation to cover the full diversity of real-world codebases.

Limitations

NanoDroid’s most significant limitation is domain shift: without a pretrained language model backbone, the $O(L)$ streams encode domain-specific

stylistic norms from scratch rather than drawing on broad pretraining, leading to sharp OOD drops when training and test domains differ (Table 9). The Refined class also remains a persistent bottleneck (F1 78.28%), as human-edited LLM code occupies an intermediate stylistic regime where statistical and structural signals alone cannot cleanly separate it from either Human or Machine code. Future work will address both fronts: knowledge distillation from large pretrained detectors could close the domain-shift gap without sacrificing inference efficiency, while targeted data augmentation and a dedicated Refined-boundary loss could sharpen NanoDroid’s weakest decision boundary. Extending the prototype bank to cover lower-resource languages and stronger adversarial humanization techniques are additional priorities for making NanoDroid deployment-ready across the full breadth of real-world coding environments.

Ethical Considerations

NanoDroid is designed to detect AI-generated code in contexts where provenance matters: academic integrity, secure code review, and competitive programming. Like any classifier, it carries a non-zero false positive rate, and a wrongful flag against a student or developer can have serious consequences. We emphasise that NanoDroid should function as a *decision-support tool* surfacing auditable evidence for human review, not as an autonomous adjudicator. The per-stream attribution and per-token AI-ness scores are designed precisely to support this workflow: every prediction comes with an explanation that a human reviewer can accept, contest, or override.

A second concern is adversarial misuse. Publishing detection methods accelerates the arms race with humanization techniques; we accept this trade-off on the grounds that transparent, interpretable detectors raise the cost of evasion more sustainably than opaque ones, and that the research community benefits from open benchmarking. We do not release model weights trained to *generate* adversarial code. Finally, NanoDroid was trained and evaluated on DroidCollection, which reflects the distribution of current LLM outputs; deployment on future models or coding styles not represented in training should be preceded by domain-specific evaluation.

Broader Impact

The integration of LLMs into software development is accelerating, and the absence of scalable, interpretable provenance tools creates real risks across education, security, and platform integrity. NanoDroid directly addresses the deployment gap: existing SOTA detectors require infrastructure that most academic institutions, open-source forges, and mid-size engineering teams cannot afford to run at submission scale. A 21 MB model with CPU-viable inference lowers this barrier substantially, making meaningful AI-provenance checking accessible to the long tail of platforms that currently have no practical option.

Beyond detection, NanoDroid’s per-stream attribution offers a novel lens for studying how human and AI coding styles differ and where they converge. As LLM-assisted development becomes the norm, understanding these stylistic boundaries has value for tool designers, educators setting expectations around AI use, and researchers studying the evolving nature of software authorship. We hope NanoDroid serves both as a practical deployment artifact and as an interpretability instrument for this broader inquiry.

References

- Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. 2024. Fast-DetectGPT: Efficient zero-shot detection of machine-generated text via conditional probability curvature. In *The Twelfth International Conference on Learning Representations*.
- Savvas Chamezopoulos, Drahomira Herrmannova, Anita De Waard, Domenic Rosati, and Yury Kashnitsky. 2024. Overview of the dagpap24 shared task on detecting automatically generated scientific paper. In *Proceedings of the Fourth Workshop on Scholarly Document Processing (SDP 2024)*, pages 7–11.
- Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537.
- Liam Dugan, Alyssa Hwang, Filip Trhlík, Andrew Zhu, Josh Magnus Ludan, Hainiu Xu, Daphne Ippolito, and Chris Callison-Burch. 2024. [RAID: A Shared Benchmark for Robust Evaluation of Machine-Generated Text Detectors](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12463–12492, Bangkok, Thailand. Association for Computational Linguistics.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547.
- Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 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)*, pages 7212–7225.
- Hanxi Guo, Siyuan Cheng, Kaiyuan Zhang, Guangyu Shen, and Xiangyu Zhang. 2025. [CodeMirage: A Multi-Lingual Benchmark for Detecting AI-Generated and Paraphrased Source Code from Production-Level LLMs](#).
- Xinlei He, Xinyue Shen, Zeyuan Chen, Michael Backes, and Yang Zhang. 2024. [MGTBench: Benchmarking Machine-Generated Text Detection](#). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 2251–2265. ACM.
- Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Code-searchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.
- Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Mei Nagappan. 2024. [Whodunit: Classifying Code as Human Authored or GPT-4 Generated - A case study on CodeChef problems](#). In *Proceedings of the 21st International Conference on Mining Software Repositories, MSR ’24*, page 394–406, New York, NY, USA. Association for Computing Machinery.
- JianWang, Shangqing Liu, Xiaofei Xie, and Yi Li. 2024. An empirical study to evaluate aigc detectors on code content. In *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*, pages 844–856.
- Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. [Supervised contrastive learning](#). *CoRR*, abs/2004.11362.
- Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751.
- Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. 2023. [Discriminating Human-authored from ChatGPT-Generated Code Via Discernable Feature Analysis](#). In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*, pages 120–127. IEEE.

