Dream at SemEval-2026 Task 13: SALSA for Single-Pass Machine-Generated Code Detection

Ruslan Berdichevsky
Dream Security Ltd.
Tel Aviv, Israel
ruslan@dreamgroup.com

Shai Nahum-Gefen
Dream Security Ltd.
Tel Aviv, Israel
shai@dreamgroup.com

Elad Ben-Zaken
Dream Security Ltd.
Tel Aviv, Israel
elad@dreamgroup.com

Abstract

Large language models have transformed code generation, raising concerns around authorship, assessment integrity, and software trust. SemEval-2026 Task 13 Subtask A operationalizes detection as binary classification over code snippets, with a particular emphasis on out-of-distribution (OOD) generalization across unseen programming languages and application domains. We propose a SALSA-style formulation, Single-pass Autoregressive LLM Structured Classification, that maps each class to a dedicated output token and trains the model to emit a *single-token* label in a structured response. Rather than engineering handcrafted features or decision rules, this formulation *delegates* the authorship decision to the model. To improve OOD robustness, we combine balanced sampling across languages with parameter-efficient fine-tuning and conservative training (low learning rate, single epoch) to avoid overfitting to the training domain. Our best system achieves OOD $F_1 = 0.789$ on the official leaderboard, substantially outperforming the CodeBERT baseline ($F_1 = 0.305$).

1 Introduction

Large language models (LLMs) have revolutionized code generation, but this has important consequences for programming skills, ethics, and assessment integrity, making the detection of LLM-generated code essential for maintaining accountability and standards (Orel et al., 2026b). While there is prior work on machine-generated code detection, it often lacks domain coverage and robustness, and typically supports only a small number of programming languages (Orel et al., 2025b,a, 2026a).

SemEval-2026 Task 13 Subtask A provides an important testbed by requiring a binary label indicating whether a code snippet is machine-generated. A central challenge is generalization to out-of-distribution (OOD) settings, since practical detectors must handle unseen scenarios beyond the training distribution.

Rather than solving the problem via hand-designed features or decision rules, we *delegate* the authorship decision to the model by specifying the task as a natural language instruction, since language models are strong general-purpose learners when prompted with task descriptions (Brown et al., 2020). However, using LLMs as classifiers this way is often brittle: the model may emit verbose explanations, inconsistent label formats, or multi-token labels. SALSA[1], Single-pass Autoregressive LLM Structured Classification (Berdichevsky et al., 2025), addresses this by mapping each class to a distinct output token and structuring the prompt with clear delimiters and a direct authorship question, turning classification into a reliable constrained next-token prediction.

During fine-tuning, the model simultaneously learns task alignment (following the classification instruction) and in-domain expertise (adapting to the training distribution). Since our primary goal is out-of-distribution (OOD) performance, we bias training toward preserving task alignment and general capabilities rather than over-fitting to the source domain. To this end, we run only one epoch and select the best checkpoint based on validation; prior work suggests that earlier training tends to improve general-purpose features, while later stages may over-specialize to the source domain and degrade OOD performance (Liu and Gurevych, 2024). We also use a low learning rate, as we empirically observe that larger learning rates tend to reduce OOD performance while improving in-distribution metrics. Together, these choices keep the fine-tuned model closer to the base model, which tends to improve generalization.

SALSA has shown consistent results across di-

[1]Our code is publicly available at https://github.com/dreamgroupai-ai/SALSA.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 354–360
July 3-4, 2026

