CodeMirage: A Multi-Lingual Benchmark for Detecting AI-Generated and Paraphrased Source Code from Production-Level LLMs

Hanxi Guo
Purdue University
guo778@purdue.edu

Siyuan Cheng
Purdue University
cheng535@purdue.edu

Kaiyuan Zhang
Purdue University
zhan4057@purdue.edu

Guangyu Shen
Purdue University
shen447@purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@purdue.edu

Abstract

Large language models (LLMs) have become integral to modern software development, producing vast amounts of AI-generated source code. While these models boost programming productivity, their misuse introduces critical risks, including code plagiarism, license violations, and the propagation of insecure programs. As a result, robust detection of AI-generated code is essential. To support the development of such detectors, a comprehensive benchmark that reflects real-world conditions is crucial. However, existing benchmarks fall short—most cover only a limited set of programming languages and rely on less capable generative models. In this paper, we present ***CodeMirage***, a comprehensive benchmark that addresses these limitations through three major advancements: (1) it spans ten widely used programming languages, (2) includes both original and paraphrased code samples, and (3) incorporates outputs from ten state-of-the-art production-level LLMs, including both reasoning and non-reasoning models from six major providers. Using ***CodeMirage***, we evaluate ten representative detectors across four methodological paradigms under four realistic evaluation configurations, reporting results using three complementary metrics. Our analysis reveals nine key findings that uncover the strengths and weaknesses of current detectors, and identify critical challenges for future work. We believe ***CodeMirage*** offers a rigorous and practical testbed to advance the development of robust and generalizable AI-generated code detectors.

1 Introduction

Large Language Models (LLMs) are rapidly evolving and demonstrating increasing capabilities in coding, fundamentally transforming the software development ecosystem. Recent LLMs such as ChatGPT [55] and Claude [4] exhibit remarkable code generation performance, producing high-quality outputs in response to concise natural language prompts. The emergence of reasoning-capable models like DeepSeek-R1 [26] has further accelerated LLM adoption among developers. According to Stack Overflow's industry report [72], 82.1% of the 65,000 surveyed developers report using ChatGPT [55] during their development workflow. Capitalizing on the strong coding abilities of LLMs, assistant tools such as GitHub Copilot [20] and Cursor [12] have been developed to enhance productivity by helping developers write, modify, and debug code directly within integrated development environments (IDEs). Furthermore, state-of-the-art LLM-based agentic systems such as OpenHands [83] achieve up to a 65.8% resolved rate on SWE-Bench [34], demonstrating the effectiveness of LLMs in addressing real-world software engineering tasks. These trends indicate that LLMs and their associated tools are becoming integral to modern software development workflows.

However, the rapid spread of AI-generated code has raised concerns about new vulnerabilities and misuse. Systematic benchmarks show that LLM outputs often ship with logic errors and latent security flaws[45, 21, 78, 96, 61, 36]. Comparative evaluations reveal that AI suggestions can embed at least as many vulnerabilities as human code[40, 82, 76, 80, 5, 77]. Furthermore, LLMs are susceptible to manipulation [36], including poisoning attacks [91, 11, 54] and prompt injections [49, 95], which can induce the generation of targeted vulnerable code. At the same time, educators warn of an impending wave of AI-driven plagiarism that evades conventional detectors [31, 74, 38, 13, 85, 69, 39], while legal scholars highlight intellectual-property [94, 43, 86, 73] and licence-compliance [88] risks. Robust AI-code detection is therefore critical for secure software supply chains, responsible academic practice, and licence compliance.

To address the challenges of AI-generated code identification, various detection methods have been proposed, leveraging statistical features of code [32], the capabilities of language models [90, 70, 92, 93, 89, 53, 52], and code embedding models [75, 46]. However, evaluations based on existing benchmarks and datasets [75, 59, 14, 62, 58, 87] often fall short in three key aspects. First, they typically cover only a narrow set of programming languages—primarily C++ and Python—while neglecting other widely used languages such as Go and HTML, resulting in limited language diversity compared to real-world software development. Second, most benchmarks rely on open-source LLMs with relatively small model sizes and lower generation quality, or include only a small number of commercial models, leaving a gap between benchmark conditions and real-world usage. Third, most existing datasets lack practical adversarial scenarios, such as paraphrasing [41, 68], which are common in practice and essential for evaluating the robustness of detection systems. Thus, a rigorous benchmark that captures real-world language diversity, modern commercial models, and adversarial scenarios is indispensable for driving meaningful progress in this emerging field.

We introduce ***CodeMirage***, a comprehensive benchmark for evaluating AI-generated code detectors under realistic and adversarial conditions, to solve the three major limitations identified in prior benchmark work. ***CodeMirage*** is constructed from real-world human-written code and enriched with both AI-generated and paraphrased variants produced by a diverse set of state-of-the-art reasoning and non-reasoning LLMs from six major commercial service providers. The paraphrasing techniques are domain-specific and tailored to source code, enabling rigorous evaluation of detector generalization and robustness.

Our key contributions are as follows:

- We present a large-scale, multilingual benchmark for AI-generated code detection, spanning 10 widely used programming languages. The dataset comprises approximately 210,000 samples, including 10,000 human-written code files sourced from GitHub [9], as well as AI-generated and paraphrased counterparts produced by 10 production-level LLMs.
- We design four progressively challenging evaluation configurations with three complementary performance metrics to facilitate rigorous and realistic assessment of detector effectiveness under various real-world scenarios.
- We conduct a comprehensive evaluation of 10 representative detectors across four methodological paradigms using ***CodeMirage***, providing insights into their accuracy, robustness, and generalization across program languages, models, and adversarial settings.

2 Background and Related Work

2.1 Taxonomy of AI-Generated Code Detection Methods

Detecting AI-generated content has been a long-standing challenge in both the natural language [79, 22, 2, 23] and computer vision domains [67, 24, 97, 15, 98], predating even the emergence of large language models (LLMs) [81, 1] and diffusion-based generative models [71, 29]. In contrast, detecting AI-generated source code is a relatively new research direction, emerging primarily in the last two years due to the rapid advancements in the coding capabilities of LLMs [55, 4].

Inspired by traditional statistical-based methods used for AI-generated text detection [64, 33], early approaches for code focus on analyzing surface-level statistical features. For example, Whodunit [32]

Table 1: Comparison between existing AI-generated code benchmarks and our ***CodeMirage***. **Gran.** = granularity (*Func*: function/snippet, *Doc*: whole file). IID = in–distribution; OOD = out-of-distribution. Baseline categories: **Z** (zero-shot detector), **E** (embedding-based detector), **F** (fine-tuning-based detector), **P** (pre-trained LLM + downstream detector). Columns "Open LLMs" and "Comm. LLMs" show whether the dataset includes *any* open-source or commercial generators.