- Yafu Li, Qintong Li, Leyang Cui, Wei Bi, Zhilin Wang, Longyue Wang, Linyi Yang, Shuming Shi, and Yue Zhang. 2024. Mage: Machine-generated text detection in the wild. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 36–53.
- Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017a. [Focal loss for dense object detection](#). *CoRR*, abs/1708.02002.
- Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017b. [A structured self-attentive sentence embedding](#).
- Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.
- Dominik Macko, Róbert Móro, Adaku Uchendu, Jason Samuel Lucas, Michiharu Yamashita, Matús Pikuliak, Ivan Srba, Thai Le, Dongwon Lee, Jakub Simko, and Mária Bieliková. 2023. [MULTITuDE: Large-Scale Multilingual Machine-Generated Text Detection Benchmark](#). *CoRR*, abs/2310.13606.
- Elyas Masrour, Bradley Emi, and Max Spero. 2025. [DAMAGE: Detecting Adversarially Modified AI Generated Text](#). *CoRR*, abs/2501.03437.
- Jocelyn Mazarura, Alta de Waal, and Pieter de Villiers. 2020. [A gamma-poisson mixture topic model for short text](#). *Mathematical Problems in Engineering*, 2020(1):4728095.
- Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International conference on machine learning*, pages 24950–24962. PMLR.
- Takeru Miyato, Andrew M Dai, and Ian Goodfellow. 2016. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*.
- Diego Mollá, Qionghai Xu, Zijie Zeng, and Zhuang Li. 2024. Overview of the 2024 ALTA Shared Task: Detect Automatic AI-Generated Sentences for Human-AI Hybrid Articles. In *Proceedings of the 22nd Annual Workshop of the Australasian Language Technology Association*, pages 197–202, Canberra, Australia. Association for Computational Linguistics.
- Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. [GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT](#). *Journal of Systems and Software*, page 112059.
- 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. Aicd bench: A challenging benchmark for ai-generated code detection. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6913–6938.
- 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, Suzhou, China. Association for Computational Linguistics.
- Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. [Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education](#). In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '24*, page 1–11, New York, NY, USA. Association for Computing Machinery.
- Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31.
- Saiful Islam Salim, Rubin Yuchan Yang, Alexander Cooper, Suryashree Ray, Saumya Debray, and Sazadur Rahaman. 2024. Impeding LLM-assisted cheating in introductory programming assignments via adversarial perturbation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 445–463, Miami, Florida, USA. Association for Computational Linguistics.
- Tatiana Shamardina, Vladislav Mikhailov, Daniil Chernianskii, Alena Fenogenova, Marat Saidov, Anas-tasiya Valeeva, Tatiana Shavrina, Ivan Smurov, Elena Tutubalina, and Ekaterina Artemova. 2022. [Findings of the The RuATD Shared Task 2022 on Artificial Text Detection in Russian](#). In *Computational Linguistics and Intellectual Technologies*, page 497–511. RSUH.
- Eduard Tulchinskii, Kristian Kuznetsov, Laida Kushnareva, Daniil Cherniavskii, Sergey I. Nikolenko, Evgeny Burnaev, Serguei Barannikov, and Irina Piontkovskaya. 2023. [Intrinsic Dimension Estimation for Robust Detection of AI-Generated Texts](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. [Attention is all you need](#). In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
- Pengyu Wang, Linyang Li, Ke Ren, Botian Jiang, Dong Zhang, and Xipeng Qiu. 2023. SeqXGPT: Sentence-level AI-generated text detection. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1144–1156, Singapore. Association for Computational Linguistics.
- Yuxia Wang, Jonibek Mansurov, Petar Ivanov, Jinyan Su, Artem Shelmanov, Akim Tsvigun, Chenxi Whitehouse, Osama Mohammed Afzal, Tarek Mahmoud, Toru Sasaki, et al. 2024. M4: Multi-generator, multi-domain, and multi-lingual black-box machine-generated text detection. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1369–1407.
- Yuxia Wang, Artem Shelmanov, Jonibek Mansurov, Akim Tsvigun, Vladislav Mikhailov, Rui Xing, Zhuohan Xie, Jiahui Geng, Giovanni Puccetti, Ekaterina Artemova, Jinyan Su, Minh Ngoc Ta, Mervat Abassy, Kareem Ashraf Elozeiri, Saad El Dine Ahmed El Eter, Maiya Goloburda, Tarek Mahmoud, Raj Vardhan Tomar, Nurkhan Laiyk, Osama Mohammed Afzal, Ryuto Koike, Masahiro Kaneko, Alham Fikri Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. 2025. [GenAI Content Detection Task 1: English and Multilingual Machine-Generated Text Detection: AI vs. Human](#). In *Proceedings of the 1st Workshop on GenAI Content Detection (GenAIDetect)*, pages 244–261, Abu Dhabi, UAE. International Conference on Computational Linguistics.
- Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024. [Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference](#).
- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120.
- Lijun Wu, Juntao Li, Yue Wang, Qi Meng, Tao Qin, Wei Chen, Min Zhang, Tie-Yan Liu, et al. 2021. R-drop: Regularized dropout for neural networks. *Advances in neural information processing systems*, 34:10890–10905.
- Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025. [Distinguishing LLM-Generated from Human-Written Code by Contrastive Learning](#). *ACM Trans. Softw. Eng. Methodol.*, 34(4).
- Xianjun Yang, Kexun Zhang, Haifeng Chen, Linda R. Petzold, William Yang Wang, and Wei Cheng. 2023. Zero-shot detection of machine-generated codes. *arXiv preprint arXiv:2310.05103*.

