AICD Bench: A Challenging Benchmark for AI-Generated Code Detection

Daniil Orel[1], **Dilshod Azizov**[1], **Indraneil Paul**[2], **Yuxia Wang**[3],
Iryna Gurevych[1,2], **Preslav Nakov**[1]

[1]Mohamed bin Zayed University of Artificial Intelligence (MBZUAI), UAE
[2]Ubiquitous Knowledge Processing Lab (UKP Lab), Department of Computer Science,
TU Darmstadt and National Research Center for Applied Cybersecurity ATHENE, Germany
[3] INSAIT, Sofia University “St. Kliment Ohridski”, Bulgaria
{name.surname}@mbzuai.ac.ae; {name.surname}@tu-darmstadt.de

Abstract

Large language models (LLMs) are increasingly capable of generating functional source code, raising concerns about authorship, accountability, and security. While detecting AI-generated code is critical, existing datasets and benchmarks are narrow, typically limited to binary human-machine classification under in-distribution settings. To bridge this gap, we introduce *AICD Bench*, the most comprehensive benchmark for AI-generated code detection. It spans *2M examples*, *77 models* across *11 families*, and *9 programming languages*, including recent reasoning models. Beyond scale, AICD Bench introduces three realistic detection tasks: (*i*) *Robust Binary Classification* under distribution shifts in language and domain, (*ii*) *Model Family Attribution*, grouping generators by architectural lineage, and (*iii*) *Fine-Grained Human-Machine Classification* across human, machine, hybrid, and adversarial code. Extensive evaluation on neural and classical detectors shows that performance remains far below practical usability, particularly under distribution shift and for hybrid or adversarial code. We release AICD Bench as a *unified, challenging evaluation suite* to drive the next generation of robust approaches for AI-generated code detection. The data and the code are available at https://huggingface.co/AICD-bench.

1 Introduction

With the rapid advances of LLMs, the field of software engineering has also changed. Today, the generation of syntactically and semantically correct functional code can be done at scale. LLMs can not only solve challenging problems from platforms like LeetCode[1], but also write production-ready code, find and fix bugs, and write unit tests (Guo et al., 2024; Rozière et al., 2023; Yang et al., 2025; Tufano et al., 2020). Thus, the need to detect such code reliably has emerged as a critical challenge.

Detecting whether a piece of code was written by a human or generated by an LLM is no longer a niche problem and is central to ensuring academic integrity (Salim et al., 2024; Finnie-Ansley et al., 2023), preventing plagiarism (Park et al., 2025), and mitigating security risks (Tihanyi et al., 2024) associated with undetected synthetic code.

In response to this demand, research on AI-generated code detection has grown rapidly, with numerous methods proposed in recent years. However, this progress has been accompanied by fragmented evaluation: most studies introduce not only a new detection model, but also a dataset tailored to a specific experimental setting. Such datasets typically cover narrow configurations, for example, a small number of programming languages (*e.g.*, only Python, C++, and Java), a limited set of LLMs (*e.g.*, API-based models only), and non-reasoning generation patterns, resulting in highly domain-specific conclusions. While impressive performance is often reported under these controlled conditions, such results provide limited insight into real-world generalization. The core limitation is *the absence of a unified, comprehensive benchmark* capable of evaluating detection methods across multiple dimensions of variation.

Existing evaluations predominantly report in-domain accuracy, neglecting out-of-distribution (OOD) generalization, which is a cornerstone of real-world deployability. Even when OOD evaluation is attempted, it typically considers only one dimension at a time: either changing the programming language (*e.g.*, from Python to Java) or changing the purpose of the code (*e.g.*, from algorithmic problem solving to real-world deployable code) or varying the generator model, but rarely both simultaneously. Such conditions yield optimistic results and do not reflect *realistic deployment*, where detectors must handle unseen models, programming languages, domains, and code written in an adversarial way.

[1] https://leetcode.com/

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 1: Long Papers, pages 6913–6938
March 24-29, 2026 ©2026 Association for Computational Linguistics

To overcome these challenges, we present *AICD Bench*, a *new large-scale benchmark* for AI-generated code detection. Unlike prior resources, *AICD Bench* systematically evaluates detectors across multiple axes of variation, including generator families, programming languages, and adversarial strategies. It introduces *three tasks* designed to capture increasingly realistic challenges: (*i*) *Robust Binary Classification* under language and domain shifts, (*ii*) *Model Family Attribution*, grouping generators by architectural lineage, and (*iii*) *Fine-Grained Human-Machine Classification*, distinguishing human, machine, hybrid and adversarial code.

Our contributions are as follows:

1. We release *AICD Bench* a comprehensive 2M sample benchmark that spans 77 generators and 9 programming languages, subsuming previous datasets, and provide standardized splits, protocols, and evaluation scripts to enable reproducible research.

2. We define novel tasks that go beyond binary classification, including model-family attribution and fine-grained classification.

3. We conduct extensive evaluations of classical and deep learning-based baselines, showing that current methods generalize poorly in out-of-distribution settings.

2 Related Work

2.1 Authorship Attribution

With the widespread use of LLMs, recognizing the author of a piece of work has become an important problem. It can be viewed from multiple perspectives: as a multiway classification problem, where each output has to be paired with an author from a candidate list (which could include LLMs), or as a binary classification problem, e.g., "*Is it written by a human or an LLM?*", "*Do these outputs share the same author?*", etc. (Uchendu et al., 2020). Previous research in both text and code domains has focused on the first perspective. For example, Wang et al. (2024a) and Orel et al. (2025a) address authorship attribution as a multi-class classification problem, where each class corresponds to a specific LLM. In addition to supervised approaches, unsupervised methods have been proposed for authorship identification, including Uniform Information Density (Venkatraman et al., 2024) and stylistic representation learning (Soto et al., 2024).

However, most existing methods aim to identify the signature of an individual model. As the number of deployed LLMs continues to grow, fine-grained attribution at the level of individual models becomes increasingly impractical. A more scalable alternative is attribution at the level of *model families*, defined as groups of related models that share a common architecture and training philosophy but differ in size, capabilities, or specialization. In *AICD Bench*, we introduce a dedicated task that targets attribution of a given code sample to its underlying model family.

2.2 Benchmarks for AI-Generated Code Detection

AI-generated code detection largely follows the trajectory of AI-generated text detection, where progress has been driven by benchmarks, such as RAID (Dugan et al., 2024), MULTITUDE (Macko et al., 2023), and MAGE (Li et al., 2024), and robust detection models developed through shared tasks (Wang et al., 2025). Early work on code detection relied on small-scale benchmarks: Nguyen et al. (2024) introduced a 7K-sample dataset and trained a CodeBERT-based classifier (Feng et al., 2020), marking one of the first systematic approaches. More recently, Xu et al. (2025) proposed a benchmark of 1.1M Java and Python code samples (550K human–AI pairs) and trained UnixCoder (Guo et al., 2022) with a contrastive objective, achieving improved detection performance.

A further step toward increased diversity was taken by Orel et al. (2025a), who released a dataset of nearly 500K code samples across C++, Java, and Python generated by five compact Code-LMs. Beyond expanding language and model coverage, they introduced a hybrid authorship identification task to distinguish human-written, machine-generated, and human–machine co-authored code. This partially addressed the need for evaluations beyond binary distinctions. More recently, the `Droid` framework (Orel et al., 2025b) extended this line of work by proposing a detection pipeline and introducing an adversarial setting in which LLMs are trained to evade detection, increasing realism and difficulty.

Despite these advances, existing resources fall short of constituting a benchmark, as they lack standardized tasks, splits, and evaluation protocols. The prominent exception is CodeMirage (Guo et al., 2025), which presents a dataset of 210K samples spanning 10 programming languages and generated by ten code LLMs.

CodeMirage evaluates ten diverse detectors, including zero-shot, pre-trained, fine-tuned, and embedding-based approaches, making it the only AI-generated code detection benchmark to date. Nevertheless, it focuses exclusively on a single, simplified task: binary classification of human-written versus AI-generated code. As reported in their results, fine-tuned models such as CodeT5+ achieve F1-scores exceeding 80% on unseen generators and paraphrased inputs, suggesting that the benchmark may already be saturated.

This situation exposes a critical gap: the lack of benchmarks that systematically evaluate detection methods across multiple dimensions of complexity, including cross-model, cross-language, cross-domain, and hybrid or adversarial generation settings, while reflecting realistic deployment scenarios. Addressing this gap is the primary motivation behind *AICD Bench*.

3 Motivation Behind the Task Design

The task design of *AICD Bench* is guided by two complementary goals: faithfully capturing the practical challenges of AI-generated code detection and addressing key limitations of existing benchmarks. As discussed in Section 2, prior datasets and evaluations have largely centered on binary classification: human-written vs. machine-generated code. While this task is fundamental, it fails to capture the increasingly diverse, hybrid, and adversarial ways in which LLMs are used in practice. In order to bridge this gap, we deliberately design a set of three tasks that progressively increase in complexity and realism.

Task 1: Robust Binary Classification. Binary detection (human-written vs. machine-generated) remains a core requirement for applications such as academic integrity enforcement, intellectual property protection, and software security. However, most existing evaluations have focused on in-distribution settings, which can substantially overestimate real-world performance. In practice, detectors must generalize to unseen programming languages and application domains, where stylistic conventions and structural patterns differ markedly. To reflect this reality, Task 1 explicitly partitions evaluation into progressively more challenging out-of-distribution splits. This design enables a systematic assessment of robustness under realistic distribution shifts, yielding a more stringent and informative evaluation than prior benchmarks.

Task 2: Model Family Attribution. Determining whether code is AI-generated is only a first step; understanding *which kind* of model produced it is essential for applications such as corporate intelligence and intellectual property protection. Rather than attributing outputs to individual models, which has become increasingly impractical as the number of code-generating LLMs grows, Task 2 focuses on *model family attribution*. By grouping generators according to shared architectures, training regimes, and design principles, this task provides actionable attribution that balances scalability with informative stylistic characterization.

Task 3: Fine-Grained Human–Machine Classification. A simple binary distinction between human- and machine-generated code overlooks the increasingly hybrid and adversarial nature of LLM-assisted development. In practice, developers frequently co-author code with LLMs, while alignment and adversarial training regimes (*e.g.*, RLHF) deliberately shape model outputs to resemble human-written code. Consequently, a realistic benchmark must distinguish between *fully human-written*, *fully AI-generated*, *hybrid*, and *adversarial* code. Task 3 addresses this need by framing detection as a multi-way classification problem that mirrors real-world deployment scenarios, in which detectors must operate reliably under ambiguity and intentional obfuscation.

4 Data

Our work builds upon `Droid` (Orel et al., 2025b), which covers 7 programming languages, 43 LLMs, and nearly 1M code samples, making it one of the largest existing resources for AI-generated code detection. While comprehensive, `DroidCollection` lacks recent reasoning-oriented models and does not cover several widely used programming languages. To address this, we extended the dataset by adding two additional languages (PHP and Rust) and incorporating a diverse set of new LLMs at a scale comparable to the original collection, explicitly including reasoning models. All newly generated samples follow the original `Droid` generation protocol to ensure consistency. We generated 500K samples using inverse-instruction prompting based on *StarCoderData* (Li et al., 2023) and *CodeNet* (Puri et al., 2021), and performed docstring- and comment-conditioned generation using data from *The Vault* (Nguyen et al., 2023) to increase diversity and coverage.

Task	Split	# Samples	Languages
Task 1	Train	500K	Python, Java, C++
	Validation	100K	Python, Java, C++
	Test	1M	Python, Java, C++, C, Golang, PHP, C#, JavaScript
Task 2	Train	500K	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust
	Validation	100K	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust
	Test	500K	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust
Task 3	Train	900K	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust
	Validation	200K	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust
	Test	1M	Python, Java, C++, C, Golang, PHP, C#, JavaScript, Rust

Table 1: *AICD Bench*: Dataset statistics across tasks, splits, sample counts, and programming languages.

AICD Bench further incorporates human-written and hybrid samples from sources that were not included in DroidCollection. Specifically, we add 50K human-written samples in C++, C, C#, Java, JavaScript, PHP, Go, and Rust from *The Heap* (Katzy et al., 2025). *The Heap* is released as a contamination-free benchmark and does not overlap with *The Stack* (Kocetkov et al., 2023), which underlies datasets such as *The Vault* and *StarCoder-Data* used in DroidCollection.