Dataset ↓ Stat.→	**#Lang**	**Gran.**	**IID**	**OOD**	**#Open LLMs**	**#Comm. LLMs**	**Reasoning Model**	**#Human Code**	**#AI Code**	**Adv. Test**	**Quality Check**	**Baseline #/Cat.**
Suh *et al.* [75]	3	Func	✓	✓	1	3	✗	$\sim 3.7k$	$\sim 29.5k$	✗	✗	8 / Z,E,F
Pan *et al.* [59]	1	Func	✓	✓	0	1	✗	$\sim 5k$	$\sim 71k$	✓	✗	5 / Z
AIGCodeSet[14]	1	Func	✓	✗	2	1	✗	$\sim 4.8k$	$\sim 2.9k$	✗	✓	3 / E,F
MAGECODE[62]	3	Doc	✓	✗	0	3	✗	$\sim 81k$	$\sim 45k$	✗	✓	8 / Z
CoDet-M4[58]	3	Func	✓	✓	4	1	✗	$\sim 252k$	$\sim 246k$	✓	✓	6 / F,P
LLMGCode[87]	8	Doc	✓	✗	1	3	✗	$< 1k$	$2k$	✗	✗	10 / Z,F,P
CodeMirage (Ours)	**10**	**Doc**	✓	✓	**4**	**6**	✓	**10*k***	**~200*k***	✓	✓	**10 / Z,E,F,P**

extracts stylometric and complexity-based features from both raw source code and its abstract syntax tree (AST). However, these methods often struggle to distinguish code generated by modern, high-performing LLMs [55, 4, 26, 37], which can mimic human coding styles more closely.

To improve detection effectiveness, recent research has explored more advanced techniques—often leveraging large language models (LLMs) or code embedding models—which can be broadly categorized into the following four methodological paradigms:

Zero-shot Detector. This category of detectors assigns detection confidence scores based on token-level statistics derived from pretrained LLMs, without requiring task-specific fine-tuning. For example, LogRank [22] and Entropy [42] rely on average next-token log-rank and entropy, respectively, to quantify AI-generated token distributions. DetectGPT [51] evaluates the divergence between original and perturbed text using a scoring model, which is a strategy extended in code-specific settings by DetectCodeGPT [70], GPT4Code [92], and AIGC Detector [90], each employing tailored perturbation schemes for code. CR [93] instead measures divergence between original and LLM-rewritten code samples. Binoculars [28] introduces a model-comparison approach, using cross-perplexity between instruction-tuned and non-instruction-tuned LLMs as a detection signal.

Embedding-based Detector. Embedding-based detectors [40] utilize pretrained code embedding models, such as CodeT5+ Embedding [84] and CodeXEmbed [46], to extract high-level semantic representations from either raw source code or abstract syntax trees (ASTs). These embeddings are then fed into lightweight classifiers, *e.g.*, MLP [66], to perform binary classification between human-written and AI-generated code.

Fine-tuning-based Detector. This class of detectors fine-tunes transformer-based models to directly capture discriminative patterns between human-written and AI-generated code. For example, GPTSniffer [52, 53] fine-tunes CodeBERT [19] on labeled code samples to perform binary classification. Other approaches [75] explore different backbone architectures, such as CodeT5+ [84] and RoBERTa [47], to enhance detection performance across varied programming languages and generative models.

Pretrained LLM with Downstream Detector. Unlike zero-shot methods, detectors in this category extract rich semantic representations or statistical signals from pretrained LLMs and train downstream classifiers on these features. For instance, MageCode [62] uses statistical features derived from the hidden state of the classification token in a pretrained CodeT5+ [84] to train a two-layer linear classifier. Some detectors originally developed for text, such as Raidar [48], could be extended to code by comparing metrics between original and LLM-rewritten samples, followed by an XGBoost [8] classifier. BiScope [27] applies a novel bi-directional cross-entropy analysis using pretrained LLMs and feeds the resulting features into a Random Forest [6] classifier.

2.2 Existing AI-generated Code Datasets and Benchmarks

Prior studies [75, 59, 14, 62, 58, 87] has laid important groundwork for building benchmarks to evaluate AI-generated code detectors. As shown in Table 1, several benchmarks introduce valuable contributions: for instance, Suh *et al.* [75] propose a large-scale function-level dataset spanning three programming languages. Pan *et al.* [59] and CoDet-M4 [58] incorporate adversarial perturbations into AI-generated code to test robustness. AIGCodeSet [14] and MAGECODE [62] employ quality

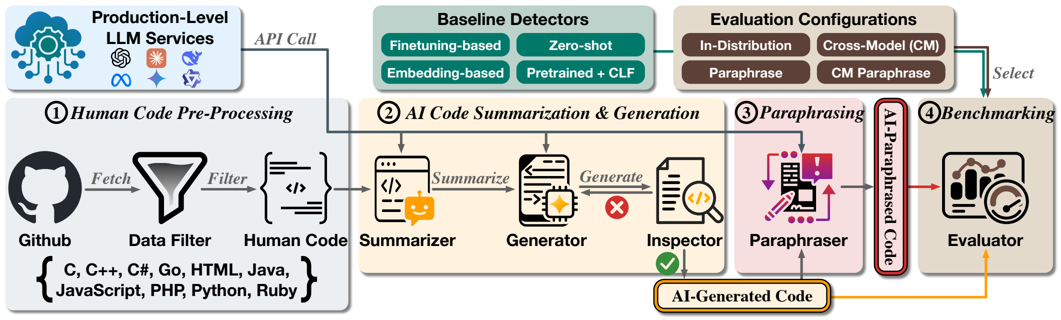

Figure 1: Overview of the ***CodeMirage*** framework. We collect and preprocess human-written code from GitHub, then leverage 10 state-of-the-art LLMs to summarize, generate, and paraphrase code with quality inspection. Finally, ***CodeMirage*** evaluates 10 baseline AI-generated code detectors across four categories under four configurations, covering a wide range of real-world scenarios.

checks during code generation. LLMGCode [87] expands language coverage to eight programming languages. Collectively, these datasets serve as solid foundations for evaluating AI-generated code detectors.

However, each of these benchmarks has notable limitations. Most cover only a small number of programming languages, rely on open-source or less capable LLMs, and none of them leverage latest reasoning models [26, 57, 35]. Furthermore, baseline evaluations in these benchmarks do not comprehensively include all four major categories of detection methods, and only two out of the six existing benchmarks include adversarial testing, which is critical for assessing real-world robustness.

To address these gaps, our proposed benchmark, ***CodeMirage***, includes: (1) code samples across 10 widely used programming languages; (2) outputs from 10 state-of-the-art production-level LLMs, including three reasoning models; (3) both out-of-distribution and adversarial evaluation settings; and (4) baselines covering all four methodological categories of AI-generated code detection.

3 *CodeMirage* Framework

3.1 Benchmark Construction

Human Code Pre-Processing. To construct a comprehensive benchmark of AI-generated and paraphrased code, we begin by sourcing high-quality human-written code samples from the CodeParrot Github-Code-Clean dataset [9], a curated subset of the original Github-Code dataset [10], as shown in Figure 1. This cleaned version filters out overly short snippets, auto-generated files, and samples with excessive alphanumeric characters. The dataset was collected and sanitized in May 2022, prior to the widespread deployment of code LLMs and AI coding agents, ensuring the selected samples are genuinely human-authored. Based on its statistics, we select the ten most commonly used programming languages—C, C++, C#, Go, HTML, Java, JavaScript, PHP, Python, and Ruby—and randomly extract 1,000 code snippets per language. Additional length-based filtering is applied during the sampling to preserve code diversity while ensuring the code remains within a controlled length scale.

Production-Level LLMs. In ***CodeMirage***, we leverage ten production-level LLMs from six leading companies to generate code samples, covering the majority of LLMs commonly used for real-world coding tasks. Among these ten models, four are open-source and three are designed with reasoning capabilities. Specifically, ***CodeMirage*** includes GPT-4o-mini [56], o3-mini [57], Claude-3.5-Haiku [3], Gemini-2.0-Flash [63], Gemini-2.0-Flash-Thinking-Experimental [35], Gemini-2.0-Pro-Experimental [37], DeepSeek-V3 [44], DeepSeek-R1 [26], Llama-3.3-70B [50], and Qwen-2.5-Coder-32B [30]. We access all ten LLMs via API-based services with default temperatures. For additional details on the LLM configurations and generation settings, please refer to Appendix A.