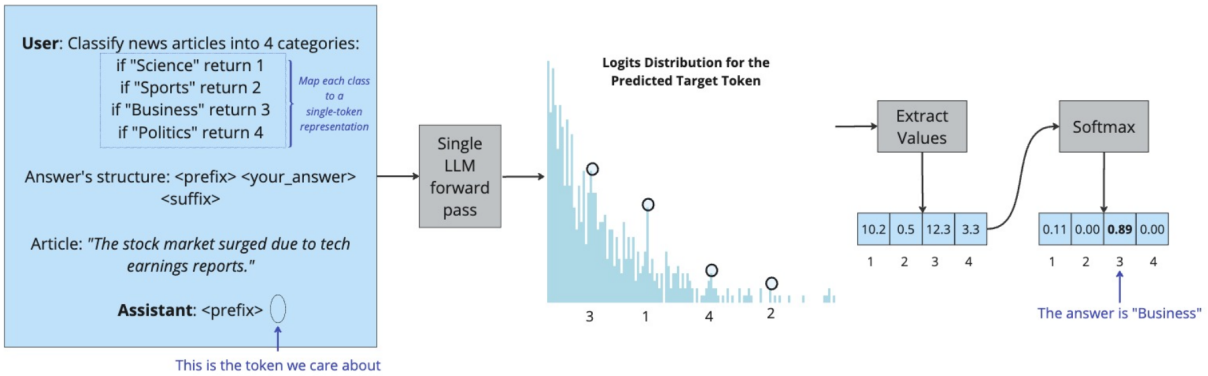

Figure 1: Overview of the SALSA classification pipeline: the structured prompt maps each class to a single token; a single LLM forward pass produces logits for the target token position; the class-token logits are extracted and passed through a softmax to obtain the predicted label.

verse classification settings, including the GLUE benchmark (Wang et al., 2019) and content moderation (Sorensen et al., 2025; Berdichevsky, 2025); here we apply it to machine-generated code detection and report strong OOD performance ($F_1 = 0.789$).

2 Background

Subtask A is a binary classification setting: given an input code snippet, the system predicts `0` (human-written) or `1` (machine-generated). The task is designed to stress-test generalization across both programming languages and application scenarios: training data uses `C++`, `Python`, and `Java` from an algorithmic domain, while evaluation includes (i) seen vs. unseen languages and (ii) seen vs. unseen domains (research/production), yielding four settings in total.

We use the official dataset distribution released by the task organizers. The dataset provides the code snippet (`code`), the binary label (`label`), and metadata including programming language (`language`). The released sizes for Subtask A are 500K training samples (238K human-written and 262K machine-generated) and 100K validation samples, and a private test set of 500K samples for which no labels or metadata are provided.

3 System Overview

3.1 SALSA

SALSA reframes classification as single-token generation over an autoregressive LLM. Each class is mapped to a unique output token, and the prompt is structured to elicit exactly that token at a fixed position. At inference time, a single forward pass suffices: the output logits are projected onto the class tokens only, yielding an efficient classifier with minimal decoding overhead. Figure 1 illustrates the end-to-end pipeline, and Figure 2 shows an example prompt for our task.

Class-to-token mapping. For Subtask A we use label tokens `"0"` and `"1"` (a single token each under the model tokenizer).

Structured prompting. We wrap the code snippet in delimiters and ask the model whether the code is machine generated. We require the answer in the form `<ANSWER>#</ANSWER>` where `#` is `0` or `1`. Figure 2 shows the prompt structure used for the task. The `/no_think` flag and an empty `<think>` block suppress chain-of-thought reasoning in Qwen3, keeping the output to a single label token (Qwen Team, 2025).

3.2 Implementation Details

Training objective We implement SALSA by supervising only the label token position in the assistant response. Concretely, the model is trained with a causal language modeling objective where the loss is evaluated exclusively at the final occurrence of the label token, so gradients update the model primarily to improve the next-token distribution over the target classes.

Parameter-efficient fine-tuning We fine-tune with LoRA, Low-Rank Adaptation (Hu et al., 2022), to reduce overfitting risk and speed up training.

Conservative fine-tuning. We use a low learning rate (5×10^{-6}), train for one epoch, and select the best checkpoint on validation. We empirically observe that more aggressive training reduces OOD