A Full Model Architecture Details

A.1 Hyperparameter Summary

Table 8: NanoDroid hyperparameters for all settings.

Hyperparameter	3-class	4-class
Epochs	15	15
Batch size	64	128
Learning rate	5×10^{-5}	5×10^{-5}
Warmup ratio	0.1	0.1
Weight decay	0.01	0.01
Gradient clip	1.0	1.0
Projection dim	128	128
Max seq. length	512	512
SupCon weight λ_{sc}	0.1	0.1
Prototype weight	0.3	0.3
Uniformity weight λ_u	0.02	0.02
Mix loss weight λ_m	0.08	0.08
Router temperature	0.85	0.85
FGM ϵ	0.25	0.25
Seed	42	42

A.2 Component Architecture

Token Embeddings. Word embeddings $\mathbf{E} \in \mathbb{R}^{50267 \times 64}$, positional embeddings $\mathbf{P} \in \mathbb{R}^{514 \times 64}$, initialized with $\mathcal{N}(0, 0.02)$.

Lightweight Transformer. 3 TransformerEncoder layers, 4 attention heads, model dim 64, feed-forward dim 256, dropout 0.1, pre-norm (norm-first=True).

APWP Probe. Embedding $\mathbf{W}_{APWP} \in \mathbb{R}^{V \times 1}$, initialized to 0 (sigmoid(0)=0.5 $\rightarrow$ uniform at init).

BTF. Hash buckets $H = 8192$, bucket embedding dim $D = 64$, MLP: 128 $\rightarrow$ 256 with LayerNorm+GELU.

FTC. Kernels [2, 3, 5, 8, 13], 64 filters each, global max-pool, projection 320 $\rightarrow$ 256 with LayerNorm+GELU.

CharCNN. Input: UTF-8 byte IDs, CHAR_MAX=1024. Architecture: Embed(257, 32) $\rightarrow$ Conv1d(32, 64, $k = 5, s = 2$)+GELU $\rightarrow$ Conv1d(64, 128, $k = 5, s = 2$)+GELU $\rightarrow$ Conv1d(128, 256, $k = 5, s = 2$)+GELU $\rightarrow$ global mean-pool.

Stylometry MLP. 14 features: avg line length, line length std, avg indent level, trailing whitespace ratio, blank line ratio, comment ratio, keyword density (for/while/if), function density, avg identifier length, bigram repetition score, operator density, bracket density, docstring

presence, avg tokens per line. Architecture: Linear(14, 64)+ReLU+Linear(64, 256).

Language Embedding. 21 languages $\times$ 32 dim $\rightarrow$ Linear(32, 256).

Stream-Level Self-Attention (SLSA). Multi-headAttention(embed_dim=256, num_heads=4, dropout=0.1, batch_first=True) with residual connection and LayerNorm.

NanoDroid Comment Stream. 8-dimensional comment features (comment count, avg comment length, ratio to code lines, docstring presence, inline comment ratio, TODO density, multi-line comment presence, comment entropy) $\rightarrow$ Linear(8, 64)+LayerNorm+GELU+Linear(64, 256).