AI Code Summarization. To generate high-quality AI-generated code samples while avoiding direct copying of human-written code, ***CodeMirage*** adopts a text-to-code generation strategy. As the first step, we produce a comprehensive yet concise summary for each human-written code sample. Since these samples are typically full documents—including library imports, class and structure definitions,

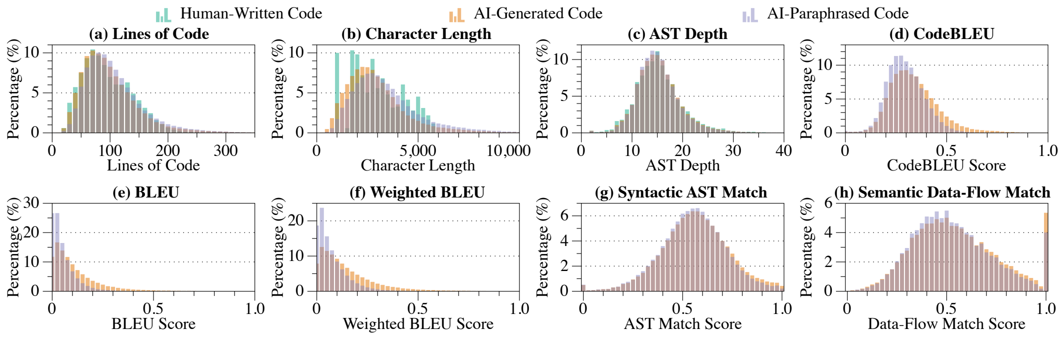

Figure 2: Benchmark statistics of ***CodeMirage***.

and function implementations—we prompt the LLM to extract and summarize key elements such as the *purpose*, *functionality*, *logic overview*, and *key features*, along with the names of relevant *libraries*, *functions*, *classes*, *structures*, and *variables*. *Optional contextual notes* are also included to account for uncommon assumptions or dependencies in the source code. This summary serves as an intermediate representation of the original code, ensuring that the LLM does not access the original human-written implementation during the following code generation step. Full prompts and summary examples are provided in Appendix B.

AI Code Generation. Given the summary of each human-written code sample, ***CodeMirage*** employs multiple production-level LLMs to generate corresponding AI-written code based on the provided description. To align the structural characteristics of the generated code with the original human-written version, we additionally supply the LLMs with metadata such as the line count and total character length. Due to the inherent uncertainty of LLMs, generated code may occasionally deviate from the desired format or content. To further ensure quality, we implement a rule-based inspector that verifies: (1) consistency with the original human-written code's line count and character length, and (2) adequate token-level divergence from the original, enforced by requiring a BLEU [60] score below 0.5 to avoid recitation. Regeneration is forced if any check fails, and samples are discarded after multiple failed attempts. Detailed prompts and generation examples are provided in Appendix C.

AI Code Paraphrasing. Paraphrasing [41, 68] is a widely adopted strategy for evaluating the robustness of AI-generated text detectors under adversarial and real-world conditions. However, in the domain of AI-generated code detection, most existing benchmarks [75, 59, 14, 62, 58, 87] do not incorporate such adversarial testing. Although some text detection studies [48, 27] have included paraphrased code in their evaluations, they rely on generic prompts and a limited number of code samples, constraining both the effectiveness and generality of their paraphrasing evaluation on code. In ***CodeMirage***, we introduce a systematic, domain-specific paraphrasing for code, covering six transformation types: *renaming*, *formatting adjustments*, *logic rewriting and replacement*, *expression variation*, *literal transformations*, and *redundancy insertion*. Detailed rules, prompt designs, and representative examples are provided in Appendix D.

3.2 Benchmark Statistics

CodeMirage spans ten programming languages, each containing 1,000 human-written code samples and 10,000 AI-generated counterparts. For every language, we obtain 1,000 outputs from each of ten production-level LLMs, yielding a 1:10 mapping between every human sample and its LLM-generated variants. Within every 1,000-sample shard (human or AI), we allocate 700 examples for training and 300 for testing.

We present four structural and semantic metrics of the dataset in Figure 2: lines of code (a), character length (b), AST depth (c), and CodeBLEU [65] score (d). The first three metrics reflect the overall structural characteristics of the code and show close resemblance between human-written and AI-generated samples. This similarity implies that naive statistical classifiers would struggle to detect AI-generated code using basic code features.

Figure 2 (d) reports the CodeBLEU score, a composite metric calculated as:

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weighted} + \gamma \cdot Match_{AST} + \delta \cdot Match_{DF}, \quad (1)$$

where each component is equally weighted with $\alpha = \beta = \gamma = \delta = 0.25$ by default. The median CodeBLEU score for AI-generated code is approximately 0.3, consistent with prior observations in text-to-code generation [16, 17, 18]. Paraphrased code yields slightly lower scores due to deliberate perturbations in both code format and structure.

To further analyze ***CodeMirage***'s code quality, we decompose the CodeBLEU score into its four subcomponents in Figure 2 (e)–(h). Both AI-generated and AI-paraphrased code show relatively low BLEU [60] and weighted BLEU [65] scores, indicating limited n-gram overlap with their human counterparts. While the syntactic AST match and semantic data-flow [25] match scores of AI code exceed 0.5 on average, suggesting that despite token-level divergence, both AI-generated and AI-paraphrased code maintains a fair level of syntactic and semantic consistency with human-written code. More detailed benchmark statistics are presented in Appendix E.

3.3 Baseline Detectors

We select ten state-of-the-art detectors spanning four categories. **Zero-shot detectors**: *LogRank* [22], *Entropy* [22, 42], and *Binoculars* [28], which rely on token-rank or entropy-related features without training. **Embedding-based detectors**: following existing studies [75], we extract representations with the *CodeXEmbed-2B* model [46] from either raw source code or its abstract-syntax tree (AST) and train a lightweight random forest [6] classifier. **Fine-tuned detectors**: we include *GPTSniffer* [53, 52], a variant built on the latest *CodeT5+* backbone [84], and a *RoBERTa* detector [47], with each fine-tuned on our training corpus. **Pretrained-LLM with downstream detector**: *Raidar* [48] and *BiScope* [27], extracting features via rewriting [48] and bi-directional cross entropy [27]. More details of the baseline detectors are presented in Appendix F.

3.4 Evaluation Metrics

To thoroughly assess the performance of the baseline detectors in different scenarios, we employ three evaluation metrics in our experiments, including the *F1 score*, *TPR@FPR=10%*, and *TPR@FPR=1%*. The *F1 score* balances precision and recall, providing an overall measure of detection accuracy without favoring AI-generated or human-written code samples. For each detector, we first identify the optimal decision threshold and then report its corresponding *F1 score*. The metric *TPR@FPR=10%* reports the true positive rate (TPR) when the false positive rate (FPR) is limited to 10%, representing scenarios that can tolerate a moderate number of false alarms. Conversely, *TPR@FPR=1%* measures the TPR at an FPR of only 1%, which is essential for applications where even a small fraction of false positives is unacceptable.

3.5 Evaluation Configurations

In ***CodeMirage***, we include four evaluation configurations to thoroughly assess baseline detectors under various real-world scenarios, including the in-distribution configuration and three out-of-distribution configurations (paraphrase configuration, cross-model configuration, and cross-model paraphrase configuration). We omit the cross language configuration because programming language can be easily identified; thus, detectors can be trained separately for each language.