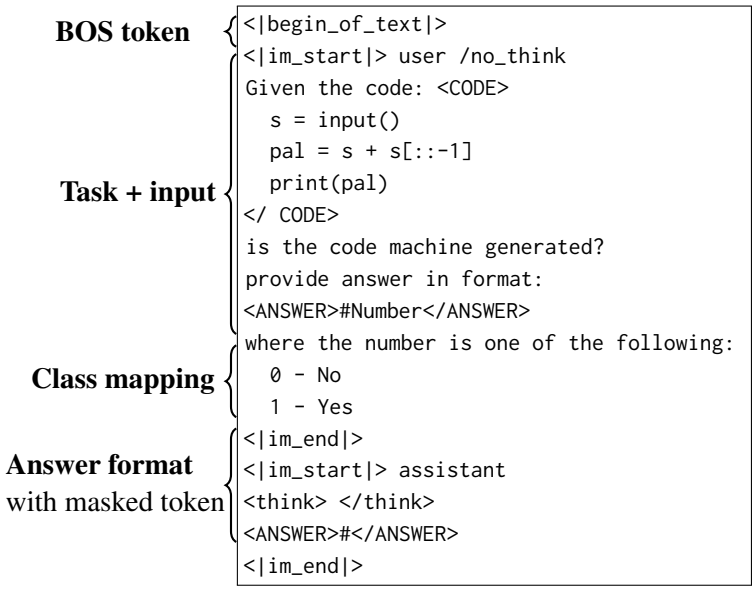

Figure 2: Example SALSA-structured prompt for machine-generated code detection (Subtask A).

Split	Language	Human (0)	Machine (1)	Total
Train	C++	11,147	12,245	23,392
	Java	9,225	10,077	19,302
	Python	218,103	239,203	457,306
	Total	238,475	261,525	500,000
Val.	C++	2,230	2,449	4,679
	Java	1,845	2,015	3,860
	Python	43,620	47,841	91,461
	Total	47,695	52,305	100,000

Table 1: Label distribution by split and programming language.

performance while improving in-distribution metrics. In particular, training beyond a single epoch consistently degrades OOD generalization, suggesting that extended exposure to the source domain causes the model to over-specialize at the expense of its pretrained representations.

Dataset splits. Table 1 shows the per-language and per-label breakdown of the training (500K) and validation (100K) splits. The dataset is heavily skewed toward `Python`, which accounts for 91.5% of all samples, while `C++` and `Java` together comprise less than 9%. The label distribution is roughly balanced overall (47.7% human vs. 52.3% machine-generated), and this balance is preserved within each language; the validation split closely mirrors the training proportions.

Balanced sampling As shown in Table 1, the training set is highly imbalanced across programming languages. To prevent majority-language dominance, we balance the training data by sampling an equal number of examples from each `(language, label)` group. The number of samples drawn per group is determined by the size of the smallest group; samples are drawn without repetition, so each training example is used at most once. This discards a substantial portion of Python training data, a deliberate trade-off: preserving Python's dominance would steer fine-tuning toward in-domain expertise on Python-specific patterns at the expense of the task alignment carried over from the pretrained starting point. In preliminary experiments, the unbalanced distribution yielded weaker task alignment (lower OOD F_1), consistent with this view.

Inference Our inference follows the SALSA principle of restricted logit projection: we generate a single token and read the model's log-probabilities for the class tokens only. The predicted label is the argmax over the two target token logits. We achieve high-throughput inference by merging the LoRA adapter into the base model and using vLLM (Kwon et al., 2023) to compute logprobs efficiently.

4 Experimental Setup

Configuration. We fine-tune Qwen3-8B, Qwen3-32B and Qwen2.5-72B-Instruct (Qwen Team, 2024) with LoRA adapters using the SALSA structured prompt described in Section 3. Full hyperparameter details are provided in Appendix A. We train for up to 1 epoch, selecting the checkpoint with the best validation F_1, and use left truncation

Model	Test F_1	Val. F_1	Train F_1
Baseline			
CodeBERT	0.305	–	–
SALSA Zero-shot - without tuning			
Qwen3-Next-80B-A3B-Instruct	0.591	0.672	0.662
Qwen2.5-72B-Instruct	0.449	0.359	0.358
Qwen3-32B	0.508	0.530	0.528
Qwen3-8B	0.436	0.523	0.523
SALSA Tuned			
Qwen2.5-72B-Instruct	0.789	0.996	0.996
Qwen3-32B	0.760	0.994	0.994
Qwen3-32B official†	0.730	0.989	0.982
Qwen3-8B	0.450	0.991	0.991

Table 2: Results on SemEval-2026 Task 13 Subtask A. CodeBERT is the organizers' baseline. Test F_1 denotes the leaderboard OOD score. Zero-shot uses the SALSA prompt without fine-tuning. †*Note:* The official Kaggle submission achieved a test (OOD) F_1 of 0.73 due to suboptimal checkpoint selection; a post-competition ablation improved this to 0.76.

at 8192 tokens to preserve the label token at the end of the prompt.