We further augmented the *AICD Bench* dataset with 100K hybrid examples from *Swallow Code* (Fujii et al., 2025), which contains high-quality programs that were automatically generated by rewriting human-authored code using the LLaMA 3.3 70B model. The distribution of dataset splits is summarized in Table 1.

A comparison of DroidCollection (Orel et al., 2025b), *CodeMirage* (Guo et al., 2025), and our dataset is shown in Table 2. We can see that overall, *AICD Bench* substantially improves both dataset scale and generator diversity, while also expanding coverage to additional programming languages and reasoning-capable models. A complete list of generator models is provided in Appendix B.

Criteria	DroidCollection	*CodeMirage*	*AICD Bench*
# samples	1.06M	210K	**2.05M**
# programming languages	7	**10**	9
# models	43	10	**77**[2]
# model families	10	6	**11**
Reasoners	✗	✓	✓

Table 2: Comparison of DroidCollection, *CodeMirage*, and *AICD Bench* in terms of size, languages, models, and model families.

Parameter	Min	Max
AST depth	2	31
Maximum line length (characters)	12	400
Average line length (characters)	5	140
Number of lines of code	6	300
Fraction of alphanumeric characters	0.2	0.75
Docstring English confidence (%)	99	100

Table 3: Filtering parameters used during the construction of *AICD Bench*.

4.1 Data Filtering and Quality Assurance

We replicate Droid's filtering pipeline to maintain consistency in data distribution and to prevent detectors from exploiting distributional artifacts rather than learning meaningful code patterns. The filtering process removes unparsable code, overly simple or excessively complex samples, non-code and auto-generated files, and non-English content, resulting in a clean and representative dataset. We filtered the samples according to the criteria listed in Table 3, which match those used in the construction of DroidCollection and works on Code-LM training (Paul et al., 2025; Lozhkov et al., 2024; Li et al., 2023; Shi et al., 2025). The resulting distributions (Appendix C) indicate that our dataset is well structured, appropriately complex, and comparable to real-world codebases (Herraiz et al., 2011; Lopes et al., 2017; Kolassa et al., 2013).

We performed de-duplication using the MinHash (Broder, 1997) algorithm with a similarity threshold of 0.8, applied jointly to the original and newly generated samples. This process ensures that no duplicates remain, even when samples overlap with those from Droid, thereby guaranteeing clean and uncontaminated data.

4.2 Robust Binary Classification Data Subset

Task 1 focuses on training binary detectors (human vs. machine) that generalize across *unseen programming languages* and *unseen domains*. The test set is divided into four progressively more challenging splits: (*i*) seen languages (C++, Java, Python) and a seen domain (algorithmic code), (*ii*) unseen languages (Golang, PHP, Rust, JavaScript, C#, C) with a seen domain, (*iii*) seen languages with unseen domains (research code and general-purpose software), and (*iv*) fully unseen languages and domains. This structure enables a systematic evaluation of robustness under increasing distributional shift.

[2] 82 models if distinct quantizations are counted separately.

4.3 Model Family Attribution Data Subset

We define the following families, together with representative generators:

- **DeepSeek-AI**: DeepSeek-V3 (DeepSeek-AI et al., 2024), DeepSeek-R1 (DeepSeek-AI et al., 2025), and DeepSeek-Coder models (Guo et al., 2024).
- **Qwen**: Qwen3 (Yang et al., 2025) and Qwen2.5 (Yang et al., 2024).
- **01-ai**: Yi-Coder models (Young et al., 2024).
- **BigCode**: StarCoder (Li et al., 2023) and StarCoder2 (Lozhkov et al., 2024).
- **Gemma**: Gemma3 (Kamath et al., 2025) and CodeGemma (Zhao et al., 2024).
- **Phi**: Phi4 (Abdin et al., 2025) and Phi3 (Abdin et al., 2024).
- **Meta-Llama**: Llama (Dubey et al., 2024) and CodeLlama (Rozière et al., 2023).
- **IBM-Granite**: Granite Code (Mishra et al., 2024) and Granite (Karlinsky et al., 2025).
- **Mistral**: Devstral[3], Mixtral (Jiang et al., 2024), and Mistral (Jiang et al., 2023).
- **OpenAI**: GPT-4o and GPT-4o-mini.
- **Gemini**: Gemini 1.5 (Reid et al., 2024) and Gemini 2.5 (Comanici et al., 2025).

We perform evaluation under two conditions: *seen authors* (generators observed during training) and *unseen authors* (previously unseen generators from known families). With 77 generators and an average of six models per family, this task requires fine-grained discrimination both within and across model lineages.

4.4 Fine-Grained Human–Machine Classification Data Subset

Task 3 distinguishes 4 categories of code: *human-written*, *machine-generated*, *hybrid* (human-authored code rewritten or completed by an LLM), and *adversarial*. The latter includes code produced using prompts or alignment strategies designed to elicit human-like outputs, including DPO-based (Rafailov et al., 2023) fine-tuning on paired human and LLM-generated solutions.

The test set contains 1M examples, with half drawn from the same sources as the training data (in-domain evaluation) and the other half drawn from *The Heap* and *Swallow Code* (out-of-domain evaluation), enabling a direct assessment of generalization under domain shift.

5 Experiments and Results

5.1 Experimental Setup

To assess the utility of *AICD Bench*, we train a suite of encoder-based classifiers on its three tasks. Unless stated otherwise, all models are trained for 3 epochs with a batch size of 64 and an input window of 512 tokens, following the configuration that yields the best performance in `Droid`. Importantly, our goal is not to obtain state-of-the-art results; rather, these experiments aim to demonstrate the benchmark's applicability and to confirm that it provides a meaningful and reliable evaluation.

We experimented with the following encoders: CodeBERT (Feng et al., 2020), CodeT5+ (Wang et al., 2023), UnixCoder (Guo et al., 2022), ModernBERT (Warner et al., 2025), RoBERTa (Liu et al., 2019), and DeBERTa-v3 (hereafter, DeBERTa) (He et al., 2023). We selected them due to their widespread use in AI-generated content detection: CodeBERT is used by Nguyen et al. (2024), CodeT5+ and UnixCoder by Xu et al. (2025) and Orel et al. (2025a), ModernBERT by Orel et al. (2025b), and RoBERTa by Wang et al. (2024b).

We use Macro-F1 as the primary evaluation metric. Since the datasets and the tasks are class-imbalanced, we focus on reliable performance across all classes rather than optimizing for the majority label. Macro-F1 computes the F1 score independently for each class and then averages across classes, ensuring that minority-class performance contributes equally to the overall score. Compared to accuracy (often dominated by frequent classes) and Micro-F1 (which aggregates over all instances and thus favors majority classes), Macro-F1 provides a fairer and more informative assessment in our setting.

In addition to encoder-based models, we train classical baselines, including Logistic Regression, SVM, and CatBoost (Prokhorenkova et al., 2018). For each, we evaluate three feature representations: (*i*) TF-IDF, (*ii*) AST-based features, which have been shown to be useful for AI-generated code detection (Orel et al., 2025a,b; Idialu et al., 2024), and (*iii*) their combination.

[3] Devstral Model

For TF-IDF, we use the full source text and tokenize it using whitespace delimiters, i.e., any sequence of spaces, tabs, or line breaks, without applying lowercasing, stop-word removal, or punctuation/identifier normalization. From this token stream, we construct uni-, bi-, and tri-gram vocabularies and compute TF-IDF features over the training corpus, starting from 5,000 n-gram features and reducing dimensionality to 500 using truncated singular value decomposition (SVD; Zhang, 2015). For AST-based features, we parse each program using Tree-Sitter[4] and extract structural signals, including overall AST depth, counts of node/construct types (*e.g.*, numbers of `if` statements, loops, and function definitions), layout/style indicators (*e.g.*, empty-line density), and code-complexity proxies. Feature dimensionality is data-driven and equals the number of distinct AST-derived attributes observed in training, yielding 543 features for the Robust Binary Classification task and 1,030 for the remaining tasks (which involve more programming languages).

We also run zero-shot Fast-DetectGPT (Bao et al., 2024), a faster version of DetectGPT (Mitchell et al., 2023) that uses human-machine differences in token-choice likelihood and has shown strong performance detecting AI text and code (Orel et al., 2025a). Since it is best suited to binary decisions, we evaluate it on Task 1 and on a binarized version of Task 3 (human vs. all other classes). Finally, we evaluate LLM performance on proposed tasks; see Section 5.2.4 for details.

5.2 Evaluation Results

Table 4 shows that deep learning models (CodeBERT, CodeT5+, ModernBERT, RoBERTa, UniXCoder, and DeBERTa) outperform classical approaches. This suggests that neural encoders' contextual representations provide more informative signals for identifying AI-generated code than structural (AST-based) or statistical (TF-IDF-based) features. Among neural models, ModernBERT achieves the strongest performance on Tasks 2 and 3, consistent with findings reported by Orel et al. (2025b). Zero-shot Fast-DetectGPT outperforms trained models in Task 1 but still falls short of the random baseline, highlighting the difficulty of robust binary detection under distribution shift.

Model	Task 1	Task 2	Task 3
Random[5]	45.73	5.69	20.34
Majority	43.83	5.43	20.05
Fast-DetectGPT	44.99	-	50.03[6]
CB$_{\text{TF-IDF}}$	29.40	5.47	22.45
CB$_{\text{AST}}$	18.02	7.78	14.59
CB$_{\text{AST \& TF-IDF}}$	18.02	0.12	3.09
SVM$_{\text{TF-IDF}}$	**43.05**	5.44	21.28
SVM$_{\text{AST}}$	18.02	5.99	3.43
SVM$_{\text{AST \& TF-IDF}}$	18.02	1.38	5.82
LR$_{\text{TF-IDF}}$	37.10	5.45	20.64
LR$_{\text{AST}}$	18.02	5.44	3.18
LR$_{\text{AST \& TF-IDF}}$	18.02	1.09	5.23
CodeBERT	28.64	23.71	55.60
CodeT5+	28.08	5.43	52.76
ModernBERT	30.61	**32.84**	**61.65**
RoBERTa	31.88	17.52	52.05
UnixCoder	25.51	26.64	54.21
DeBERTa	34.13	12.08	54.65

Table 4: Macro F1-score of baselines across all tasks. Best model performance per task is **bolded**, while dummy baselines (random or majority) are underlined where they outperform trained models. (Logistic Regression - LR; CatBoost - CB. Task 1 corresponds to the Robust Binary Classification task, Task 2 corresponds to the Model Family Attribution task, and Task 3 corresponds to the Fine-Grained Human-Machine Classification task.)

On Task 3, even when evaluated in a binarized setting, Fast-DetectGPT lags behind transformer-based detectors, indicating that hybrid and adversarial samples diverge substantially from what this detector characterizes as machine-generated.

5.2.1 Task 1: Robust Binary Classification

In this task, classical models achieve unexpectedly strong performance (see Table 4): SVM and Logistic Regression with TF-IDF features outperform all deep learning models. Because the task primarily probes out-of-distribution generalization, this behavior is consistent with classical learning theory, where increased model complexity can exacerbate overfitting to spurious correlations (Rohlfs, 2025). At the same time, both random and majority-class baselines achieve even higher scores, highlighting a severe train–test distribution shift that limits reliable generalization.

Further analysis indicates that language-independent cues, such as variable naming patterns, play a central role in distinguishing human-written from AI-generated code (see Appendix D.1), which helps explain why TF-IDF features are particularly effective in this setting.

[4] https://tree-sitter.github.io/tree-sitter/

[5] Scores differ from 1/#classes due to class imbalance.

[6] Binarized.

Setting	Macro F1	STD (±)
Seen domain, seen language	0.63	0.30
Seen domain, unseen language	0.42	0.12
Unseen domain, unseen language	0.21	0.07
Unseen domain, seen language	0.20	0.08

Table 5: **Task 1 (Robust Binary Classification):** experiments across different domain/language settings. We report Macro F1, averaged over detectors.

We further analyze model performance across four input scenarios: (*i*) seen language and seen domain, (*ii*) unseen language in a seen domain, (*iii*) seen language in an unseen domain, and (*iv*) unseen language and an unseen domain. As shown in Table 5, domain shift emerges as the dominant source of error: models perform nearly as poorly when only the domain is unseen as when both the domain and the language are unseen. This behavior is expected, as coding style and conventions vary substantially across domains; for instance, research code often contains extensive explanatory comments, whereas algorithmic problem solutions are typically concise and minimally annotated.