In-Distribution Configuration. This configuration evaluates the in-distribution stability of each detector in multiple LLMs and programming languages. For each language, we pair the human-written training set with the training samples produced by a single LLM, train the detector on this combined data, and determine the optimal decision threshold. We then test the detector on the human-written test set together with the test samples generated by the same LLM.

Paraphrase Configuration. This setting evaluates each detector's out-of-distribution performance when the AI-generated code is adversarially paraphrased. Specifically, we train the detector and select its optimal threshold same as in the in-distribution configuration, but we test on *paraphrased* code produced by the same LLM that generated the original samples.

Cross-Model Configuration. This setting evaluates detector's robustness against unseen LLMs. For each programming language, we train the detector and choose its optimal threshold on a training set consisting of human-written samples and AI-generated samples from a *single* LLM. We then test

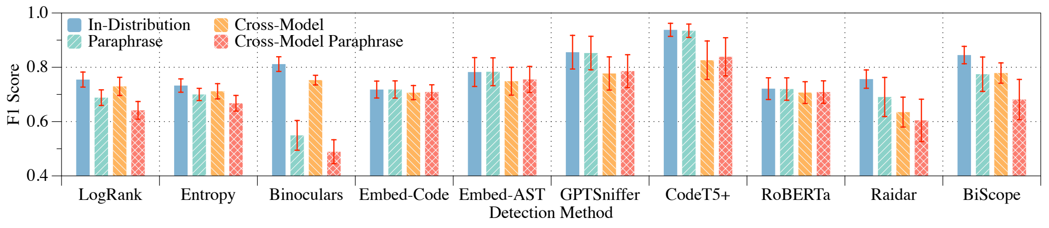

Figure 3: **Comparison Between Evaluation Configurations and Detectors.** The bar chart presents the average F1 scores of baseline detectors across all the programming languages and LLMs.

the detector on human test samples paired with AI-generated samples from all *other* LLMs. The detector's scores on these unseen-model test sets are averaged to yield the overall cross-model result.

Cross-Model Paraphrase Configuration. This scenario mirrors real-world conditions in which code samples are both generated by unseen LLMs and subsequently paraphrased. We adopt the testing procedure of the cross-model configuration, but pair human test samples with paraphrased test samples produced by the other LLMs. The detector's average score over all such paraphrased, unseen-model test sets is reported as the cross-model paraphrase result.

4 Evaluation Results and Insights

We conduct an extensive evaluation using ***CodeMirage*** in various scenarios and summarize the observations into nine findings. We present representative processed results in the main text and include the full experimental results in Appendix H.

4.1 Comparison Between Evaluation Configurations and Detectors

We first evaluate the performance of the various detectors under four distinct configurations 3.5. The results are presented in Figure 3, where the x-axis lists the detectors and the y-axis represents the F1 score. Each bar corresponds to a specific evaluation configuration. Notably, to ensure a fair and unbiased comparison, each bar reflects the average F score obtained across ten programming languages and ten LLMs, with error bars indicating one standard deviation.

Finding 1: *In-distribution testing consistently outperforms all out-of-distribution scenarios.*

This is intuitive and reasonable given the shared distribution between training and test sets. Under out-of-distribution settings, cross-model testing yields a larger performance drop than paraphrasing in most cases, since paraphrasing leverages the same LLM and thus incurs a smaller distribution shift than code generation by a different LLM. However, some corner cases, e.g., LogRank and Binoculars, deviate from this trend. As zero-shot methods, they are particularly sensitive to token-level features, and paraphrasing induces greater token variance than cross-model evaluation.

Furthermore, different detection methods exhibit varying performance. According to subsection 3.3, these methods fall into four categories.

Finding 2: *Fine-tuning-based methods outperforms other types.*

Fine-tuned detectors, e.g., GPTSniffer and CodeT5+, lead the pack. Zero-shot approaches, e.g., LogRank and Entropy, perform poorest, which makes sense given their limited feature extraction when confronted with the complexity of code. Embedding-based detectors, e.g., Embed-Code and Embed-AST, sit in the middle but impressively maintain stable accuracy even under out-of-distribution evaluation, thanks to their reliance on code representations that generalize across LLMs. Pretrained LLMs paired with downstream classifiers, e.g., Raidar and BiScope, match embedding methods in-distribution but suffer a larger drop on out-of-distribution tests, reflecting subtle shifts in the features they extract across different models and paraphrased inputs.

Finding 3: *Fine-tuning approaches using backbone LLMs pre-trained on larger code corpora achieve superior performance.*

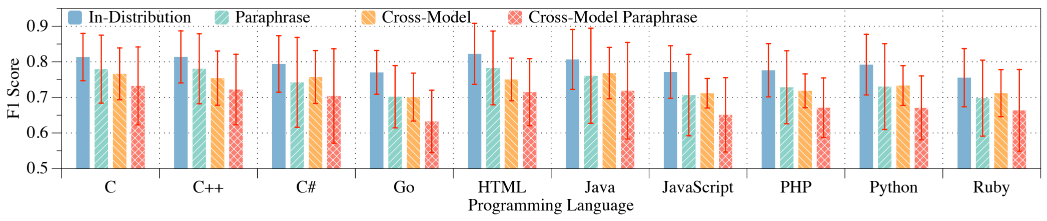

Figure 4: **Comparison Between Different Programming Languages.** The bar chart presents the average F1 scores of baseline detectors on different programming languages across LLMs.

Performance varies across fine-tuning methods. For example, CodeT5+ slightly outperforms GPT-Sniffer, and both surpass RoBERTa. This gap reflects their pre-training corpora: GPTSniffer's CodeBERT backbone is trained on six programming languages, whereas CodeT5+'s backbone covers nine. In contrast, RoBERTa is pretrained solely on natural-language text. Consequently, backbones exposed to more and broader code samples exhibit superior coding proficiency, and hence better detection capability.

Finding 4: *Fine-tuning–based detectors are prone to overfitting.*

We also observe that fine-tuning–based methods (e.g., GPTSniffer and CodeT5+) exhibit a larger performance drop from in-distribution to cross-model evaluations than other approaches. This is likely due to their overfitting tendencies and should be taken into account in real-world deployments.

Finding 5: *ASTs provide a superior feature representation compared to raw source code.*

Two embedding–based detectors demonstrate comparable performance, with Embed-AST marginally outperforming Embed-Code. This suggests that AST-based embeddings capture the program's syntactic hierarchy and semantic relationships, e.g., control flow and data dependencies, more effectively than raw code tokens, making them more robust to superficial variations like naming or formatting.

4.2 Comparison Between Different Programming Languages

We evaluate detection performance across ten programming languages using ***CodeMirage***. The results are shown in Figure 5, where the x-axis lists the languages and the y-axis denotes the F1 score. To minimize bias, each bar aggregates results from experiments with all ten LLMs and ten detectors. Its height indicates the average F1 score, and the error bars represent one standard deviation.

Finding 6: *Detection is Consistent across Programming Languages, with Common Languages Performing Slightly Better.*

We observe only slight performance differences among languages, with similar patterns across evaluation configurations. Notably, less common languages exhibit marginally lower performance. For example, C++ achieves higher F1 scores than Go or Ruby. This discrepancy arises because several detection methods, e.g., Biscope [27] and Raidar [48], rely on pre-trained LLMs for feature extraction. These models are pre-trained on large online corpora containing more examples of common languages (e.g., C++) than atypical ones (e.g., Go), resulting in stronger representations for the former. Hence detection performances are better detection on those common languages.