Metric. We report macro-F_1. For in-distribution performance, we evaluate on the official Kaggle validation split as described in the challenge ($F_1 = 0.996$ in our experiment). For out-of-distribution performance, we report the competition OOD test F_1 (0.789) on the Kaggle leaderboard.

5 Results

Table 2 summarizes our results. The official CodeBERT baseline scores 0.305 OOD F_1. Our best submission, Qwen2.5-72B-Instruct fine-tuned with SALSA, achieves 0.789 OOD F_1, a substantial improvement over both the baseline and the zero-shot upper bound for that model (0.449).

Zero-shot prediction bias. The precision/recall breakdown in Table 3 reveals that the aggregate F_1 scores in Table 2 mask qualitatively different failure modes. Qwen3-Next-80B-A3B-Instruct is the only zero-shot model with balanced precision and recall (0.707 / 0.638), while Qwen3-32B and Qwen2.5-72B-Instruct are strongly biased toward predicting human-written code, achieving recall of only 0.242 and 0.035 respectively despite high precision. This near-total failure to detect machine-generated code explains the val/OOD inversion for Qwen2.5-72B-Instruct (val F_1=0.359 vs. OOD F_1=0.449): the OOD distribution is apparently less dominated by machine-generated examples, so the model's conservative bias is less penalized. The

Model (Zero-shot)	Prec.	Rec.	Val F_1
Qwen3-Next-80B-A3B-Instruct	0.707	0.638	0.672
Qwen3-32B	0.844	0.242	0.530
Qwen3-8B	0.572	0.397	0.523
Qwen2.5-72B-Instruct	0.910	0.035	0.359

Table 3: Zero-shot validation performance. Prec./Rec. denote precision and recall for the machine-generated class; F_1 is macro-averaged. The high-precision/low-recall asymmetry indicates a bias toward predicting human-written code; per-language breakdown in Appendix C.

breakdown also directly motivates our balanced sampling strategy: a model that defaults to predicting "human" will score well on a skewed distribution but generalize poorly. The per-language breakdown in Appendix C shows no strong language-level skew in the validation set.

Validation results. All fine-tuned models achieve near-perfect validation F_1: 0.991 (8B), 0.994 (32B), and 0.996 (72B), essentially matching their training-split F_1. The absence of a train–val gap indicates in-distribution fitting without overfitting. While the differences are small in absolute terms, the scores increase monotonically with model size, consistent with the OOD trend. We use the validation score for checkpoint selection within training, but do not draw strong conclusions from cross-model comparisons given the narrow range. Extended validation error analysis is provided in Appendix D.

Effect of fine-tuning. Notably, all zero-shot models already outperform the CodeBERT baseline (0.305), with scores of 0.359–0.672, suggesting that general-purpose LLMs carry useful prior knowledge for this task even without task-specific training. Fine-tuning with SALSA yields substantial further gains for larger models, with both Qwen2.5-72B-Instruct and Qwen3-32B improving markedly over their zero-shot counterparts. For Qwen3-8B, the gain is negligible, indicating that fine-tuning alone is insufficient for small models in this setting.

6 Conclusion

We applied SALSA to SemEval-2026 Task 13 Subtask A, achieving an OOD F_1 of 0.789 with Qwen2.5-72B-Instruct, placing among the top scores on the official leaderboard. SALSA is a natural fit for this task: by reducing the output space

to two tokens and delegating the authorship decision to the model via structured prompting and tuning, it yields stable, efficient classification. OOD performance improves monotonically with model scale, which we attribute to larger models retaining more of their pretrained knowledge after conservative fine-tuning rather than over-specializing to the training domain. Interestingly, Qwen2.5-72B-Instruct leads after fine-tuning, yet its zero-shot performance is clearly weaker than Qwen3 models, suggesting that Qwen3's stronger instruction alignment is beneficial in the zero-shot regime while Qwen2.5's broader pretraining knowledge transfers better under fine-tuning. The MoE model achieves the strongest zero-shot alignment; since our current implementation does not support MoE fine-tuning, exploring that direction is left for future work. Future work should focus on improving task instruction alignment without inducing training-domain specialization.