NanoDroid Router. Context-conditioned routing: Linear(512, 128)+GELU+Linear(128, 1) per stream; Softmax($\tau = 0.85$) $\rightarrow$ weighted sum of attended streams.

Classification Head. Linear(256, 256)+GELU+Dropout(0.1) +Linear(256, C) where $C \in \{2, 3, 4\}$.

Projection Head (SupCon). Linear(256, 128)+ReLU+Linear(128, 128).

Prototype Bank. Buffer shape: $[C, 21, 256]$ per-class per-language EMA centroids, updated with momentum = 0.99 during training. At inference, prototype cosine-similarity logits (temperature 0.5) are blended with classifier logits: $\ell_{\text{final}} = \ell_{\text{cls}} + 0.3 \cdot \ell_{\text{proto}}$.

B Dataset: DroidCollection

We use **DroidCollection** (Orel et al., 2025b) as our primary dataset.

B.1 Statistics

The collection comprises approximately **1.06 million** samples, pre-partitioned into Train 847k (80%), Dev 106k (10%), Test 106k (10%). The test split which we use for all reported numbers—contains 49,932 Human, 42,664 / 26,524 Machine (3-class / 4-class split), 13,230 Refined, and 16,140 Adversarial samples.

B.2 Labeling Schema

DroidCollection uses a 4-class schema: (1) **Human**: ground-truth human code from competitive programming and OSS; (2) **Machine**: LLM zero-shot/few-shot outputs; (3) **Machine-Refined**: human code modified by an LLM; (4) **Adversarial**:

Table 9: OOD weighted F1 (%) on DroidCollection. Rows indicate the training domain and columns indicate the test domain.

Train ↓ / Test →	General use Algorithmic Research		
Algorithmic problems	45.05	89.68	58.29
General use code	89.78	26.65	78.87
Research code	37.79	39.38	59.51

Split	Total	Human	Machine	Refined	Adv.
Train	846,598	399,448	212,193	105,842	129,115
Dev	105,824	49,930	26,524	13,231	16,139
Test	105,826	49,932	26,524	13,230	16,140

Table 10: Exact split sizes and per-class counts for the 4-class DroidCollection setup.

code explicitly humanized to evade detectors. Experiments use 2-, 3-, and 4-class subsets by collapsing or including the adversarial class.

B.3 Diversity

The dataset covers 9 programming languages (Python, Java, C++, C, C#, Go, JavaScript, TypeScript, PHP), 43 coding models across 13 families (GPT-4o, Llama-3, Qwen-2.5, Phi-3, etc.), and three coding domains: general use, algorithmic/competitive, and research/data science. DroidCollection will be described consistently as containing 9 programming languages in our evaluation. The “21 languages” phrase refers to a fixed 21-entry language-embedding vocabulary for compatibility with broader corpora; DroidCollection uses a subset, and unseen languages map to Unknown.

B.4 Detailed Split Statistics

Table 10 reports exact split sizes, per-class counts, and code-length statistics for DroidCollection under the 4-class labeling scheme used throughout this paper. The test split (105,826 examples) matches the sum of the per-class counts reported in Section 4.3 and Table 3 (49,932 Human + 26,524 Machine + 13,230 Refined + 16,140 Adversarial). Length statistics are computed over the full test split and reported as medians with 90th-percentile (p90) values to characterize the tail.

We additionally provide label-by-language and label-by-domain breakdowns to support auditability of the evaluation setup; see Figures 11 and 13 (Appendix D.5) for the corresponding per-language and per-domain class-conditional recall heatmaps.

Split	Median chars	Median lines	Median tokens
Train	913	30	93
Dev	913	30	94
Test	918	30	94

Table 11: Code-length statistics (median values) across splits. On the test split, p90 values are 2,933 characters, 95 lines, and 281 whitespace tokens, indicating a moderate-length right tail.

C Extended Related Work and Dataset Comparisons

Table 12 (reproduced from Orel et al. 2025b) contextualizes DroidCollection against prior AI-generated code detection datasets.

Table 12: Comparison of AI-generated code detection datasets (from Orel et al. 2025b, Table 1).

Dataset	Size	Languages	Models
GPT-Sniffer	7.4K	1	1
CodeGPTSensor	1.1M	1	1
Whodunit	1.6K	1	10
CodeBet	501K	2	5
CodeMirage	210K	1	10
DroidCollection	1.06M	9	46

Table 13: Seed sensitivity analysis of NanoDroid on the 3-class DroidCollection benchmark.