4.3 Comparison Between Different LLMs

We evaluate the detection performance of code generated by different LLMs, with results shown in Figure 5. The x-axis represents the generative models, while the y-axis indicates the F1 score. Each bar color corresponds to one of four evaluation settings.

Finding 7: *Detection performance is generally similar across LLMs, with GPT and Llama showing slightly higher scores.*

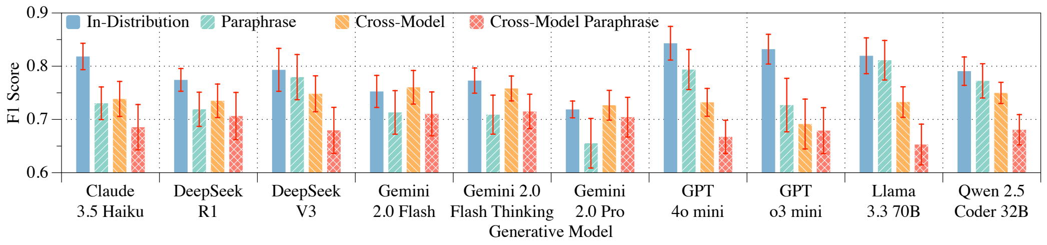

Figure 5: **Comparison Between Different LLMs.** The bar chart shows the average F1 scores of baseline detectors on different LLMs across programming languages.

Among all models, GPT-4o mini achieves the highest F1 scores, particularly under the In-Distribution and Paraphrase settings, suggesting that its code style is more consistent or distinctive, making detection easier. Claude 3.5 Haiku and Llama 3.3 70B also demonstrate strong performance, especially under In-Distribution, likely due to their more recognizable or less variable code patterns. In contrast, Cross-Model Paraphrase consistently yields the lowest F1 scores (around 0.65–0.7), highlighting it as the most challenging scenario for detection. Models such as Gemini 2.0 Pro and Qwen 2.5 Coder 32B exhibit lower detectability across settings, especially under paraphrased or cross-model conditions, indicating that their outputs may be more diverse or stylistically more similar to human's, thereby reducing their distinctiveness.

Finding 8: *Reasoning models exhibit a larger performance drop after paraphrasing.*

We observe that for non-reasoning models (DeepSeek V3, GPT4o mini, Llama 3.3 70B, and Qwen 2.5 Coder 32B), paraphrasing has minimal impact on performance. In contrast, reasoning models (e.g., GPT o3 mini) suffer a more pronounced decline. This likely stems from their stronger comprehension abilities: they better interpret paraphrased inputs and adjust outputs to match human-style reasoning, making any deviations more evident after paraphrasing.

4.4 Comparison Between Different Evaluation Metrics

In previous experiments, we mainly use F1 score, which is a threshold-dependent measure that balances precision and recall, but F1 can be misleading in real-world detection tasks. As it gives equal weight to false positives and false negatives and depends on a single decision threshold, it often fails to reflect performance in imbalanced settings or under strict false-alarm constraints. By contrast, reporting the true positive rate at low false-positive rates directly measures how many genuine positives the model catches when false alarms must be kept to a minimum [7]. Therefore, we introduce two additional metrics, i.e., TPR@FPR=10% and 1%, to better assess detector practicality.

Finding 9: *There is a significant gap between laboratory evaluations and practical use.*

Results in Appendix G indicates that despite decent F1 scores, all detectors suffer a dramatic drop in true-positive rate once the false-positive rate is constrained, showing that they fail to catch enough positives under realistic, low-alarm requirements and are therefore impractical.

5 Conclusion

In this paper, we introduce ***CodeMirage***, a comprehensive and large-scale benchmark for AI-generated code detection, consisting of 10 widely used programming languages and approximately 210,000 samples in total. The dataset includes human-written code, as well as AI-generated and paraphrased variants created using 10 state-of-the-art production-level LLMs, including three recent reasoning models, with quality control to ensure reliability. We evaluate 10 representative detectors spanning four methodological categories and provide extensive analysis from multiple perspectives, revealing key strengths and limitations of each approach. We believe the breadth and depth of ***CodeMirage*** offer a strong foundation for advancing the development of more robust and generalizable detectors.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Arslan Akram. An empirical study of ai generated text detection tools. *arXiv preprint arXiv:2310.01423*, 2023.

[3] Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku, 2024.

[4] Anthropic. Claude 3.7 Sonnet and Claude Code, 2025.

[5] Owura Asare, Meiyappan Nagappan, and Nirmal Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*, 28(6):129, 2023.

[6] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[7] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. Membership inference attacks from first principles. In *2022 IEEE symposium on security and privacy (SP)*, pages 1897–1914. IEEE, 2022.

[8] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, Mu Li, Junyuan Xie, Min Lin, Yifeng Geng, Yutian Li, Jiaming Yuan, and David Cortes. *xgboost: Extreme Gradient Boosting*, 2025. R package version 3.0.1.1.

[9] CodeParrot. Github code clean dataset, 2022.

[10] CodeParrot. Github code dataset, 2022.

[11] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In *IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 280–292, 2024.

[12] Cursor. Cursor: The AI Code Editor, 2023.

[13] Nassim Dehouche. Plagiarism in the age of massive generative pre-trained transformers (gpt-3). *Ethics in Science and Environmental Politics*, 21:17–23, 2021.

[14] Basak Demirok and Mucahid Kutlu. Aigcodeset: A new annotated dataset for ai generated code detection. *arXiv preprint arXiv:2412.16594*, 2024.

[15] Brian Dolhansky, Joanna Bitton, Ben Pflaum, Jikuo Lu, Russ Howes, Menglin Wang, and Cristian Canton Ferrer. The deepfake detection challenge (dfdc) dataset. *arXiv preprint arXiv:2006.07397*, 2020.

[16] Yihong Dong, Ge Li, and Zhi Jin. Codep: grammatical seq2seq model for general-purpose code generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 188–198, 2023.

[17] Jessica López Espejel, Mahaman Sanoussi Yahaya Alassan, Walid Dahhane, and El Hassane Ettifouri. Jacotext: a pretrained model for java code-text generation. *arXiv preprint arXiv:2303.12869*, 2023.

[18] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *https://arxiv.org/abs/2002.08155*, 2020.

[20] Friedman, Nat. Introducing GitHub Copilot: your AI pair programmer, 2022.

[21] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 761–773, 2023.

[22] Sebastian Gehrmann, Hendrik Strobelt, and Alexander M Rush. Gltr: Statistical detection and visualization of generated text. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[23] Soumya Suvra Ghosal, Souradip Chakraborty, Jonas Geiping, Furong Huang, Dinesh Manocha, and Amrit Bedi. A survey on the possibilities & impossibilities of ai-generated text detection. *Transactions on Machine Learning Research (TMLR)*, 2023.

[24] David Güera and Edward J Delp. Deepfake video detection using recurrent neural networks. In *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6, 2018.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021.

[26] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[27] Hanxi Guo, Siyuan Cheng, Xiaolong Jin, Zhuo Zhang, Kaiyuan Zhang, Guanhong Tao, Guangyu Shen, and Xiangyu Zhang. Biscope: Ai-generated text detection by checking memorization of preceding tokens. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:104065–104090, 2024.

[28] Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Hamid Kazemi, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Spotting llms with binoculars: Zero-shot detection of machine-generated text. In *International Conference on Machine Learning (ICML)*, 2024.