As shown in Appendix D, the AST features decrease the performance of the model when paired with TF-IDF. We believe that it is reasonable, since the structure of the unseen data is very different from the training data, so the AST features cannot be used as a robust representation. Statistical features from TF-IDF, at the same time, can handle not only the syntactical, but also some stylistical (naming conventions, etc.) properties of the code, which can be used to identify LLM-generated code.

5.2.2 Task 2: Model Family Attribution

As shown in Tables 4 and 6, this task is the most challenging among all three tasks, exhibiting the lowest peak and average performance across models. This difficulty is largely attributable to the increased number of classes: while Robust Binary Classification involves only two labels and Fine-Grained Human-Machine Classification involves four, model family attribution requires discrimination among 12 classes.

A pronounced performance gap is also observed between in-domain and out-of-domain generators. As shown in Table 6, attributing previously unseen generators to their corresponding families proves particularly difficult, suggesting substantial variation within model families and limited transferability of learned stylistic signals.

Category	Macro F1	STD (±)
In-domain Generator	0.149	0.154
Out-of-Domain Generator	0.046	0.070

Table 6: **Task 2 (Model Family Attribution):** average Macro-F1 across models by generator type.

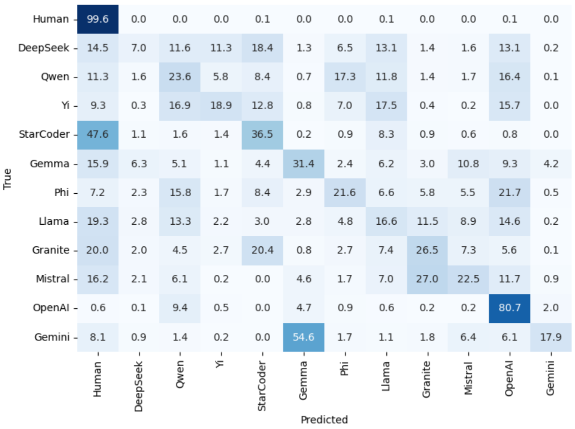


Figure 1: **Task 2 (Model Family Attribution):** ModernBERT confusion matrix. The values are row-normalized percentages, showing the proportion of each true class assigned to each predicted class.

Moreover, the standard deviation of performance exceeds the mean, which together with Figure 6 indicates high variability and instability across classification models.

Figure 1 further illustrates these challenges. The best-performing model, ModernBERT, almost perfectly identifies human-written code and OpenAI-generated samples, which can be attributed to the low intra-family variability of these classes: they comprise only one (human) and two (GPT-4o and GPT-4o-mini) generators, respectively. In contrast, Gemini samples are frequently misclassified as Gemma, likely reflecting shared design and training characteristics, as both model families are developed by Google. We also observe that StarCoder outputs are often misclassified as human-written code, which is consistent with the fact that the StarCoder family includes multiple base models and is trained predominantly on large volumes of open-source human-authored code. This training regime may lead to generations that closely resemble natural coding patterns in terms of structure, naming conventions, and style. A detailed, model-wise analysis for this task is provided in Appendix E.

Category	Macro F1	STD (±)
In-domain	0.366	0.302
Out-of-Domain	0.119	0.094

Table 7: **Task 3 (Fine-Grained Human-Machine Classification)**: average Macro-F1 across classifiers by generator type with std.

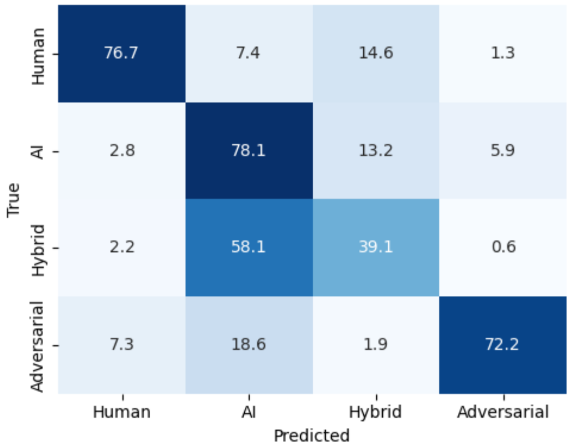


Figure 2: **Task 3 (Fine-Grained Human-Machine Classification)**: ModernBERT confusion matrix. The values are row-normalized percentages, showing the proportion of each true class assigned to each predicted class.

5.2.3 Task 3: Fine-Grained Human-Machine Classification

As shown in Tables 4 and 7, in this multi-class setting, deep learning models consistently outperform classical approaches. Among the classical baselines, the strongest results are obtained using TF-IDF features alone, which we attribute to the sparsity of the AST-based representation: out of 1,030 AST features, only 75% are non-zero. This sparsity limits the effectiveness of structural cues in distinguishing subtle authorship differences across fine-grained classes.

A substantial performance gap emerges when comparing in-domain and out-of-domain evaluation, particularly for samples drawn from *Swallow Code* and *The Heap*. This gap indicates that models struggle to generalize across data distributions that differ from those observed during training (see Table 7). Notably, this degradation persists even for strong encoder models, suggesting that fine-grained detection is especially sensitive to domain shift. A more detailed dataset-level analysis is provided in Appendix F for completeness and further insights.

Model	Task 1	Task 2	Task 3
Gemini$_{zs}$	57.15	**5.43**	25.95
Gemma$_{zs}$	58.45	5.37	22.21
Gemini$_{cot}$	**62.31**	5.54	**28.14**
Gemma$_{cot}$	54.94	5.37	22.23

Table 8: **Zero-shot LLM experiments across all three tasks:** Macro F1-scores of LLMs with simple zero-shot prompting (zs) and chain-of-thought prompting (cot).

An examination of the predictions from the best-performing model, ModernBERT, reveals that most errors occur on out-of-domain examples, especially *Swallow Code* instances labeled as *hybrid*. These examples are predominantly misclassified as fully AI-generated (see Figure 2). In addition, adversarial examples are frequently misclassified as either AI-generated or human-written, which is expected given that such code is deliberately crafted to mimic human authorship.

Overall, although this task is comparatively easier than the other two, as is reflected in generally higher scores across models, the best achieved performance (a Macro-F1 of 61.65) remains well below practical requirements. This underscores the continued difficulty of fine-grained human–machine discrimination and highlights the challenge posed by *AICD Bench*.

5.2.4 Zero-Shot LLM Experiments

Next, we evaluate the ability of large language models to perform the tasks defined in *AICD Bench*, in a zero-shot setting, using two representative LLMs: Gemini-2.5-Flash and Gemma3-27B. To ensure cost-efficient yet representative evaluation, we sample 5% of the data per task, stratified by label, domain, and programming language. We consider two evaluation settings: zero-shot classification and classification with chain-of-thought (CoT) prompting. The prompts used for both settings are provided in Appendix C.1.

As shown in Table 8, zero-shot CoT prompting consistently improves the performance across tasks. In particular, Gemini with CoT prompting achieves the highest score on Task 1 among all evaluated models. However, Task 2 remains highly challenging for LLMs, and the performance on Task 3 is substantially lower than that of trained detection models.

5.3 Error Analysis

We analyze errors by examining the cases where all evaluated models fail; representative examples are shown in Appendix G.

For Task 1, we observe a clear asymmetric failure pattern: all models misclassify certain human-written samples as AI-generated, while no AI-generated samples are misclassified by all models. This suggests that detectors effectively identify AI-generated code, likely due to shared stylistic artifacts, but struggle with the broader syntactic, structural, and stylistic diversity of human-written code. Manual inspection reveals no consistent LLM-like traits (e.g., boilerplate, verbose comments, repetitive patterns, or unusual naming), indicating that these are genuine human examples flagged because they fall outside the "typical" human patterns seen during training.

For Task 2, we observe no consistent error patterns across models. In contrast, for Task 3, all models fail on hybrid and human-written samples from *Swallow Code* and *The Heap*, which are absent from the training data. This further highlights the strong sensitivity of detectors to domain shifts, particularly in fine-grained classification settings.

We also apply SHAP (Lundberg and Lee, 2017) to analyze which input features most strongly influence the model predictions; illustrative examples are shown in Appendix H. For Task 1, deep models exhibit a pronounced bias toward competitive-programming artifacts: surface-level templates (e.g., `#include`, `#define`, `class Solution`, loop scaffolding, and typed signatures) are overemphasized relative to semantic content, while explanatory or demonstration tokens (e.g., `the answer would`, `TreeNode`, `sum`, `summary`, `//`) often trigger incorrect AI predictions under domain shift.

In contrast, for Task 3 especially for hybrid samples the most informative cues arise from *mixed contexts*, such as non-algorithmic headers interleaved with code, shell/URL/path fragments near code tokens, and documentation-like text embedded within code (docstrings, small unit tests, usage examples, repeated imports, and high-level task verbs such as `deploy`, `retrieve`, or `server`).

Features that push predictions away from the hybrid class include runnable scaffolding (e.g., `if __name__ == __main__`), dense CLI/IO patterns, explicit error handling (`try/except`), and human-written license headers. Overall, codestyle mixing is the dominant signal for identifying hybrid code.

6 Conclusion and Future Work

We presented *AICD Bench*, the largest and most comprehensive benchmark for AI-generated code detection to date. It comprises 2M code samples from 77 generators, including strong reasoning-oriented models, across 9 programming languages, and introduces three complementary tasks: robust binary classification under distribution shifts, model family attribution, and fine-grained human–machine classification. By jointly emphasizing scale, diversity, and realistic evaluation settings, *AICD Bench* substantially extends the scope of prior benchmarks.

Extensive evaluations with both neural and classical baselines demonstrate that these tasks remain far from solved. Current detectors struggle to generalize across programming languages, domains, and generator families, and perform particularly poorly on hybrid and adversarial code. While ModernBERT achieves the strongest overall performance, leading on two of the three tasks, even the best-performing models fall well short of practical requirements. Notably, a simple SVM outperforms deep learning models on robust binary classification, yet still performs below random guessing, highlighting severe generalization challenges and the urgent need for new detection paradigms.

Beyond exposing these limitations, *AICD Bench* provides a standardized and extensible evaluation framework with unified tasks, splits, and protocols, enabling reproducible comparison and systematic progress. By moving beyond oversimplified in-distribution binary detection, it reorients the field toward realistic, deployment-driven challenges.

In future work, we plan to investigate adversarial and domain-adaptive training strategies aimed at improving robustness under distribution shift, such as adversarial data augmentation, domain-invariant representation learning, and curriculum-based adaptation across programming languages and domains. We also intend to develop meta-models that explicitly promote generalization across languages, domains, and generator families, for example by learning shared detection priors or dynamically combining task-specific detectors. Finally, we aim to build an automated pipeline to continuously expand the benchmark with newly released LLMs, emerging programming languages, and diverse sources of human-written code, ensuring that *AICD Bench* remains representative of evolving real-world coding practices.

Limitations

Potential Data Contamination. As with many public benchmarks, there is a risk that *AICD Bench* may become saturated over time as models are increasingly tuned to its specific distributions. This is particularly problematic given that the benchmark is intended to assess out-of-distribution robustness. To mitigate this risk, we plan to introduce a private evaluation split with hidden labels, where submissions will be evaluated through an online platform rather than via local testing.

Reliance on DroidCollection. Our benchmark builds substantially on `DroidCollection`. Despite incorporating additional models, two supplementary datasets, and new programming languages, it remains influenced by the underlying distribution of that resource. Consequently, (*i*) none of the `Droid` generators can be directly evaluated on *AICD Bench*, and (*ii*) the observed performance trends may still reflect distributional skews inherited from `DroidCollection` rather than those of AI-generated and human-written code encountered in real-world settings. We plan to address this limitation by iteratively updating the benchmark with newer and more diverse data sources, thereby reducing reliance on any single dataset.

Constraints on Code Diversity. Although *AICD Bench* aims to approximate real-world coding scenarios, its coverage remains constrained along two dimensions. First, the dataset is filtered using parameters such as AST depth, line length, and code size, which ensures clean samples but excludes highly verbose, overly complex, extremely short, or poorly structured code commonly found in practice. As a result, the benchmark may not fully capture the syntactic, structural, and stylistic variability of human-written code. Second, the benchmark currently spans nine widely used programming languages (C++, C, C#, Go, Java, JavaScript, PHP, Python, and Rust), which, while representative of many real-world applications, form a finite set and do not test generalization beyond this scope.