References

Ruslan Berdichevsky. 2025. 4th place solution (jigsaw agile community rules) — salsa-based writeup reference. Kaggle Writeup.

Ruslan Berdichevsky, Shai Nahum-Gefen, and Elad Ben Zaken. 2025. Salsa: Single-pass autoregressive LLM structured classification. arXiv preprint. ArXiv:2510.22691.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*.

Cecilia Chen Liu and Iryna Gurevych. 2024. Early period of training impacts adaptation for out-of-distribution generalization: An empirical study. arXiv preprint. ArXiv:2403.15210.

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, Vienna, Austria. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. 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)*, Rabat, Morocco. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026b. 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 31251–31277.

Qwen Team. 2024. Qwen2.5 technical report.

Qwen Team. 2025. Qwen3 technical report.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3514.

Jeffrey Sorensen, Lucas Dos Santos, Lucy Vasserman, María Cruz, Tin Acosta, and Walter Reade. 2025. Jigsaw — agile community rules classification. `https://kaggle.com/competitions/jigsaw-agile-community-rules`. Kaggle.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 7th International Conference on Learning Representations*.

A Hardware and Hyperparameters

All models were fine-tuned on 2 NVIDIA B200 GPUs using DeepSpeed (Rasley et al., 2020) ZeRO stage 0 and for more than 70B stage 3. We trained

Model	Lang	N	TP	TN	FP	FN	F_1	Prec.	Rec.
Qwen3-Next-80B-A3B-Instruct	C++	4,679	1,277	2,033	197	1,172	0.700	0.866	0.521
	Java	3,860	1,040	1,710	135	975	0.704	0.885	0.516
	Python	91,461	31,028	30,134	13,486	16,813	0.669	0.697	0.649
	All	**100,000**	**33,345**	**33,877**	**13,818**	**18,960**	**0.672**	**0.707**	**0.638**
Qwen3-32B	C++	4,679	564	2,174	56	1,885	0.530	0.910	0.230
	Java	3,860	361	1,808	37	1,654	0.490	0.907	0.179
	Python	91,461	11,746	41,368	2,252	36,095	0.532	0.839	0.246
	All	**100,000**	**12,671**	**45,350**	**2,345**	**39,634**	**0.530**	**0.844**	**0.242**
Qwen3-8B	C++	4,679	875	1,318	912	1,574	0.464	0.490	0.357
	Java	3,860	666	1,274	571	1,349	0.490	0.538	0.331
	Python	91,461	19,213	29,592	14,028	28,628	0.528	0.578	0.402
	All	**100,000**	**20,754**	**32,184**	**15,511**	**31,551**	**0.523**	**0.572**	**0.397**
Qwen2.5-72B-Instruct	C++	4,679	106	2,188	42	2,343	0.364	0.716	0.043
	Java	3,860	60	1,815	30	1,955	0.352	0.667	0.030
	Python	91,461	1,644	43,512	108	46,197	0.360	0.938	0.034
	All	**100,000**	**1,810**	**47,515**	**180**	**50,495**	**0.359**	**0.910**	**0.035**

Table 4: Zero-shot per-language results on the validation set. TP/TN/FP/FN are counts for the machine-generated class (label=1). The human-prediction bias of Qwen2.5-72B-Instruct and Qwen3-32B is consistent across all languages.

for at most one epoch with a learning rate of 5×10^{-6}, a per-GPU batch size of 1, and gradient accumulation over 32 steps (effective batch size 64). Input sequences were left-truncated to 8,192 tokens to ensure the label token is always retained at the end of the prompt. LoRA adapters were configured with rank r=16, scaling factor α=32, and dropout 0.05.

B Inference Runtime

Table 5 reports wall-clock inference times for a full evaluation run over the 500K test records, measured on 2 B200 GPUs using vLLM (Kwon et al., 2023).

Model	Runtime (hh:mm)
Qwen3-Next-80B-A3B-Instruct	1:59
Qwen2.5-72B-Instruct	4:10
Qwen3-32B	2:11
Qwen3-8B	0:56

Table 5: Wall-clock inference time for 500K records on 2 B200 GPUs using vLLM.