[29] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:6840–6851, 2020.

[30] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[31] James Hutson. Rethinking plagiarism in the era of generative ai. *Journal of Intelligent Communication*, 3(2):20–31, 2024.

[32] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M Atlee, and Mei Nagappan. Whodunit: Classifying code as human authored or gpt-4 generated-a case study on codechef problems. In *International Conference on Mining Software Repositories (MSR)*, pages 394–406, 2024.

[33] Daphne Ippolito, Daniel Duckworth, Chris Callison-Burch, and Douglas Eck. Automatic detection of generated text is easiest when humans are fooled. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1808–1822, 2020.

[34] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.

[35] Patrick Kane. Access the latest 2.0 experimental models in the gemini app., 2025.

[36] Sabrina Kaniewski, Dieter Holstein, Fabian Schmidt, and Tobias Heer. Vulnerability handling of ai-generated code-existing solutions and open challenges. In *Conference on AI, Science, Engineering, and Technology (AIxSET)*, pages 145–148, 2024.

[37] Koray Kavukcuoglu. Gemini 2.0 is now available to everyone, 2025.

[38] Mustafa Ali Khalaf. Does attitude towards plagiarism predict aigiarism using chatgpt? *AI and Ethics*, 5(1):677–688, 2025.

[39] Mohammad Khalil and Erkan Er. Will chatgpt g et you caught? rethinking of plagiarism detection. In *International Conference on Human-Computer Interaction*, pages 475–487, 2023.

[40] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? In *IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2445–2451. IEEE, 2023.

[41] Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Wieting, and Mohit Iyyer. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[42] Thomas Lavergne, Tanguy Urvoy, and François Yvon. Detecting fake content with relative entropy scoring. In *Proceedings of the International Conference on Uncovering Plagiarism, Authorship and Social Software Misuse (PAN)*, volume 377, pages 27–31, 2008.

[43] Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. Protecting intellectual property of large language model-based code generation apis via watermarks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2336–2350, 2023.

[44] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[45] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:21558–21572, 2023.

[46] Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codex-embed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024.

[47] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[48] Chengzhi Mao, Carl Vondrick, Hao Wang, and Junfeng Yang. Raidar: generative ai detection via rewriting. In *International Conference on Learning Representations (ICLR)*, 2024.

[49] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot. In *International Conference on Software Engineering (ICSE)*, pages 2149–2160, 2023.

[50] Meta. Llama 3.3: Model cards & prompt formats, 2024.

[51] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International Conference on Machine Learning (ICML)*, pages 24950–24962. PMLR, 2023.

[52] Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. Is this snippet written by chatgpt? an empirical study with a codebert-based classifier. *arXiv preprint arXiv:2307.09381*, 2023.

[53] Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *Journal of Systems and Software*, 214:112059, 2024.

[54] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. Poisoned chatgpt finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned ai models. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1141–1159, 2024.

[55] OpenAI. Introducing ChatGPT, 2022.

[56] OpenAI. Gpt-4o mini: advancing cost-efficient intelligence, 2024.

[57] OpenAI. Openai o3-mini: Pushing the frontier of cost-effective reasoning, 2025.

[58] Daniil Orel, Dilshod Azizov, and Preslav Nakov. Codet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. *arXiv preprint arXiv:2503.13733*, 2025.

[59] Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. Assessing ai detectors in identifying ai-generated code: Implications for education. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 1–11, 2024.

[60] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318, 2002.

[61] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 754–768, 2022.

[62] Hung Pham, Huyen Ha, Van Tong, Dung Hoang, Duc Tran, and Tuyen Ngoc Le. Magecode: Machine-generated code detection method using large language models. *IEEE Access*, 2024.

[63] Sundar Pichai, Demis Hassabis, and Koray Kavukcuoglu. Introducing gemini 2.0: our new ai model for the agentic era, 2024.

[64] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48. Citeseer, 2003.

[65] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[66] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[67] Andreas Rössler, Davide Cozzolino, Luisa Verdoliva, Christian Riess, Justus Thies, and Matthias Nießner. Faceforensics: A large-scale video dataset for forgery detection in human faces. *arXiv preprint arXiv:1803.09179*, 2018.

[68] Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. Can ai-generated text be reliably detected? *arXiv preprint arXiv:2303.11156*, 2023.

[69] Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. Automated detection of ai-obfuscated plagiarism in modeling assignments. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 297–308, 2024.

[70] Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *International Conference on Software Engineering (ICSE)*, pages 51–62, 2025.

[71] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning (ICML)*, pages 2256–2265. pmlr, 2015.

[72] Stack Overflow. 2024 Stack Overflow Developer Survey, 2024.

[73] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A Heymann, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. Developer perspectives on licensing and copyright issues arising from generative ai for coding. *arXiv preprint arXiv:2411.10877*, 2024.

[74] Aiste Steponenaite and Basel Barakat. Plagiarism in ai empowered world. In *International Conference on Human-Computer Interaction*, pages 434–442, 2023.

[75] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting ai-generated source code: How far are we? In *International Conference on Software Engineering (ICSE)*, 2025.

[76] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):1–48, 2025.

[77] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C Cordeiro. How secure is ai-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2):1–42, 2025.

[78] Rebeka Tóth, Tamas Bisztray, and László Erdődi. Llms in web development: Evaluating llm-generated php code unveiling vulnerabilities and limitations. In *International Conference on Computer Safety, Reliability, and Security*, pages 425–437, 2024.

[79] Adaku Uchendu, Zeyu Ma, Thai Le, Rui Zhang, and Dongwon Lee. Turingbench: A benchmark environment for turing test in the age of neural text generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2001–2016, 2021.

[80] Jaideep Vaidya and Hafiz Asif. A critical look at ai-generate software: Coding with the new ai tools is both irresistible and dangerous. *IEEE Spectrum*, 60(7):34–39, 2023.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[82] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*, 2024.

[83] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025.

[84] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. Codet5+: Open code large language models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1069–1088, 2023.

[85] Yunkai Xiao, Soumyadeep Chatterjee, and Edward Gehringer. A new era of plagiarism the danger of cheating using ai. In *International Conference on Information Technology Based Higher Education and Training (ITHET)*, pages 1–6, 2022.

[86] Jialiang Xu, Shenglan Li, Zhaozhuo Xu, and Denghui Zhang. Do llms know to respect copyright notice? In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 20604–20619, 2024.

[87] Jinwei Xu, He Zhang, Yanjin Yang, Zeru Cheng, Jun Lyu, Bohan Liu, Xin Zhou, Lanxin Yang, Alberto Bacchelli, Yin Kia Chiam, et al. Investigating efficacy of perplexity in detecting llm-generated code. *arXiv preprint arXiv:2412.16525*, 2024.

[88] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. Licoeval: Evaluating llms on license compliance in code generation. *arXiv preprint arXiv:2408.02487*, 2024.

[89] Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–31, 2025.

[90] Zhenyu Xu and Victor S Sheng. Detecting ai-generated code assignments using perplexity of large language models. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 38, pages 23155–23162, 2024.

[91] Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. An {LLM-Assisted}{Easy-to-Trigger} backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *USENIX Security Symposium (USENIX Security)*, pages 1795–1812, 2024.

[92] Xianjun Yang, Kexun Zhang, Haifeng Chen, Linda Petzold, William Yang Wang, and Wei Cheng. Zero-shot detection of machine-generated codes. *arXiv preprint arXiv:2310.05103*, 2023.

[93] Tong Ye, Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 39, pages 968–976, 2025.