To mitigate these limitations, future iterations of *AICD Bench* will (*i*) relax filtering thresholds to include more atypical and low-quality human-written code, and (*ii*) expand language coverage to less mainstream or domain-specific languages (e.g., Swift, Kotlin, MATLAB), enabling evaluation under broader and more challenging distributional shifts.

Ethical Statement

Data Collection and Privacy. *AICD Bench* is constructed from publicly available research corpora and model outputs generated via documented interfaces. We comply with platform terms of service and respect upstream licenses. Human-written code is sourced from published research datasets; no private repositories, paywalled content, or sensitive personal identifiers are included. Where required, we preserve original attributions and sufficient metadata for license compliance.

Our work aims to promote transparency in AI-assisted coding and support applications such as plagiarism detection, compliance, and provenance auditing. To mitigate misuse risks (*e.g.*, detector evasion), we do not release exploit-oriented prompt details and clearly document the limitations of current detectors.

Bias. Both human- and LLM-authored code may reflect biases arising from data availability, platform popularity, community conventions, and training corpora. As a result, *AICD Bench* may inherit distributional skews (*e.g.*, language, domain, or style imbalances) that affect external validity. We mitigate these risks through diverse sampling across platforms, languages, and generator families, though perfect representativeness cannot be guaranteed. Future releases will include more detailed bias analyses and refined sampling strategies.

Risk of Misuse. Although *AICD Bench* strengthens detection research, it could also be misused to develop evasion strategies. To reduce this risk, we avoid releasing adversarial prompts and emphasize that the benchmark is for research use only.

Broader Impact. By framing detection as a multitask, realism-driven benchmark, we aim to move the field beyond oversimplified binary detection toward practical robustness. Despite its limitations, *AICD Bench* provides a foundation for more transparent, accountable, and trustworthy AI-assisted programming.

Acknowledgments

At TU Darmstadt, this research has been supported by the German Federal Ministry of Research, Technology and Space and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

Marah I Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat S. Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, Piero Kauffmann, Yash Lara, Caio César Teodoro Mendes, Arindam Mitra, Besmira Nushi, Dimitris Papailiopoulos, Olli Saarikivi, Shital Shah, Vaishnavi Shrivastava, and 4 others. 2025. Phi-4-Reasoning technical report. *ArXiv preprint*, abs/2504.21318.

Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat S. Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, and 68 others. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *ArXiv preprint*, abs/2404.14219.

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 *Proceedings of the International Conference on Learning Representations*, ICLR '24, Vienna, Austria.

Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES 1997*, pages 21–29, Salerno, Italy.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 81 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *ArXiv preprint*, abs/2507.06261.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 81 others. 2025. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv preprint*, abs/2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 80 others. 2024. Deepseek-V3 technical report. *ArXiv preprint*, abs/2412.19437.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, and 82 others. 2024. The Llama 3 herd of models. *ArXiv preprint*, abs/2407.21783.

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*, ACL '24, pages 12463–12492, Bangkok, Thailand.

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 Conference on Empirical Methods in Natural Language Processing*, EMNLP '20, pages 1536–1547, Online. Association for Computational Linguistics.

James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI wants to know if this will be on the exam: Testing OpenAI's Codex on CS2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*, ACE '23, page 97–104, Melbourne, VIC, Australia. Association for Computing Machinery.

Kazuki Fujii, Yukito Tajima, Sakae Mizuki, Hinari Shimada, Taihei Shiotani, Koshiro Saito, Masanari Ohi, Masaki Kawamura, Taishi Nakamura, Takumi Okamoto, Shigeki Ishida, Kakeru Hattori, Youmi Ma, Hiroya Takamura, Rio Yokota, and Naoaki Okazaki. 2025. Rewriting pre-training data boosts LLM performance in math and code. *ArXiv preprint*, abs/2505.02881.

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*, ACL '22, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the large language model meets programming - the rise of code intelligence. *ArXiv preprint*, abs/2401.14196.

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. *ArXiv preprint*, arXiv:2506.11059.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2023. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In *Proceedings of the International Conference on Learning Representations*, ICLR '23, Kigali, Rwanda. OpenReview.net.

Israel Herraiz, Daniel Germán, and Ahmed E. Hassan. 2011. On the distribution of source code file sizes. In *Proceedings of the 6th International Conference on Software and Database Technologies*, volume 2 of *ICSOFT '11*, pages 5–14, Seville, Spain.

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, Lisbon, Portugal. Association for Computing Machinery.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *ArXiv preprint*, abs/2310.06825.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, and 7 others. 2024. Mixtral of Experts. *ArXiv preprint*, abs/2401.04088.

Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean-Bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, and 191 others. 2025. Gemma 3 technical report. *ArXiv preprint*, abs/2503.19786.

Leonid Karlinsky, Assaf Arbelle, Abraham Daniels, Ahmed Nassar, Amit Alfassi, Bo Wu, Eli Schwartz, Dhiraj Joshi, Jovana Kondic, Nimrod Shabtay, Pengyuan Li, Roei Herzig, Shafiq Abedin, Shaked Perek, Sivan Harary, Udi Barzelay, Adi Raz Goldfarb, Aude Oliva, Ben Wieles, and 43 others. 2025. Granite Vision: A lightweight, open-source multimodal model for enterprise intelligence. *ArXiv preprint*, abs/2502.09927.

Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. 2025. The Heap: A contamination-free multilingual code dataset for evaluating large language models. In *IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering*, Forge@ICSE '25, pages 151–155, Ottawa, ON, Canada. IEEE.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Carsten Kolassa, Dirk Riehle, and Michel Lind. 2013. A model of the commit size distribution of open source. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '13, pages 52–66, Špindlerův Mlýn, Czech Republic. Springer.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research*, 2023.

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*, ACL '24, pages 36–53, Bangkok, Thailand.

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*, abs/1907.11692.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjàvu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA).

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 38 others. 2024. StarCoder 2 and the Stack v2: The next generation. *ArXiv preprint*, abs/2402.19173.

Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st Int. Conference on Advances in Neural Information Processing Systems*, NeurIPS '17, Long Beach, California, USA. Curran Associates Inc.

Dominik Macko, Robert Moro, Adaku Uchendu, Jason Lucas, Michiharu Yamashita, Matúš Pikuliak, Ivan Srba, Thai Le, Dongwon Lee, Jakub Simko, and

Maria Bielikova. 2023. MULTITuDE: Large-scale multilingual machine-generated text detection benchmark. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, EMNLP '23, pages 9960–9987, Singapore.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, and 27 others. 2024. Granite Code Models: A family of open foundation models for code intelligence. *ArXiv preprint*, abs/2405.04324.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. 2023. DetectGPT: zero-shot machine-generated text detection using probability curvature. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23, Honolulu, Hawaii, USA. JMLR.org.

Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The Vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '23, pages 4763–4788, Singapore. 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 '25, pages 10570–10593, Vienna, Austria. 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*, EMNLP '25, pages 31263–31289, Suzhou, China. Association for Computational Linguistics.

Shinwoo Park, Hyundong Jin, Jeong won Cha, and Yo-Sub Han. 2025. Detection of LLM-paraphrased code and identification of the responsible LLM using coding style features. *ArXiv preprint*, abs/2502.17749.

Indraneil Paul, Haoyi Yang, Goran Glavaš, Kristian Kersting, and Iryna Gurevych. 2025. ObscuraCoder: Powering efficient code LM pre-training via obfuscation grounding. In *Proceedings of the International Conference on Learning Representations*, ICLR '25, Singapore.

Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. In *Proceedings of the Conference on Neural Information Processing Systems*, NeurIPS '18, pages 6639–6649, Montréal, Canada.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, NeurIPS '21, virtual.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Proceedings of the Conference on Neural Information Processing Systems*, NeurIPS '23, New Orleans, LA, USA.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, and 34 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *ArXiv preprint*, abs/2403.05530.

Chris Rohlfs. 2025. Generalization in neural networks: A broad survey. *Neurocomputing*, 611:128701.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code Llama: Open foundation models for code. *ArXiv preprint*, abs/2308.12950.

Saiful Islam Salim, Rubin Yuchan Yang, Alexander Cooper, Suryashree Ray, Saumya Debray, and Sazzadur 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*, EMNLP '24, pages 445–463, Miami, Florida, USA. Association for Computational Linguistics.

Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, and 1 others. 2025. Natural language outlines for code: Literate programming in the llm era. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE '25, pages 150–161.

Rafael A. Rivera Soto, Kailin Koch, Aleem Khan, Barry Y. Chen, Marcus Bishop, and Nicholas Andrews. 2024. Few-shot detection of machine-generated text using style representations. In *Proceedings of the Twelfth International Conference on Learning Representations*, ICLR '24, Vienna, Austria. OpenReview.net.

Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2024. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2).

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers. *ArXiv preprint*, abs/2009.05617.

Adaku Uchendu, Thai Le, Kai Shu, and Dongwon Lee. 2020. Authorship attribution for neural text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, EMNLP '20, pages 8384–8395, Online. Association for Computational Linguistics.

Saranya Venkatraman, Adaku Uchendu, and Dongwon Lee. 2024. GPT-who: An information density-based machine-generated text detector. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 103–115, Mexico City, Mexico. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, EMNLP '22, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yuxia Wang, Jonibek Mansurov, Petar Ivanov, Jinyan Su, Artem Shelmanov, Akim Tsvigun, Osama Mohammed Afzal, Tarek Mahmoud, Giovanni Puccetti, Thomas Arnold, Alham Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. 2024a. M4GT-bench: Evaluation benchmark for black-box machine-generated text detection. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, ACL '24, pages 3964–3992, Bangkok, Thailand. 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, Thomas Arnold, Alham Fikri Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. 2024b. 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*, EACL '24, pages 1369–1407, St. Julian's, Malta. Association for Computational Linguistics.

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 Etter, Maiya Goloburda, Tarek Mahmoud, Raj Vardhan Tomar, Nurkhan Laiyk, and 7 others. 2025. GenAI content detection task 1: English and multilingual machine-generated text detection: AI vs. human. In *Proceedings of the Workshop on GenAI Content Detection*, pages 244–261, Abu Dhabi, UAE.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Griffin Thomas Adams, Jeremy Howard, and Iacopo Poli. 2025. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, ACL '25, pages 2526–2547, Vienna, Austria. Association for Computational Linguistics.

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 Transactions on Software Engineering Methodology*, 34(4).

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *ArXiv preprint*, abs/2505.09388.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024. Qwen2.5 technical report. *ArXiv preprint*, abs/2412.15115.

Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, and 11 others. 2024. Yi: Open foundation models by 01.AI. *ArXiv preprint*, abs/2403.04652.

Zhihua Zhang. 2015. The singular value decomposition, applications and beyond. *ArXiv preprint*, abs/1510.08532.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, and 7 others. 2024. CodeGemma: Open code models based on Gemma. *ArXiv preprint*, abs/2406.11409.

Appendix

A Data Statement

A.1 General Information

Dataset Title AICD Bench

Dataset Version 1.0 (September 2025)

Data Statement Version 1.0 (September 2025)

A.2 Executive Summary *AICD Bench* is designed for a rigorous and standalone evaluation of systems that distinguish human-written code from machine-generated, hybrid, and adversarial code across multiple programming languages, code generators, and domains. It compiles a large and diverse set of code snippets drawn from widely used programming platforms and contemporary LLM code generators, ensuring broad language and domain coverage.

Intended Use: *AICD Bench* is intended exclusively for research, particularly for developing and evaluating models that detect machine-generated code. Researchers can analyze how programming languages, generation models, and application domains affect detection accuracy and robustness. The dataset aims to support improved automated code assessment, ethical use, and accountability in software engineering.

Usage Restrictions: The dataset is provided solely for academic and research purposes. Commercial use is prohibited without prior written consent from the dataset creators. Users must follow ethical guidelines and ensure that the findings do not violate privacy, intellectual property, or other legal constraints. Redistribution is not permitted without authorization from the dataset custodians.

Source: We open-source all the splits of *AICD-Bench* on Hugging Face.[7]

License Creative Commons Attribution–NonCommercial 4.0 International (CC BY–NC 4.0). We will release **AICD Bench** under CC BY–NC 4.0. You may share and adapt the dataset for *non-commercial* research with appropriate attribution; no additional restrictions may be applied. For any commercial use, prior written permission from the authors is required. See the full license at https://creativecommons.org/licenses/by-nc/4.0/. The dataset is provided "as is," without warranties of any kind.