Seed	Wt. F1 (%)
1	91.56
2	91.33
3	91.32
4	91.21
42	91.18
Mean ± Std	91.32 ± 0.15

D Interpretability Analysis

D.1 Ablations, Routers and Geometry

A core advantage of NanoDroid over backbone-based detectors is that interpretability is not bolted on after the fact: it is an architectural property. Every prediction decomposes into the contributions of named, human-readable evidence streams, every token carries an explicit AI-likelihood weight (APWP), and every class is anchored to per-language prototype centroids that can be inspected directly. This matters operationally. A detector flagging a student’s submission, a CI commit, or a competitive-programming entry as machine-generated must be auditable by a human reviewer;

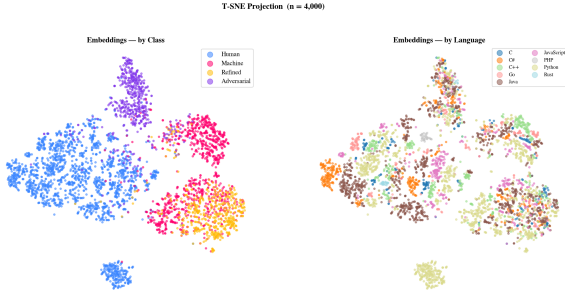


Figure 2: T-SNE projection of the learned representation space, colored by class (left) and by programming language (right). The class panel shows that Human occupies a coherent region, Machine and Adversarial form distinct but adjacent zones, and Refined sits on the boundary between them, exactly the position a true bridge class should occupy under the DroidCollection labeling schema. The language panel rules out the failure mode where the embedding space is silently organized by programming language rather than by authorship.

a 90% confidence score from a 125M-parameter backbone is not a justification, whereas “BTF and FTC together account for 78% of the routed evidence, and the FTC signature matches the machine-refined prototype” is.

What the ablations show. Table 6 makes the division of labor clear. Removing FTC causes by far the largest drop (−4.41 Weighted F1), indicating that multi-scale local pattern extraction is the dominant discriminative signal in NanoDroid. APWP is the strongest complementary stream (−1.72), showing that learned token-level AI-likeness remains useful even after fusion with richer features. The remaining streams have small average leave-one-out effects, which is what we would expect from genuinely complementary cues rather than redundant copies of the same signal.

What the router learns. Table 14 shows a subtle but important distinction between *importance* and *usage*. The router assigns the largest average mass to BTF (0.451), followed by Comment (0.190) and FTC (0.154), meaning that cheap transition and comment-pattern signals are consulted frequently. Yet the ablation results show FTC is more critical than its raw routing mass suggests. Taken together, these results indicate that BTF acts as a broad, high-coverage signal, while FTC contributes the strongest high-value evidence when fine-grained structural discrimination is needed. Stylometry receives slightly higher weight on correct predictions (0.107 vs. 0.104), suggesting that global writing style is especially useful once the model is on the

Table 14: Average router weights in the 3-class model. Weights are averaged over all test examples, with the second column restricted to correctly classified samples.

Stream	Mean Weight	Mean Weight (Correct)
APWP	0.006	0.006
BTF	0.451	0.465
FTC	0.154	0.137
CharCNN	0.095	0.095
Stylometry	0.104	0.107
Comment	0.190	0.190

right side of the decision boundary.

Geometry and qualitative analysis. Figure 2 confirms that the supervised contrastive objective and prototype bank induce a semantically meaningful embedding space: Human and Machine are well separated, and Refined lies between them as expected. The main paper focuses on these quantitative takeaways, while Figure 3 exposes how the fusion module combines streams and Figure 4 highlights why comment-derived signals remain useful, especially for harder boundary cases. Figure 5 further consolidates these signals into a unified interpretability view, showing how stream disagreement and mixed-authorship indicators expose stylistic collisions between Human- and Machine-like code. The visualization also explains why Refined and Adversarial samples remain the most challenging categories for the model. We defer the additional plots to Appendix.

D.2 Additional Diagnostic and Interpretability Analysis

This section collects the full set of diagnostic figures supporting Section D, organized into model trust (calibration and training stability), error structure, cross-language and cross-domain robustness, and the mechanistic behavior of the streams and the learned representation space. All figures are computed on the full test split, and cross-references to figures and tables in the main paper are intentional and not duplicated here.