[94] Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. Codeipprompt: intellectual property infringement assessment of code language models. In *International Conference on Machine Learning (ICML)*, pages 40373–40389, 2023.

[95] Binqi Zeng, Quan Zhang, Chijin Zhou, Gwihwan Go, Yu Jiang, and Heyuan Shi. Inducing vulnerable code generation in llm coding assistants. *arXiv preprint arXiv:2504.15867*, 2025.

[96] Ying Zhang, Wenjia Song, Zhengjie Ji, Na Meng, et al. How well does llm generate security tests? *arXiv preprint arXiv:2310.00710*, 2023.

[97] Mingjian Zhu, Hanting Chen, Qiangyu Yan, Xudong Huang, Guanyu Lin, Wei Li, Zhijun Tu, Hailin Hu, Jie Hu, and Yunhe Wang. Genimage: A million-scale benchmark for detecting ai-generated image. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:77771–77782, 2023.

[98] Bojia Zi, Minghao Chang, Jingjing Chen, Xingjun Ma, and Yu-Gang Jiang. Wilddeepfake: A challenging real-world dataset for deepfake detection. In *Proceedings of the 28th ACM international conference on multimedia*, pages 2382–2390, 2020.

To further support and validate our ***CodeMirage*** benchmark, we provide the following supplementary materials:

- Appendix A: Detailed descriptions of the production-level LLMs used in ***CodeMirage*** and their corresponding generation settings.
- Appendix B: Prompts used in the code summarization phase and representative examples.
- Appendix C: Prompts used in the code generation phase and representative examples.
- Appendix D: Domain-specific transformation rules, prompts, used in the code paraphrasing phase with representative examples.
- Appendix E: Comprehensive statistics and distributions of the ***CodeMirage*** dataset.
- Appendix F: Detailed descriptions of the baseline detectors included in our evaluation.
- Appendix G: Supplementary results based on *TPR@FPR* metrics.
- Appendix H: Extended and detailed experimental results across all evaluation settings.
- Appendix I: Additional discussion on the limitations and future improvement directions.

A Details of Generative Models and Generation Settings

Table 2: Detailed configurations of the production-level LLMs used in ***CodeMirage***.

LLM Name	API / Model Path	Hyper-Parameter
Claude-3.5-Haiku [3]	`Anthropic/claude-3-5-haiku-20241022`	temperature = 1.0
GPT-4o-mini [56]	`OpenAI/gpt-4o-mini-2024-07-18`	temperature = 1.0

GPT-o3-mini [57]	`OpenAI/o3-mini-2025-01-31`	temperature = 1.0
		reasoning_effort = medium

Gemini-2.0-Flash [63]	`Google/gemini-2.0-flash`	temperature = 1.0
Gemini-2.0-Flash-Thinking [35]	`Google/gemini-2.0-flash-thinking-exp-01-21`	temperature = 1.0
Gemini-2.0-Pro [37]	`Google/gemini-2.0-pro-exp-02-05`	temperature = 1.0
DeepSeek-V3 [44]	`deepseek-ai/DeepSeek-V3`	temperature = 1.0
DeepSeek-R1 [26]	`deepseek-ai/DeepSeek-R1`	temperature = 1.0
Llama-3.3-70B [50]	`meta-llama/Llama-3.3-70B-Instruct`	temperature = 0.6
Qwen-2.5-Coder-32B [30]	`Qwen/Qwen2.5-Coder-32B-Instruct`	temperature = 0.7

In ***CodeMirage***, we adopt ten widely used production-level LLMs from six leading AI companies, including three reasoning models. Detailed configurations and generation settings for these models are presented in Table 2. For key generation hyper-parameters such as `temperature` and `reasoning_effort`, we use either default values or officially recommended settings to reflect realistic usage. Importantly, we avoid setting `temperature` to zero, as doing so would produce overly deterministic outputs that are easier to detect. Instead, we adopt general-purpose settings for high-quality while more diverse and less predictable code generation.

B Additional Details of AI Code Summarization

To generate high-quality and representative summaries that comprehensively describe the characteristics of a code sample while preventing the leakage of concrete code, we design a structured summarization prompt covering eight key aspects. We then prompt the LLMs to act as summarizers, generating summaries based on the input code file using this carefully crafted prompt. The full summarization prompt used in ***CodeMirage*** is as follows:

Summarization Prompt

Analyze the provided code snippet and generate a concise and informative description of its functionality, purpose, and design. Avoid directly including or mirroring the given code. Focus on abstracting the logic, functionality, and intent.

Follow the output format:

1. **Purpose:** A high-level summary of what the code is intended to achieve.

2. **Functionality:** Describe the main tasks performed by the code, including inputs, outputs, and their roles, without referencing exact code or variable names.
3. **Logic Overview:** Explain the key logic, algorithms, or patterns conceptually, avoiding specific code structures or syntax.
4. **Key Features:** Highlight unique approaches or techniques without mentioning explicit implementation details.
5. **Key Libraries:** Include the key libraries used in the code snippet.
6. **Key Functions:** Include the key function names used in the code snippet.
7. **Key Class, Structure, Variable:** Include the key class, structure, and variable names used in the code snippet.
8. **Contextual Notes (optional):** Any assumptions, dependencies, or inferred background information.

Below, we present a concrete Python code example with the summary generated by GPT-o3-mini [57], illustrating the consistency and descriptive quality of the summarization process in ***CodeMirage***.

Python Summary Example

Human-Written Code

```
import logging
import voluptuous as vol
from homeassistant.components.binary_sensor import
    (BinarySensorDevice, PLATFORM_SCHEMA, SENSOR_CLASSES_SCHEMA)
from homeassistant.components import enocean
from homeassistant.const import (CONF_NAME, CONF_ID, CONF_SENSOR_CLASS)
import homeassistant.helpers.config_validation as cv

_LOGGER = logging.getLogger(__name__)
DEPENDENCIES = ['enocean']
DEFAULT_NAME = 'EnOcean binary sensor'

PLATFORM_SCHEMA = PLATFORM_SCHEMA.extend({
    ...
})

def setup_platform(hass, config, add_devices, discovery_info=None):
    ...
class EnOceanBinarySensor(enocean.EnOceanDevice, BinarySensorDevice):
    def __init__(self, dev_id, devname, sensor_class):
        ...
    @property
    def name(self):
        ...
    @property
    def sensor_class(self):
        ...
    def value_changed(self, value, value2):
        ...
...
```

GPT-o3-mini Generated Summary

1. **Purpose:** The code integrates binary sensors that communicate using an EnOcean protocol into a larger home automation framework. It enables sensor events to be received, processed, and dispatched within the automation system.
2. **Functionality:** The implementation defines a platform component for binary sensors. It validates configuration inputs, creates ...
3. **Logic Overview:** The code establishes a configuration schema to ensure correct setup parameters. It then creates a ...
4. **Key Features:** Validates configuration using a dedicated schema to ensure data consistency. Leverages inheritance to ...
5. **Key Libraries:** Logging, Voluptuous for configuration validation, ...
6. **Key Functions:** ... An event handler method that processes changes in sensor values and triggers events ...
7. **Key Class, Structure, Variable:** A sensor class that ... , Variables for device identification, sensor naming, ...
8. **Contextual Notes:** The design assumes integration within a home automation ecosystem where components communicate via ...

For more examples and summaries across different programming languages and generative models, please refer to our ***CodeMirage*** dataset.