B Added Models

We enlarge the `DroidCollection` (Orel et al., 2025b) with additional models for nearly every family of models. Table 9 indicates which models were taken from the original `DroidCollection`, and which were introduced in *AICD Bench*.

C Data Distribution

Figure 3 shows that the code in our dataset is consistent with healthy coding practices of real-world projects. Namely, AST depth is moderately concentrated (most of the values are between 10 and 15), indicating a prevalence of structured but not overly nested control flow; the alphanumeric fraction peaks at 0.65 with a small standard deviation, reflecting clear, meaningful identifiers in the code, not noise, logs or obfuscated files; average line length clusters tightly between 20-60 characters, while max line length exhibits a right-skewed tail extending beyond 200 characters reflecting adherence to the best practices of code structuring; and file size is heavily skewed toward short snippets (<100 lines), with a median of nearly 40 lines, indicative of fine-grained code units that would fit into the context window of most of the classifiers.

C.1 Prompt Templates

Task 1 zero-shot prompt

```
Reply with exactly one word
Human or AI.
Given this code,
identify its origin
{code}
```

Task 2 zero-shot prompt

```
Reply with exactly one word from
this list:
human, deepseek-ai,
Qwen,01-ai,
bigcode,Gemma,
Phi,meta-llama,ibm-granite,mistralai,
OpenAI,Gemini.
Given this code,
identify its origin
{code}
```

Task 3 zero-shot prompt

```
Reply with exactly one word
Human, AI, Hybrid or Adversarial.
Given this code,
was it written by Human, AI,
in Hybrid collaboration or by an
```

[7] https://huggingface.co/AICD-bench

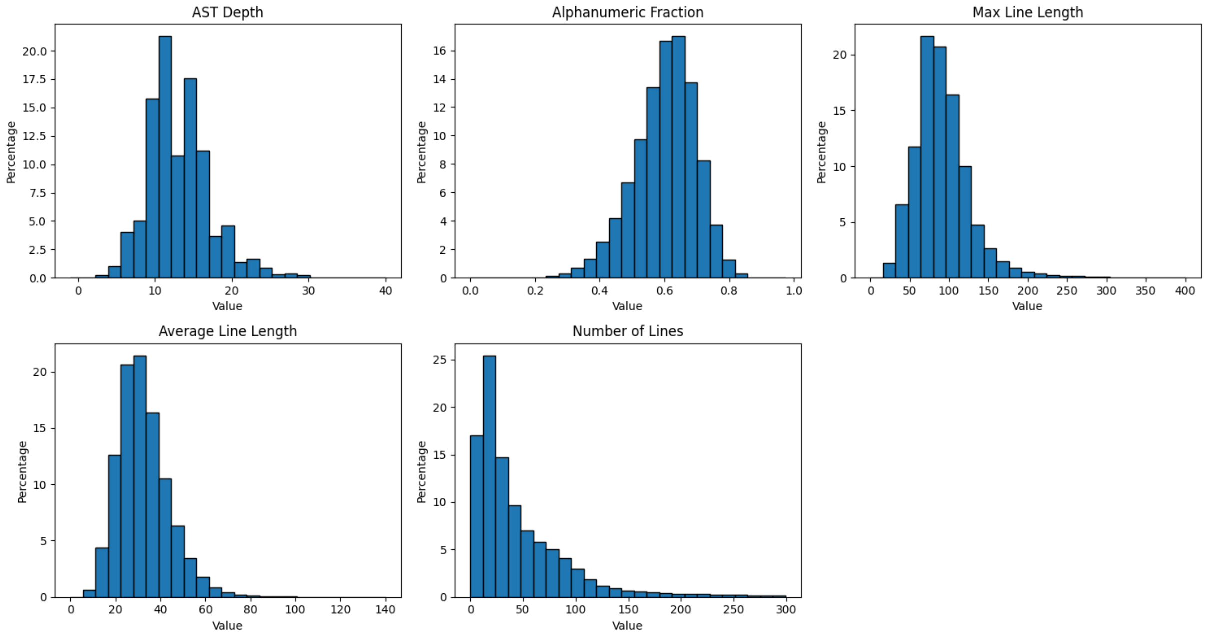


Figure 3: Distribution of code properties.

```
Adversarial model which
tried to fool the detector mimicing
human? {code}
```

For CoT prompts, we summarized the key code features per class using Gemini-2.5-Flash.

Task 1 CoT prompt

```
You are a code-origin classifier.
Infer whether a given code snippet
was written by a Human or an AI
    assistant.
Reason silently and do NOT reveal
your reasoning.
Output exactly one word: Human or
    AI.

When deciding, consider
(but do not list) signals such as:
- Naming:
    verbosity/descriptiveness,
camelCase/snake_case consistency,
    oddly generic names.
- Comments/docstrings: density,
    uniform template-like
    phrasing, section headers,
    per-function summaries.
- Type hints & annotations:
    pervasive, perfectly
    consistent typing vs.
    ad-hoc/mixed usage.
- Structure/style: consistent
    formatter fingerprints
    (Black/Prettier), tidy
    imports, exhaustive
    edge-cases, spotless spacing;
  versus idiosyncratic style,
      inconsistencies, TODO/WIP
      markers.
- Boilerplate/templates: license
    blocks, README headers,
    instructional comments,
    scaffolded regions, foo/bar
    placeholders,
  repeated auto-generated patterns.
- API/library usage: unused
    imports, verbose defensive
    code, redundant checks,
    generic logging.
- Testing/examples: synthetic
    data, contrived examples,
    pedantic error messages.
- AI artifacts: phrases like "This
    function...", tutorial-like
    narration, or code mirroring
    common LLM examples.
- Human artifacts: partial
    implementations, inline hacks,
    commented-out experiments,
    domain-specific shortcuts,
    mismatched styles, quick fixes.

If uncertain, choose the more
    likely origin using the
    balance of signals.
Respond with a single word only.

Code:
‘‘‘{code}‘‘‘
```

Task 2 CoT prompt

```
You are a code-origin classifier.
Infer which single
author produced the code:
human, deepseek-ai,
Qwen, 01-ai, bigcode,
Gemma, Phi, meta-llama,
ibm-granite, mistralai,
OpenAI, Gemini.

Think silently to compare candidate
```

Model Family	Model
Yi	Yi-Coder-9B
	Yi-Coder-9B-Chat
	Yi-Coder-1.5B-Chat
	Yi-Coder-1.5B
GPT	GPT-4o-mini
	GPT-4o
Qwen	Qwen2.5-Coder-7B
	Qwen2.5-Coder-7B-Instruct
	Qwen2.5-Coder-1.5B-Instruct
	Qwen2.5-Coder-32B-Instruct
	Qwen2.5-72B-Instruct
	Qwen2.5-Coder-1.5B
	Qwen2.5-Coder-14B-Instruct
	QwQ-32B
	Qwen3-14B
	Qwen3-32B
	Qwen3-235B-A22B
	Qwen3-30B-A3B
Gemma	codegemma-7b-it
	codegemma-7b
	codegemma-2b
	gemma-3-27b-it
	gemma-3n-e4b-it
	gemma-3-12b-it
	gemma-3-4b-it
Deepseek	deepseek-coder-6.7b-instruct
	deepseek-coder-6.7b-base
	deepseek-coder-1.3b-instruct
	deepseek-coder-1.3b-base
	DeepSeek-R1
	DeepSeek-V3-0324
Granite	granite-8b-code-instruct-4k
	granite-8b-code-base-4k
	granite-3.2-2b-instruct
	granite-3.3-8b-base
	granite-3.3-8b-instruct
	granite-34b-code-instruct-8k
	granite-3b-code-base-128k
	granite-3b-code-instruct-128k
Llama	Llama-3.1-8B-Instruct
	Llama-3.2-3B
	Llama-3.1-70B-Instruct
	Llama-3.3-70B-Instruct
	Llama-3.3-70B-Instruct-Turbo
	Llama-3.2-1B
	Llama-3.1-8B
	CodeLlama-70b-Instruct-hf
	CodeLlama-34b-Instruct-hf
	CodeLlama-7b-hf
	Llama-3.2-11B-Vision-Instruct
	Llama-3.2-90B-Vision-Instruct
	Llama-4-Maverick-17B-128E-Instruct-FP8
	Llama-4-Scout-17B-16E-Instruct
	Meta-Llama-3.1-405B-Instruct
Phi	Phi-3-small-8k-instruct
	Phi-3-mini-4k-instruct
	phi-4
	Phi-3-medium-4k-instruct
	phi-2
	Phi-3.5-mini-instruct
	Phi-4-multimodal-instruct
Mistral	Mistral-Small-24B-Instruct-2501
	Devstral-Small-2505
	Mistral-7B-Instruct-v0.3
	Mistral-Nemo-Instruct-2407
	Mixtral-8x7B-Instruct-v0.1
BigCode	starcoder2-15B
	starcoder
	starcoder2-7b
	starcoder2-3b
	starcoderbase-1b
	starcoderbase-3b
Gemini	**gemini-1.5-flash**
	gemini-1.5-flash-8b
	gemini-2.0-flash
	gemini-2.0-flash-lite
	gemini-2.5-flash-preview-05-20

Table 9: Model families and their selected models used in *AICD Bench*. **Bold** entries are models that are not used in `DroidCollection`. *Italic* entries are reasoning models.

```
profiles;
do not reveal reasoning.
Output exactly one word:
one of the labels above.

Use discriminative,
code-level signals:

GENERAL
- Language & libs:
default choices (Python/JS/Java),
PyTorch vs. TF/JAX,
numpy/pandas usage,
pytest vs. unittest.
- Structure:
module/import ordering,
helper naming density,
dataclass/typing/async usage,
error-handling patterns.
- Textual:
identifier casing/length,
docstring style
(Google/NumPy/ReST/none),
comment tone,
message phrasing.
- Artifacts: template scaffolds,
README-like headers,
tutorial narrations,
synthetic examples,
placeholder vars.

CANDIDATE PROFILES
(heuristics; match by best fit):
- OpenAI: PEP8-clean Python,
f-strings,
type hints moderate,
Google/NumPy docstrings,
careful edge-case checks.
- Gemini: tendency to verbose
instructional comments,
TF/JAX-friendly snippets,
longer docstrings with bullets.
- meta-llama: PyTorch-first
demos,
concise comments,
torchvision/transformers idioms,
manual seed-setting.
- mistralai: compact Python,
minimal ceremony,
itertools/collections use,
terse error messages/tests.
- Qwen: occasional
bilingual tokens/comments,
pandas/NumPy-heavy utilities,
explicit dtype handling.
- deepseek-ai:
performance-leaning tweaks,
vectorization/numba hints,
assert-style sanity checks.
- 01-ai: straightforward
baseline patterns,
minimal comments,
direct loops over abstractions.
- bigcode: repository/tooling
scaffolds,
license headers or
codegen-ready templates,
typed stubs.
- Gemma: JAX/Flax hints or TF ops,
functional style utilities,
explicit shapes in comments.
- Phi: didactic
step-by-step snippets,
simple class wrappers,
explicit prints/logs for tracing.
- ibm-granite:
```

```
enterprise-style structure,
logger configuration blocks,
clear exceptions/messages.
- human: mixed
styles/inconsistencies,
partial impls/TODOs,
pragmatic hacks,
commented-out experiments.

If uncertain,
choose the single most
plausible author by
strongest profile match.

Code:
‘‘‘{code}‘‘‘
```

Task 3 CoT prompt

```
You are a code-origin classifier.
Infer whether a given code snippet
    was written by a Human, an AI
    assistant, Hybridly generated,
    or generated by an Adversarial
    model trained to mimic humans
    and fool detectors.
Reason silently and do NOT reveal
    your reasoning.
Output exactly one word: Human,
    AI, Hybrid, or Adversarial.

When deciding, consider (but do
    not list) signals such as:
Naming: verbosity/descriptiveness,
    camelCase/snake_case
    consistency, oddly generic
    names.
Comments/docstrings: density,
    uniform template-like
    phrasing, section headers,
    per-function summaries.
Type hints and annotations:
    pervasive, perfectly
    consistent typing vs.
    ad-hoc/mixed usage.
Structure/style: consistent
    formatter fingerprints
    (Black/Prettier-like), tidy
    import grouping, exhaustive
    edge-cases, spotless spacing;
versus idiosyncratic style, small
    inconsistencies, TODO/WIP
    markers.
Boilerplate/templates: license
    blocks, README-like headers,
    instructional comments,
    scaffolded regions,
    placeholder variables
    (foo/bar), repeated patterns
    that look auto-generated.
API/library usage: unused imports,
    overly defensive patterns,
    verbose step-by-step code
    where a library call would
    suffice, redundant checks,
    generic logging.
Testing/examples: synthetic data,
    contrived examples, consistent
    pedantic error messages.
Artifacts of AI text: phrases like
    "This function...", "The
    following code...", or
    tutorial-like narration, or
    code that mirrors common LLM
    examples.
Human artifacts: partial
    implementations, inline hacks,
    commented-out experiments,
    domain-specific shortcuts,
    mismatched styles from
    multiple edits,
    quick-and-dirty error handling.

If uncertain, choose the more
    likely origin using the
    balance of signals.
Again: respond with a single word
    only.

Classify the origin of this code
    as Human, AI, Hybrid, or
    Adversarial.
Return exactly one word: Human,
    AI, Hybrid, or Adversarial.

Code:
‘‘‘{code}‘‘‘
```

D Task 1 Detailed Performance

Figure 5 illustrates that the domain shift has a greater impact than the language shift. Models perform better when the programming language changes, as long as the code remains within the seen domain of algorithmic problems. Notably, performance degrades markedly on unseen domains. In fact, when both the domain and the language are unseen, performance is no worse than when only the domain is unseen, suggesting that the shift in the domain outweighs the task's overall difficulty.