D.3 Calibration and Training Stability

A useful detector is one whose confidence can be trusted, not just one whose accuracy is high. The two figures in this subsection answer that question from complementary angles: reliability diagrams check whether the predicted probabilities are well calibrated, and the training trajectories check that the headline test number is the result of a stable

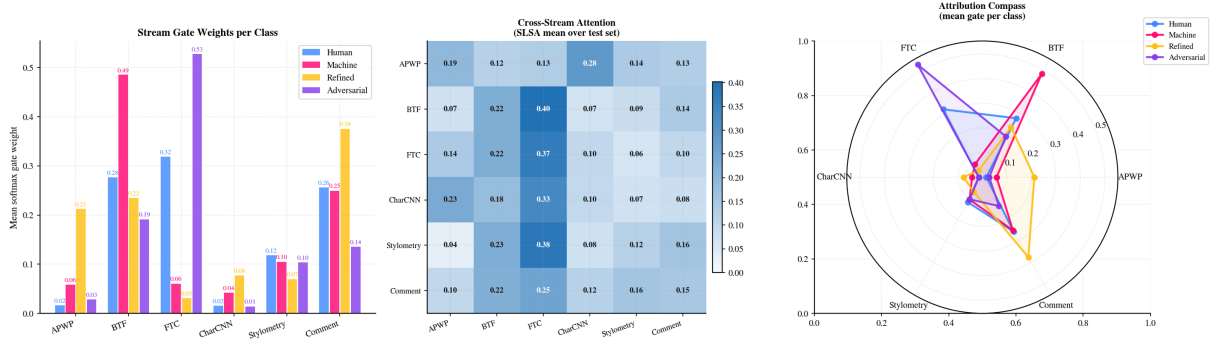


Figure 3: The grouped bars (left) show per-class softmax routing weights, the cross-stream attention map (center) shows which streams attend to which inside SLSA, and the polar compass (right) compresses each class into a recognizable evidence shape. Different labels are decided by different mixtures of evidence rather than a single monolithic decision rule, which is the property that makes the model auditable.

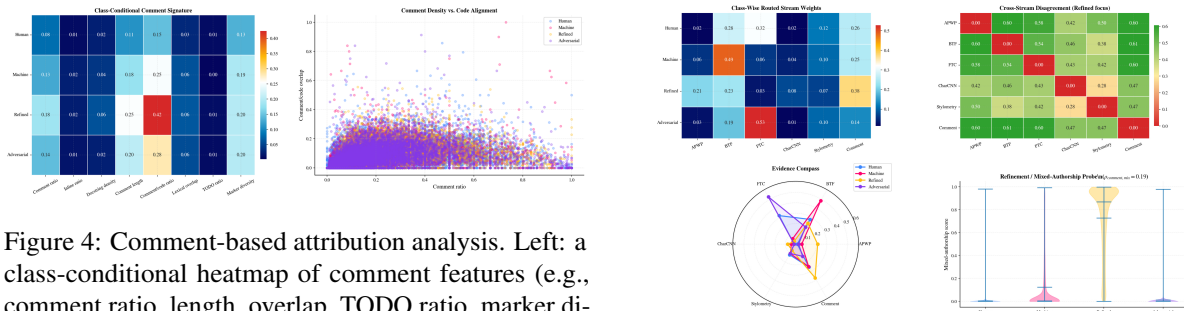


Figure 4: Comment-based attribution analysis. Left: a class-conditional heatmap of comment features (e.g., comment ratio, length, overlap, TODO ratio, marker diversity) showing how commenting patterns differ across Human, Machine, Refined, and Adversarial code. Right: scatter plot of comment ratio vs. comment/code overlap, illustrating whether comments are merely present or semantically aligned with code. These patterns help explain why the comment stream influences routing decisions.

optimization process rather than a memorized endpoint.

D.4 Error Structure

The aggregate weighted F1 hides which mistakes the model actually makes and how those mistakes trade off at different operating points. The three figures below decompose the errors into a confusion matrix (showing where they accumulate), precision-recall curves (showing how they trade off as thresholds are moved), and an error attribution view (showing which classes and languages drive the residual error budget).

D.5 Cross-Language and Cross-Domain Robustness

A code detector that performs well only on its highest-resource language or only on its training domain is not deployable. The four figures in this subsection probe two distributional axes: program-

Figure 5: Integrated evidence board for the 4-class model. The figure combines routed stream weights, disagreement structure, and mixed-authorship indicators in a unified interpretability view. High disagreement is often informative rather than undesirable, highlighting stylistic collisions such as machine-generated code edited to appear human-written. These patterns help explain why adversarial and refined examples remain difficult to classify.

ming language (syntax, ecosystem, and tokenizer behavior) and collection domain (repository style and problem source). Across both axes, performance shifts are concentrated in the Refined class rather than spread across all four labels.