C Zero-Shot Per-Language Breakdown

Table 4 reports precision, recall, and macro-F_1 for each zero-shot model broken down by programming language on the validation set. The human-prediction bias of Qwen2.5-72B-Instruct and Qwen3-32B is consistent across all three languages, confirming it is a model-level property rather than a language-specific artifact. Qwen3-Next-80B-A3B-Instruct maintains the most balanced recall across languages, though recall drops notably for `C++` and `Java` compared to `Python`.

	Pred. Human (0)	Pred. AI (1)
Actual Human (0)	47,496	199
Actual AI (1)	402	51,903

Table 6: Validation confusion matrix for Qwen3-32B.

D Validation Error Analysis

We analyze the prediction errors of our lead Qwen3-32B checkpoint on the 100K validation split. The model reaches a macro-F_1 of 0.994, with 601 misclassified samples.

Confusion Matrix. A slight bias toward the human label can be observed in Table 6. However, because the overall separation is nearly perfect, this effect is too small to support a strong conclusion.

Per-language breakdown. Despite balanced sampling across (`language`, `label`) groups, `Python` retains a noticeably lower error rate than `C++` and `Java` (Table 7), likely reflecting broader Python coverage in Qwen3's pretraining.

Per-generator error rate. Table 8 reports the validation error rate for each AI source model. The hardest generators to detect are from the IBM Granite family, followed by `CodeLlama-34b-Instruct` and `deepseek-coder-6.7b-base`. The table suggests that *base* variants are often harder to detect

Lang.	N	Err.	Prec.	Rec.	F_1
Python	91,461	462	0.9949	0.9950	0.9949
C++	4,679	71	0.9845	0.9853	0.9848
Java	3,860	68	0.9821	0.9828	0.9824

Table 7: Per-language validation performance for fine-tuned Qwen3-32B. "Err." is the number of misclassified samples; precision, recall, and F_1 are macro-averaged.

Generator	N	Err. rate
Yi-Coder-9B-Chat	1,665	0.00%
Phi-3-mini-4k-instruct	1,667	0.00%
Phi-3-small-8k-instruct	1,814	0.06%
deepseek-coder-1.3b-base	1,113	0.09%
Phi-3-medium-4k-instruct	3,165	0.13%
deepseek-coder-1.3b-instruct	682	0.15%
starcoder2-7b	1,350	0.15%
Qwen2.5-Coder-1.5B-Instruct	1,903	0.16%
starcoder2-3b	1,760	0.17%
Yi-Coder-1.5B	1,732	0.17%
CodeLlama-70b-Instruct-hf	1,742	0.23%
Phi-3.5-mini-instruct	2,047	0.24%
Llama-3.2-1B	760	0.26%
Yi-Coder-1.5B-Chat	1,341	0.30%
Llama-3.2-3B	1,550	0.32%
deepseek-coder-6.7b-instruct	1,126	0.36%
Qwen2.5-Coder-7B-Instruct	1,324	0.38%
human	47,695	0.42%
Qwen2.5-Coder-32B-Instruct	1,597	0.50%
CodeLlama-7b-hf	1,128	0.53%
Qwen2.5-Coder-1.5B	1,306	0.54%
codegemma-2b	1,484	0.67%
Llama-3.1-8B	1,351	0.74%
Llama-3.1-8B-Instruct	1,653	0.85%
phi-2	1,966	0.97%
starcoder2-15b	1,872	1.07%
starcoder	1,857	1.24%
Qwen2.5-Coder-7B	1,427	1.33%
codegemma-7b	1,258	1.43%
Yi-Coder-9B	1,950	1.49%
Llama-3.3-70B-Instruct	1,756	1.59%
deepseek-coder-6.7b-base	1,207	2.15%
CodeLlama-34b-Instruct-hf	1,544	2.59%
granite-8b-code-instruct-4k	846	2.60%
granite-8b-code-base-4k	1,362	4.11%

Table 8: Per-generator validation error rate for fine-tuned Qwen3-32B.

than *instruct* variants, though this trend is not uniform. This may indicate that instruction tuning can produce more stylized, template-like outputs that are easier to separate from human code.