Another key observation is that deep learning models exhibit strong overfitting to the training settings: they achieve near-perfect accuracy in identifying machine-generated code when both the domain and programming language match those in the training data. However, their performance drops sharply when faced with out-of-distribution examples. It persists even when the domain remains seen, but the programming language is unseen, which is a much easier setting. In contrast, the performance gap for classical models is not that large in these settings.

When comparing the programming language-wise performance, shown in Figure 4, it is clear that the models generally perform better when the input is Python, which is expected since Python is the language most prevalent in the training set. For other programming languages, performance is quite similar, with the exception of PHP, which syntactically differs from the rest and, thus yielding the lowest performance.

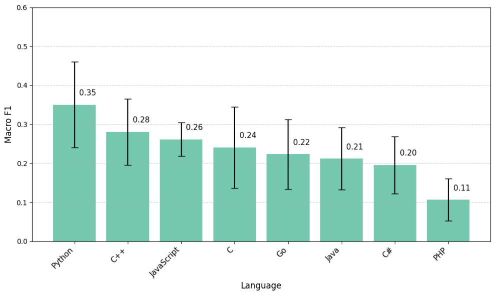


Figure 4: **Task 1 (Robust Binary Classification):** detectors' language-wise performance.

D.1 SVM Performance Explained

To understand why the TF-IDF SVM outperformed other baselines, we examined which n-grams most strongly indicated human vs. AI code. For interpretability, we back-projected the SVM coefficients from the SVD latent space into the original TF-IDF feature space and ranked tokens by their contributions.

We found clear stylistic differences. AI-generated code often uses verbose, prompt-echoing identifiers like *answer*, *output*, *result*, *tests*, and *index*. In contrast, human-written code tended to use shorter, organic identifiers like *li*, *nums*, *pos*, *a1*, and *cur*. These patterns are not tied to any specific programming language, which helps explain why the SVM generalizes well to unseen languages: the classifier captures stylistic regularities in human vs. AI code rather than language-specific syntax.

E Task 2 Detailed Performance

Figure 6 shows that classical machine learning models achieve nearly zero macro F1-score on out-of-domain generators, indicating a severe lack of generalization. Among these models, TF-IDF consistently yields the best performance, outperforming AST-based representations and the combination of TF-IDF and AST features. By contrast, deep learning models exhibit significantly better generalization capabilities. Among them, ModernBERT achieves the highest performance across in-domain and out-of-domain generators.

The low macro F1-score of classical models on Task 2 is further illustrated in Figure 7: these models tend to predict a single dominant class, resulting in poor performance across minority classes. Similarly, among deep learning models, CodeT5+ also shows a strong bias towards predicting a single class. In contrast, the other deep learning models, particularly ModernBERT are less biased.

F Task 3 Detailed Performance

From Figure 8, we observe that only the CatBoost model can utilize AST features to work at least in-domain settings. Other classical models fail shortly when trying to use these features. In case of TF-IDF, on the other hand, all classical models have demonstrated comparable performance for both in-domain and out-of-domain data. Deep learning models, similarly to Task 1, achieve good in-domain performance (over 71%) while performing poorly out-of-domain (below 25%).

When analyzing the confusion matrices for the models (Figure 9), it is obvious that the classical models barely learned anything: they were mainly predicting a single class. Deep learning models, in contrast, tend to learn correct class assignments. Interestingly, adversarial code samples are often misclassified as human- or AI-written, highlighting that the adversarial generation achieved its goals. Additionally, hybrid cases are being misclassified with AI-generated ones, as hybrid generation also involves code rewriting that can drastically alter the initial human-written code structure.

G Examples of Model Failures

Tables 10 to 12 showcase where all baselines fail. In Task 1, models often fail on very short code snippets. Boilerplate code is also frequently mislabeled as AI-generated, likely because such patterns are common in Code-LM training data, leading models to misclassify them as AI output. For Task 2, we omit predicted classes since models rarely agree; each typically assigns a different class to the same input, revealing task difficulty. In Task 3, errors again focus on boilerplate code. Also, misclassifications mostly occur among similar categories, AI-generated, hybrid, and adversarial, indicating that models struggle to discern fine-grained distinctions between them.

H SHAP analysis

Figure 10 compares token-level SHAP visualizations for correctly and incorrectly classified samples in Task 1, while Figure 11 shows the analogous comparison for hybrid cases in Task 3. We also manually inspect additional examples; the figures show representative samples that illustrate the attribution patterns observed in a more general sense. Analysis of other tasks and classes did not yield useful information .

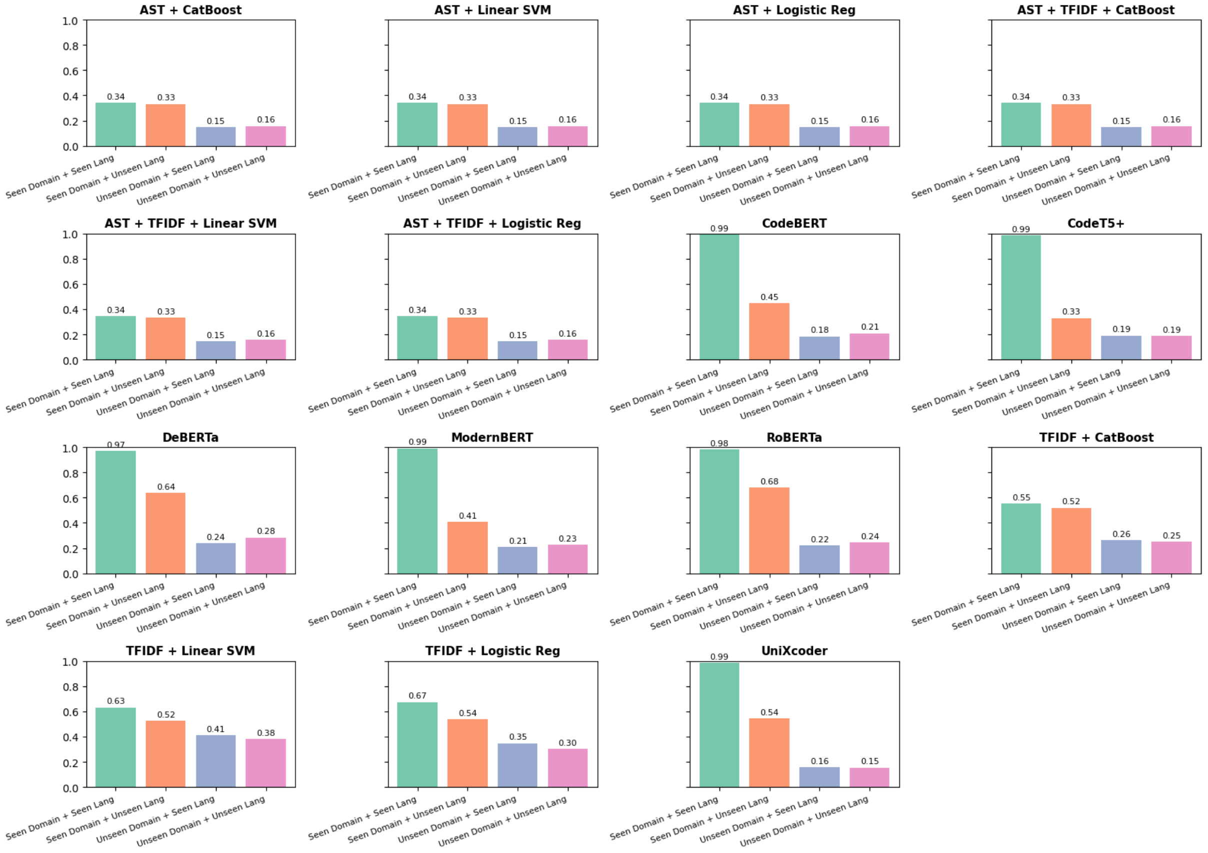


Figure 5: **Task 1 (Robust Binary Classification):** performance evaluation of the detectors.

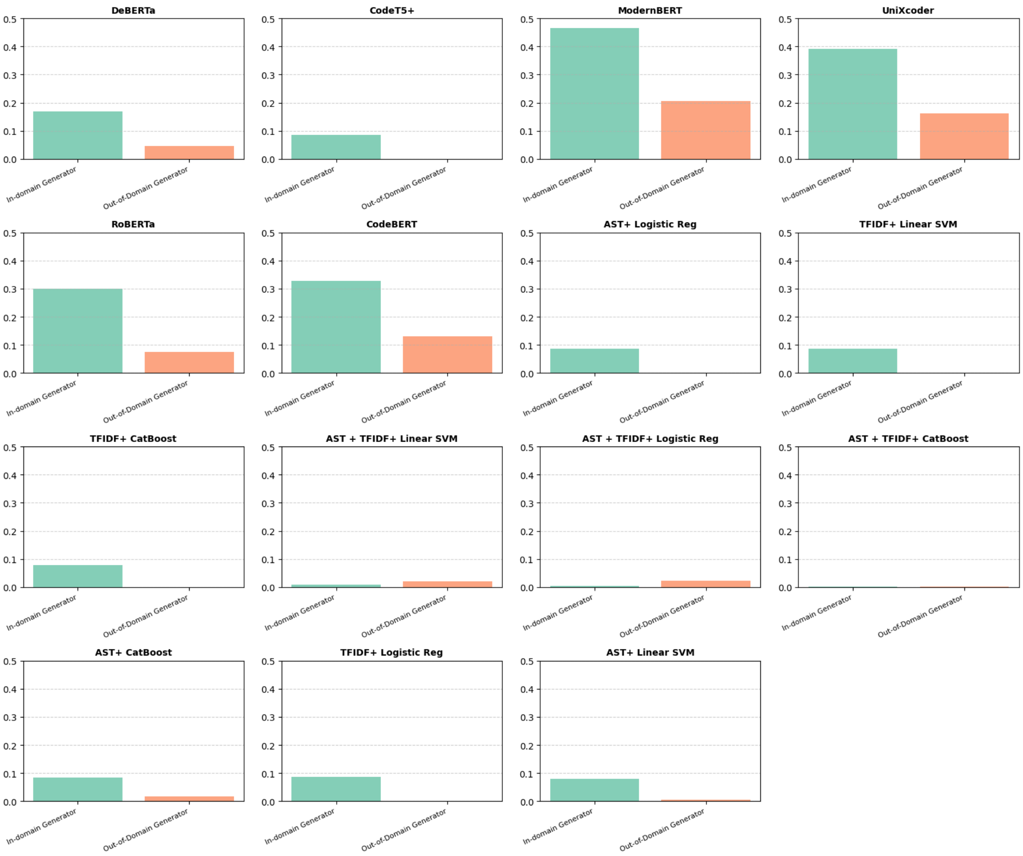


Figure 6: **Task 2 (Model Family Attribution):** performance evaluation of the detectors.