D.6 Stream Mechanisms and Representation Geometry

This subsection answers the question that backbone-based detectors cannot: what evidence drove the prediction? The first figure opens up the comment stream, which carries small average ablation impact but unusually high routing weight, and shows that it keys on editorial pattern rather than comment presence. The remaining three figures expose the joint structure of the routed evidence, the per-class prototype geometry, and the t-SNE pro-

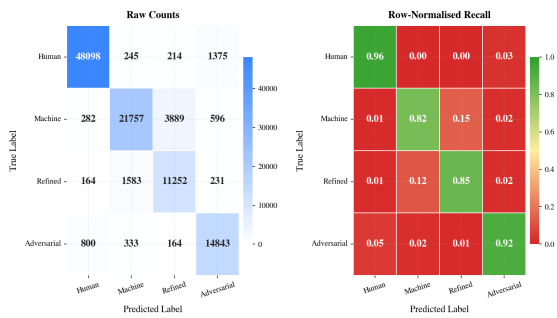


Figure 6: Confusion matrices of NanoDroid on the 4-class test set. The raw-count matrix highlights error concentration, while the row-normalized matrix shows class-wise recovery rates. Human samples are classified reliably, whereas Refined code remains the most challenging class due to overlap with both Machine and Adversarial samples.

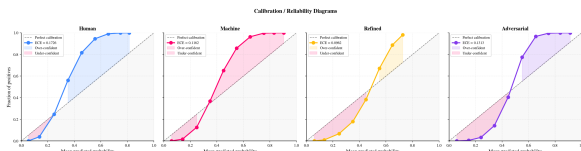


Figure 7: Class-wise reliability diagrams comparing predicted confidence to empirical correctness. Human and Adversarial track the diagonal closely, supporting use of their probability outputs as deployment-ready risk signals. Machine and Refined deviate more in the mid-confidence bins, the calibration signature of classes whose decision boundaries genuinely overlap rather than of an under-trained model.

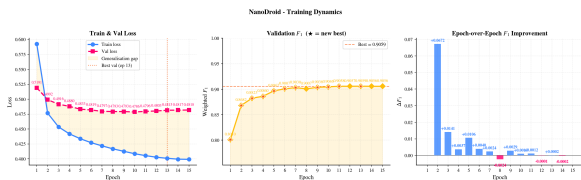


Figure 8: Training and validation trajectories. Validation weighted F1 climbs rapidly in the first three epochs and plateaus without late divergence; the small train and validation loss gap is consistent with XL-SupCon and R-Drop acting as regularizers rather than capacity boosts. The trajectory rules out the failure mode where the headline test number reflects an unstable or memorized endpoint.

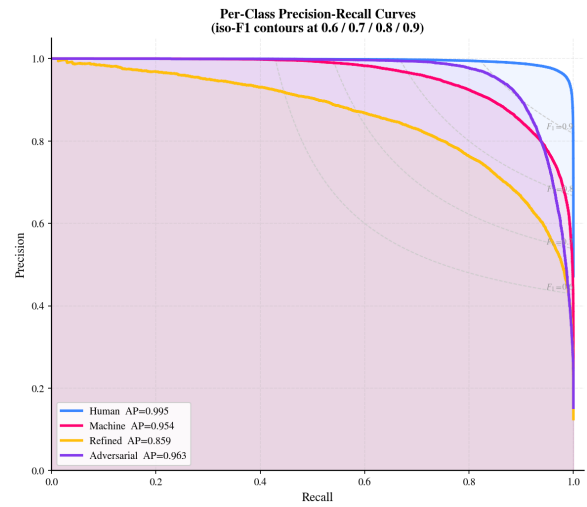


Figure 9: Per-class precision-recall curves with iso-F1 contours. Human and Adversarial hug the upper-right corner, so their decision thresholds can be moved without sharply degrading either metric. Refined shows the steepest trade-off, which means deployments that prioritize recall on refined code should lower its threshold and accept additional Machine-to-Refined false positives.

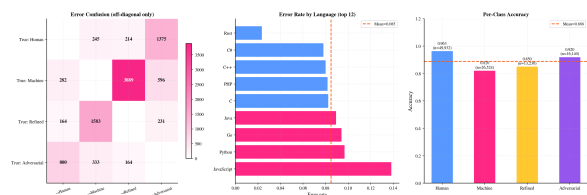


Figure 10: Dominant off-diagonal mistakes, per-language error rates, and per-class accuracy. Mistakes concentrate on Refined against Machine and Refined against Adversarial, never on Human against Machine in either direction. The model has therefore learned the high-level human and non-human boundary cleanly and fails only in intermediate stylistic regimes.