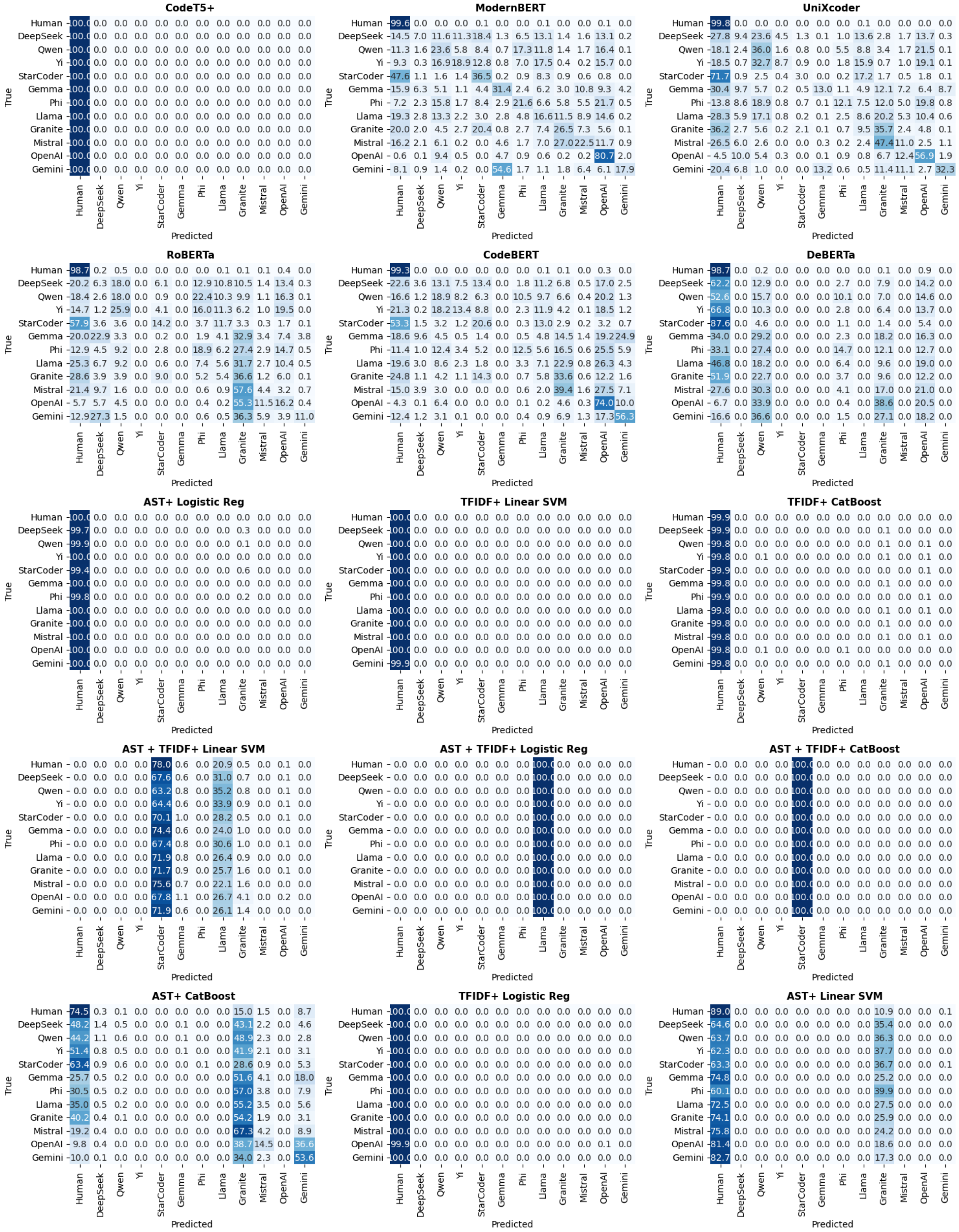


Figure 7: **Task 2 (Model Family Attribution):** confusion matrices for detectors.

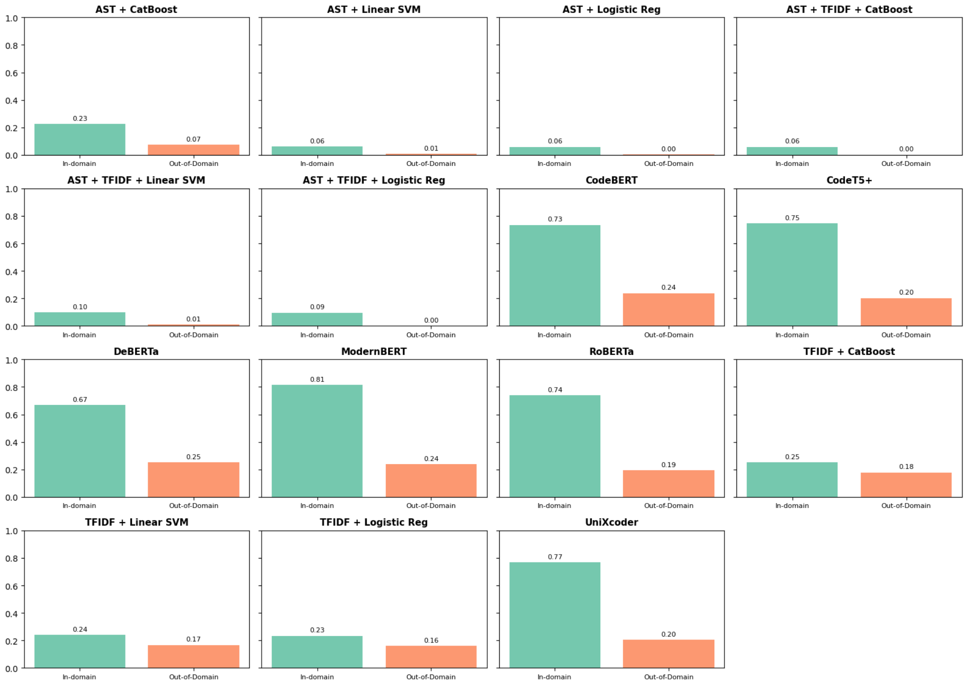


Figure 8: **Task 3 (Fine-Grained Human-Machine Classification)**: performance evaluation of the detectors.

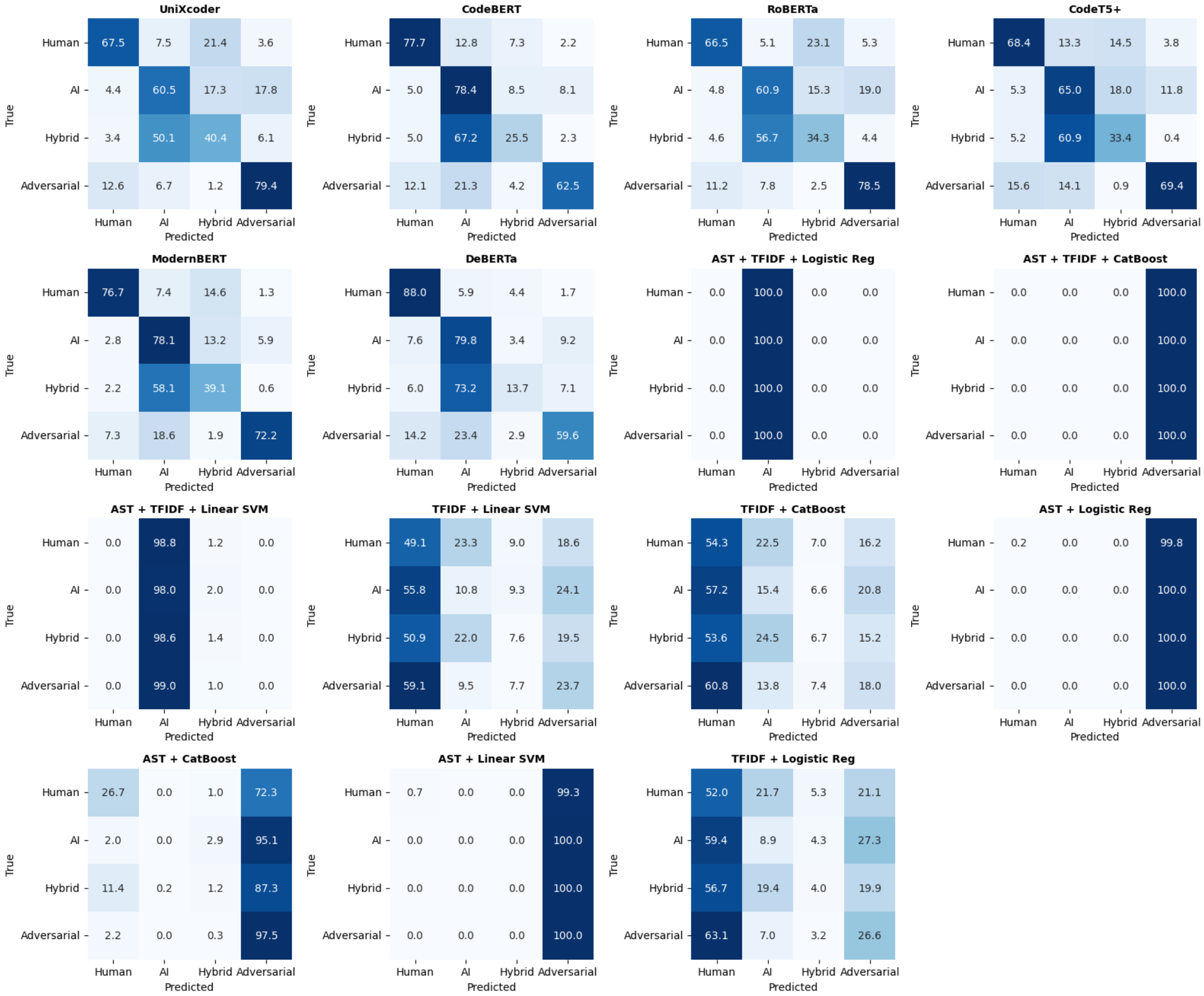


Figure 9: **Task 3 (Fine-Grained Human-Machine Classification)**: confusion matrices for detectors.

<table>
<tr><th>Code</th><th>True Label</th><th>Predicted Label</th><th>Possible explanation</th></tr>
<tr><td>

```python
def max(a, b):
    return a if a > b else b


def min(a, b):
    return a if a < b else b


class StringHash:
    def __init__(self, lst):
        """two mod to avoid hash crush"""
        # use two class to compute is faster!!!
        self.n = len(lst)
        self.p = random.randint(26, 100)
        self.mod = random.randint(10 ** 9 + 7, 2 ** 31 - 1)
        self.pre = [0] * (self.n + 1)
        self.pp = [1] * (self.n + 1)
        for j, w in enumerate(lst):
            self.pre[j + 1] = (self.pre[j] * self.p + w) % self.mod
            self.pp[j + 1] = (self.pp[j] * self.p) % self.mod
        return

    def query(self, x, y):
        """range hash value index start from 0"""
        # assert 0 <= x <= y <= self.n - 1
        if y < x:
            return 0
        # with length y - x + 1 important!!!
        ans = (self.pre[y + 1] - self.pre[x] * self.pp[y - x + 1]) % self.mod
        return ans


class Solution:
    def countPrefixSuffixPairs(self, words: List[str]) -> int:
        ans = 0
        st = "".join(words)
        sh1 = StringHash([ord(w) - ord("a") for w in st])
        sh2 = StringHash([ord(w) - ord("a") for w in st])
        pre = defaultdict(int)
        length = 0
        for word in words:
            m = len(word)
            for i in range(1, m+1):
                prefix = (sh1.query(length, length+i-1), sh2.query(length, length+i-1), i)
                suffix = (sh1.query(length+m-1-i+1, length + m-1), sh2.query(length+m-1-i+1, length + m-1), i)
                if prefix == suffix:
                    ans += pre[prefix]

            prefix = (sh1.query(length, length + m - 1), sh2.query(length, length + m - 1), m)
            pre[prefix] += 1
            length += m
        return ans
```

</td><td>Human-Written</td><td>AI-Generated</td><td>Dosctrings in StringHash do not look human-written. That may be the case that StringHash is just a Boilerplate implementation</td></tr>
<tr><td>

```python
s = input().strip()

x = int(input().strip())

if (s=="ABC" and x<2000) or (s=="ARC" and x<2800) or (s=="AGC" and x>=1200):

    print('yes')

else:

    print('no')
```

</td><td>Human-Written</td><td>AI-Generated</td><td>Short code with not enough signal for classification</td></tr>
<tr><td>

```python
N,*A=map(int,open(0).read().split())

T=[*zip(A,A[4:]+A[3:4])]

print(sum(max(0,a+b-N)for a,b in T),sum(map(min,T)))
```

</td><td>Human-Written</td><td>AI-Generated</td><td>Short code with not enough signal for classification</td></tr>
</table>

Table 10: **Task 1 (Robust Binary Classification):** examples of model misclassification and their possible explanations.

<table>
<tr><th>Code</th><th>Model Family</th></tr>
<tr><td>

```java
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.security.KeyStore;
import java.security.spec.PKCS8EncodedKeySpec;
import javax.security.auth.callback.*;
import javax.security.auth.Subject;
import javax.security.auth.spi.*;
import java.util.*;
import java.nio.file.*;
import java.security.*;
import java.util.logging.*;

public class IoTSecurityFramework {

    private static final String ALGORITHM = "AES/CBC/PKCS5Padding";
    private static final Logger logger = Logger.getLogger(IoTSecurityFramework.class.getName());
    private SecretKey secretKey;
    private IvParameterSpec ivSpec;

    public IoTSecurityFramework() throws Exception {
        // Generate symmetric key for AES
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128); // Key size
        secretKey = keyGen.generateKey();
        ivSpec = new IvParameterSpec(new byte[16]); // Initialization vector
    }

    public byte[] encrypt(String data) throws Exception {
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
        return cipher.doFinal(data.getBytes());
    }

    public String decrypt(byte[] encryptedData) throws Exception {
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
        return new String(cipher.doFinal(encryptedData));
    }

    public boolean authenticateDevice(String deviceId, String sessionToken) {
        // Mock authentication
        return "validDeviceId".equals(deviceId) && "validSessionToken".equals(sessionToken);
    }
    public void roleBasedAccessControl(String role) throws SecurityException {
        // Simple RBAC check
        if (!"admin".equals(role)) {
            throw new SecurityException("Access Denied for role: " + role);
        }
    }
    public void logAccessEvent(String message) {
        logger.info(message);
    }
    public void monitorForIntrusion() {
        // Placeholder for intrusion detection mechanism
        System.out.println("Monitoring for unauthorized access...");
    }
    public boolean secureCodingGuide() {
        // Placeholder for guide return
        System.out.println("Guidelines: Validate inputs, Use prepared statements, Error handling.");
        return true;
    }
    public static void main(String[] args) throws Exception {
        IoTSecurityFramework securityFramework = new IoTSecurityFramework();
        // Sample test
        String originalData = "Sensitive IoT Data";
        byte[] encryptedData = securityFramework.encrypt(originalData);
        String decryptedData = securityFramework.decrypt(encryptedData);
        securityFramework.logAccessEvent("Data encrypted and decrypted successfully.");
        System.out.println("Decrypted data: " + decryptedData);
        securityFramework.monitorForIntrusion();
        try {
            securityFramework.roleBasedAccessControl("user");
        } catch (SecurityException e) {
            securityFramework.logAccessEvent(e.getMessage());
        }

        securityFramework.secureCodingGuide();
    }
}
```

</td><td>Granite</td></tr>
<tr><td>

```python
def data_gen_args(self, context: str) -> dict:
        if context == 'train':
            return dict(
                horizontal_flip=self.horizontal_flip,
                vertical_flip=self.vertical_flip,
                image_size=self.crop_size
            )
        return dict(image_size=self.crop_size)
```

</td><td>Mistral</td></tr>
</table>

Table 11: **Task 2 (Model Family Attribution):** code snippets that caused errors.

<table>
<tr><th>Code</th><th>True Label</th><th>Predicted Label</th><th>Possible explanation</th></tr>
<tr><td><pre>
import pygame
import sys
import time

# Constants
SCREEN_WIDTH = 1026
SCREEN_HEIGHT = 700
WINNING_SOUND_FILE = 'Sound Effects/smb_stage_clear.wav'
BACKGROUND_IMAGE_FILE = "Images/Background2.jpg"
FONT_FILE = 'font.ttf'
WIN_TEXT = 'YOU WON!'
WIN_TEXT_COLOR = (50, 205, 50)
FONT_SIZE = 26
GAME_DURATION = 6 # seconds

class Game:
    """Represents the game."""

    def __init__(self):
        """Initializes the game."""
        pygame.init()
        self.screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
        pygame.display.set_caption("Super Python Bros.")
        self.clock = pygame.time.Clock()
        self.winning_sound = self.load_winning_sound()
        self.background_image = self.load_background_image()
        self.win_text = self.render_win_text()

    def load_winning_sound(self) -> pygame.mixer.Sound | None:
        """Loads the winning sound effect."""
        try:
            return pygame.mixer.Sound(WINNING_SOUND_FILE)
        except pygame.error as e:
            print(f"Error loading winning sound: {e}")
            return None

    def load_background_image(self) -> pygame.Surface | None:
        """Loads the background image."""
        try:
            return pygame.image.load(BACKGROUND_IMAGE_FILE).convert()
        except pygame.error as e:
            print(f"Error loading background image: {e}")
            return None

    def render_win_text(self) -> pygame.Surface:
        """Renders the win text."""
        font = pygame.font.Font(FONT_FILE, FONT_SIZE)
        return font.render(WIN_TEXT, False, WIN_TEXT_COLOR)

    def build_win_menu(self) -> None:
        """Builds the win menu."""
        # Clear the screen with the background image
        self.screen.blit(self.background_image, (0, 0))
        # Render the win text
        self.screen.blit(self.win_text, (400, 345))
        # Update the display
        pygame.display.flip()

    def run(self) -> None:
        """Runs the game."""
        if self.winning_sound:
            self.winning_sound.play()

        end_time = time.time() + GAME_DURATION
        while time.time() < end_time:
            self.build_win_menu()

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        pygame.quit()
                        sys.exit()

            self.clock.tick(60)

if __name__ == "__main__":
    game = Game()
    game.run()
</pre></td><td>Hybrid</td><td>mainly AI-Generated, few Adversarial</td><td>Since it is a hybrid case, misclassification happens with two most similar cases</td></tr>
<tr><td><pre>
package zmq.util;

// Emulates the errno mechanism present in C++, in a per-thread basis.
public final class Errno
{
    private static final ThreadLocal<Integer> local = ThreadLocal.withInitial(() -> 0);

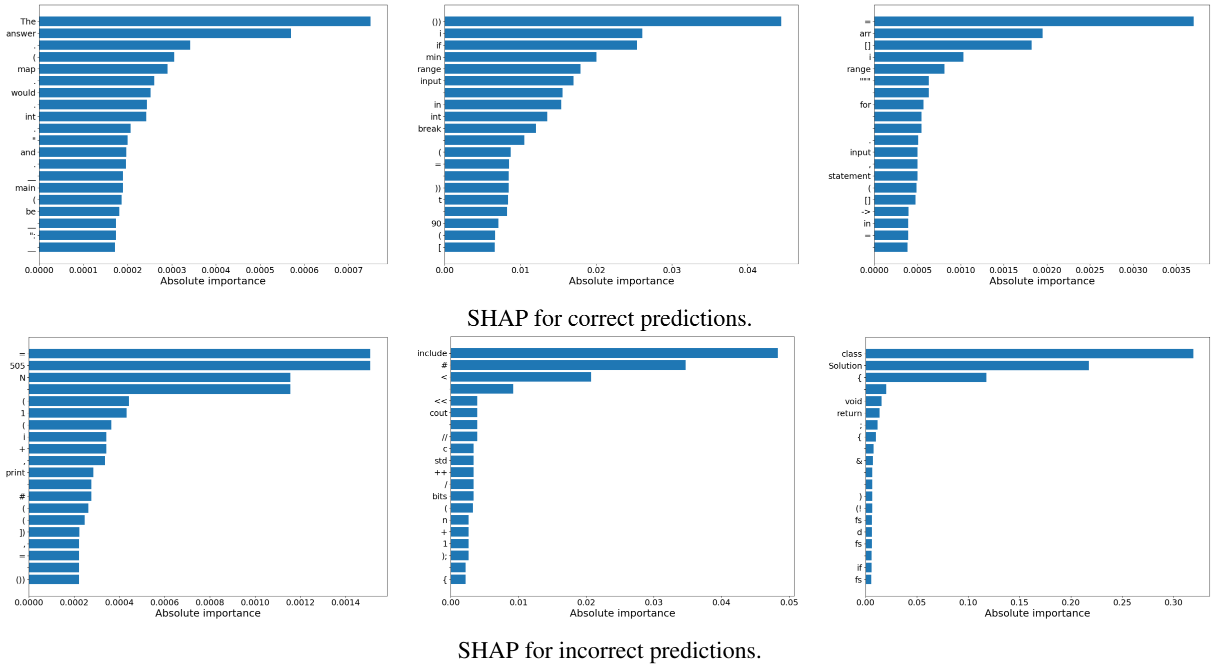
    public int get()
    {
        return local.get();
    }

    public void set(int errno)
    {
        local.set(errno);
    }

    public boolean is(int err)
    {
        return get() == err;
    }

    @Override
    public String toString()
    {
        return "Errno[" + get() + "]";
    }
}
</pre></td><td>Human-Written</td><td>All classes except Human-Written</td><td>Too boiler-plate like</td></tr>
</table>

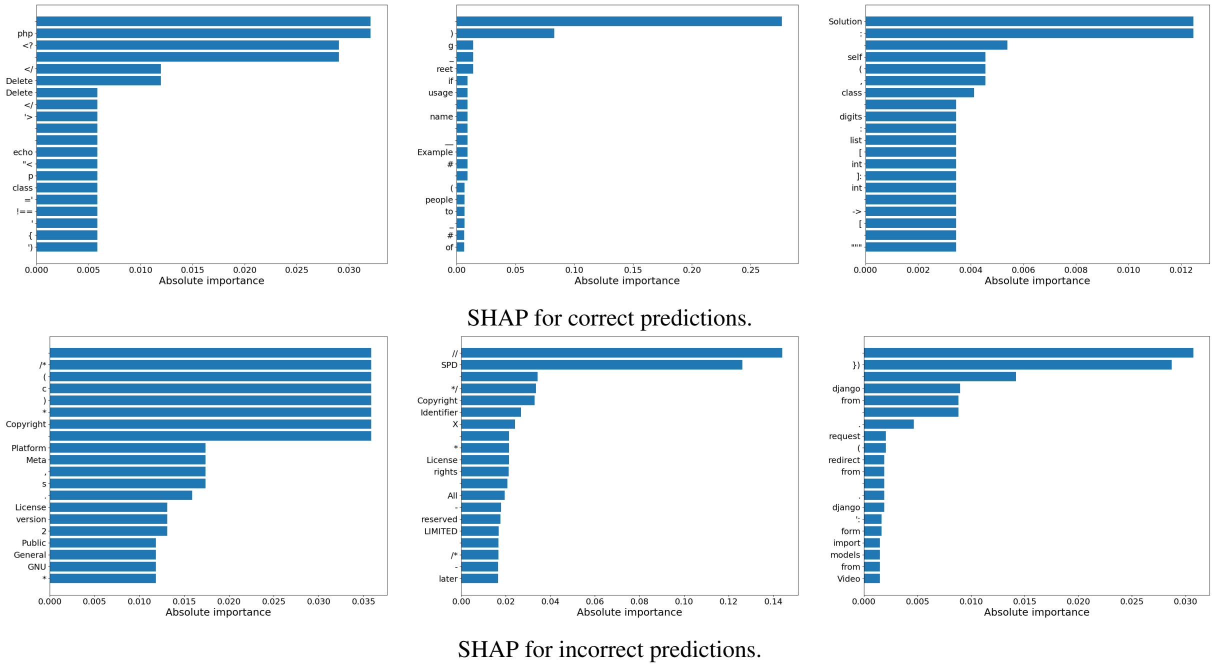
Table 12: **Task 3 (Fine-Grained Human-Machine Classification)**: examples of model misclassification and their possible explanations.



SHAP for correct predictions.

SHAP for incorrect predictions.

Figure 10: **Task 1 (Robust Binary Classification):** token-level SHAP visualizations comparing correct vs. incorrect predictions.



SHAP for correct predictions.

SHAP for incorrect predictions.

Figure 11: **Task 3 (Fine-Grained Human-Machine Classification):** token-level SHAP visualizations comparing correct vs. incorrect predictions for hybrid class prediction.