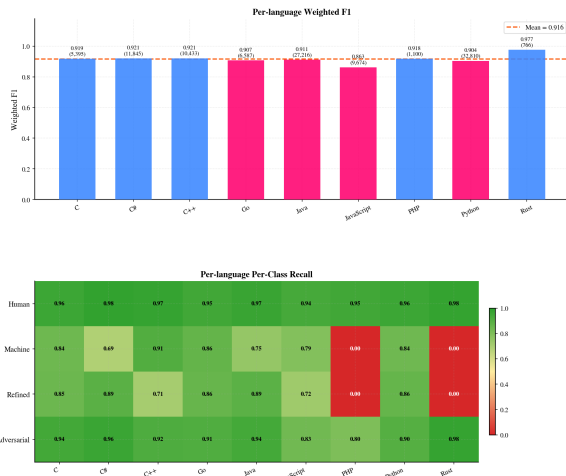


Figure 11: Per-language weighted F1 (top) with class-conditional recall (bottom). Performance is stable across all nine languages within roughly five F1 points, ruling out the case in which the global score is dominated by Python. The recall heatmap shows the recurring pattern: Human and Adversarial recall stay strong everywhere, while Refined is the first class to drop under language shift, evidence that the Refined bottleneck is class-level rather than a language-coverage artifact.



Figure 12: Full language by class recall heatmap. Cold cells identify the language and class pairs where deployment requires caution; warm cells identify regimes where the model can be deployed with high confidence. For per-language deployment decisions this is more informative than the aggregate weighted F1.

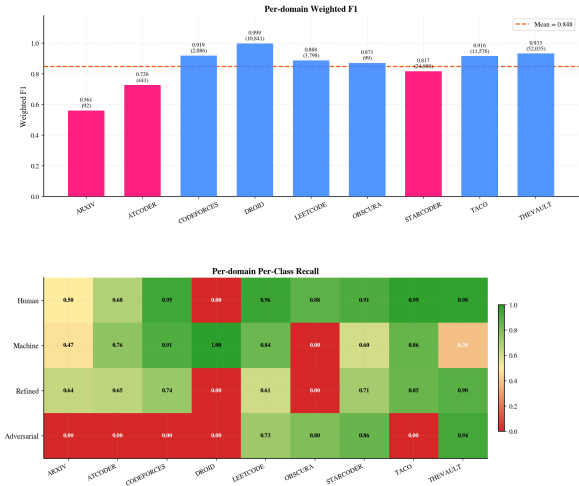


Figure 13: Per-domain weighted F1 (top) with class-conditional recall (bottom). Where the language plots stress-test syntactic and ecosystem variation, this view stress-tests collection-source variation. Performance shifts are moderate and structured: harder domains are harder because Refined recall drops, not because the overall system fails uniformly.

jection of the full representation space. Together they explain why Refined remains the hardest class and why disagreement across streams can be used as a built-in uncertainty signal.

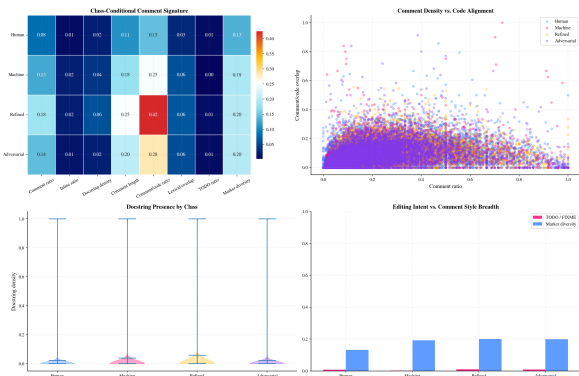


Figure 14: Class-conditional comment statistics (left): comment ratio, length, code overlap, TODO ratio, and marker diversity differ systematically across Human, Machine, Refined, and Adversarial. The scatter of comment ratio against comment-code overlap (right) confirms that the comment stream keys on the shape and code-alignment of comments rather than on their mere presence, editorial behavior that is difficult to fake consistently across long files.

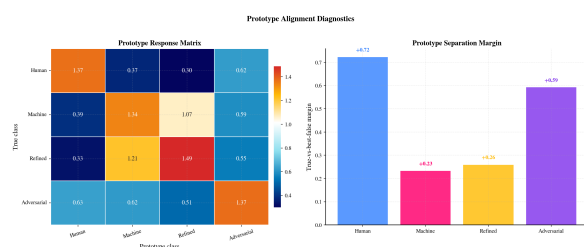


Figure 15: Per-language EMA prototype responses (left) and class separation margins (right). The diagonal-dominant response matrix confirms that prototype anchors are meaningful rather than decorative. Refined carries the tightest separation margin, giving a representation-level explanation for why it remains the hardest class: its evidence is structured enough to form a recognizable region but not far enough from Machine or Human to be cleanly carved out.