C Additional Details of AI Code Generation

After generating a comprehensive yet concise summary of the human-written code, we prompt LLMs to produce corresponding AI-generated versions. To preserve structural consistency, we explicitly include the statistical properties (*e.g.*, line of code and character length) of the original code in the prompt. The complete code generation prompt used in ***CodeMirage*** is presented below:

Code Generation Prompt

Generate a {Language} code snippet with around {Line_of_Code} lines and {Char_Length} character length following the description. Only output the code, no any other information.

{Summary}

The generated code is then passed through a rule-based inspector, as described in subsection 3.1, to filter out overly short, overly long, or null outputs. Below, we show a Python code example generated by GPT-o3-mini [57], based on the summary presented in Appendix B:

AI-Generated Python Example

GPT-o3-mini Code Generation

```python
import logging
import voluptuous as vol
from homeassistant.const import CONF_NAME, CONF_DEVICE_ID
...
DOMAIN = "enocean_binary_sensor"
CONF_SENSOR_TYPE = "sensor_type"
...
class EventBus:
    def fire(self, event_type, event_data):
        ...
class EnOceanDevice:
    def __init__(self, device_id):
        ...
    def send_message(self, message):
        ...
...
class BinarySensorDevice:
    ...
class EnOceanBinarySensor(EnOceanDevice, BinarySensorDevice):
    ...
...
if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    hass = Hass()
    config = {
        CONF_DEVICE_ID: "enocean_001",
        ...
    }
    setup_platform(hass, config)
```

We observe that though the detailed implementation of the AI-generated code differs from the human-written version, it employs the same key libraries and similar variables to achieve comparable functionality, illustrating the quality of the AI-generated code. More examples are presented in ***CodeMirage*** dataset.

D Additional Details of AI Code Paraphrasing

We further evaluate the robustness of detectors under adversarial paraphrasing, a prevalent and practical challenge in real-world applications. Our adversarial setup is grounded in prompt-based paraphrasing, which represents one of the most accessible and commonly used evasion strategies in practice. Unlike prior works on natural language paraphrasing [41, 68], which rely on generic and concise prompts, we propose a code-specific paraphrasing prompt tailored to the programming domain. This prompt incorporates seven transformation strategies specifically designed for source code. The full prompt is provided below:

Paraphrasing Prompt

Transform the given code to bypass AI-generated code detectors by mimicking human coding styles. Make structural, stylistic, and naming changes while preserving exact functionality. Apply the following:

1. Rename variables, functions, and classes with meaningful, human-like names.
2. Adjust formatting (indentation, spacing, line breaks) and reorganize imports logically.
3. Rewrite logic with equivalent constructs (*e.g.*, refactor loops, conditionals).
4. Replace common standard library functions with alternatives or custom helpers.
5. Vary expressions (*e.g.*, 'a + b' → 'sum([a, b])') and simplify inline logic into reusable functions.
6. Transform literals (*e.g.*, single vs. double quotes, '10' → '0xA') and reorder independent code blocks.
7. Add subtle redundancies (*e.g.*, intermediate variables) to resemble human-written code.

Keep the line of code and character length similar to the original code and only output the transformed code. No any other additional text and information.

We then apply the paraphrasing prompt to the AI-generated code sample and obtain its paraphrased version. Below, we present the paraphrased code example produced by GPT-o3-mini [57], based on the AI-generated example provided in Appendix C.

AI-Paraphrased Python Example

GPT-o3-mini Paraphrased Code

```
import logging
import voluptuous as schema_validator

from homeassistant.const import CONF_NAME, CONF_DEVICE_ID
import homeassistant.helpers.config_validation as config_val

logger = logging.getLogger(__name__)
SENSOR_DOMAIN = 'enocean_binary_sensor'
KEY_SENSOR_KIND = 'sensor_type'
...
SensorConfigSchema = schema_validator.Schema({
    ...
})

class SimulatedEventBus:
    def trigger(self, event_category, particulars):
        ...
    ...

class BaseEnOceanDevice:
    def __init__(self, dev_identifier):
        ...
    def dispatch_message(self, msg_payload):
        ...
class BasicBinarySensor:
    ...
class EnOceanSensorClient(BaseEnOceanDevice, BasicBinarySensor):
    ...
...
if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    simulated_hass = SimulatedHomeAssistant()
    input_configuration = {
        ...
    }
    initialize_platform(simulated_hass, input_configuration)
```

Compared to the original AI-generated code, the paraphrased version uses different aliases for imported libraries, introduces redundant classes and variables, and modifies function and class names with different implementations, while preserving the overall program functionality. Additional examples can be found in the full ***CodeMirage*** dataset.

Table 3: ***CodeMirage***'s data quantity statistics across different LLMs and programming languages.

LLM	Paraphrase	Python	Java	JavaScript	C++	C	C#	Go	Ruby	PHP	HTML
Human	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Claude-3.5-Haiku	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
DeepSeek-R1	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	999	1,000	999	1,000	1,000	1,000	1,000
DeepSeek-V3	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Gemini-2.0-Flash	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Gemini-2.0-Flash-Thinking	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Gemini-2.0-Pro	✗	1,000	1,000	1,000	1,000	1,000	998	1,000	1,000	998	999
	✓	1,000	1,000	1,000	1,000	1,000	998	1,000	1,000	998	999
GPT-4o-mini	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Llama-3.3-70B	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
GPT-o3-mini	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Qwen-2.5-Coder-32B	✗	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
	✓	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000

E Additional Statistics of *CodeMirage*'s Dataset

In subsection 3.2, we present the detailed data quality statistics of the ***CodeMirage*** dataset across eight metrics. In this section, we further provide data quantity statistics, as shown in Table 3. For both human-written code and most AI-generated code, we collect or craft 1,000 samples per programming language (700 for training and 300 for test). However, some LLMs do not achieve this target for specific languages — e.g., Gemini-2.0-Pro [37] on C# — due to generation refusals caused by the model's output filtering policies. Despite these occasional omissions, the overall quality of ***CodeMirage***'s dataset remain unaffected.

F Additional Details of Baseline Detectors

In this section, we provide additional introduction and implementation details for each of the ten baseline detectors evaluated in ***CodeMirage***.

LogRank* [22] & *Entropy* [42].** These two baseline detectors represent classic zero-shot detection approaches that rely on pretrained LLMs. The underlying intuition is that LLMs are more familiar with AI-generated text or code, resulting in lower token-level log-rank or entropy values compared to human-written content. Both methods compute the average token-level statistic (log-rank or entropy) over the input, which is then used as the detection score. In ***CodeMirage, we implement these detectors using the state-of-the-art open-source pretrained model *Llama-3.2-3B-Instruct*[1] as the scoring backbone.

Binoculars* [28].** *Binoculars* is a state-of-the-art zero-shot detector based on the insight that AI-generated text or code tends to receive more consistent scores across different LLMs than human-written content. To exploit this property, the method feeds the input simultaneously into two distinct LLMs and computes a novel *cross-perplexity* metric as the detection score. In ***CodeMirage, we adopt the official implementation[2] of *Binoculars* to ensure reproducibility and optimized performance.

***Embed-Code* [75] & *Embed-AST* [75].** These two embedding-based methods leverage pretrained code embedding models to extract semantic representations of entire code files. *Embed-Code* encodes the raw source code directly, while *Embed-AST* first parses the code into its abstract syntax tree (AST) using *tree-sitter*[3], and then encodes the AST. The embeddings are then passed to a supervised

[1] https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct

[2] https://github.com/ahans30/Binoculars

[3] https://github.com/tree-sitter/tree-sitter

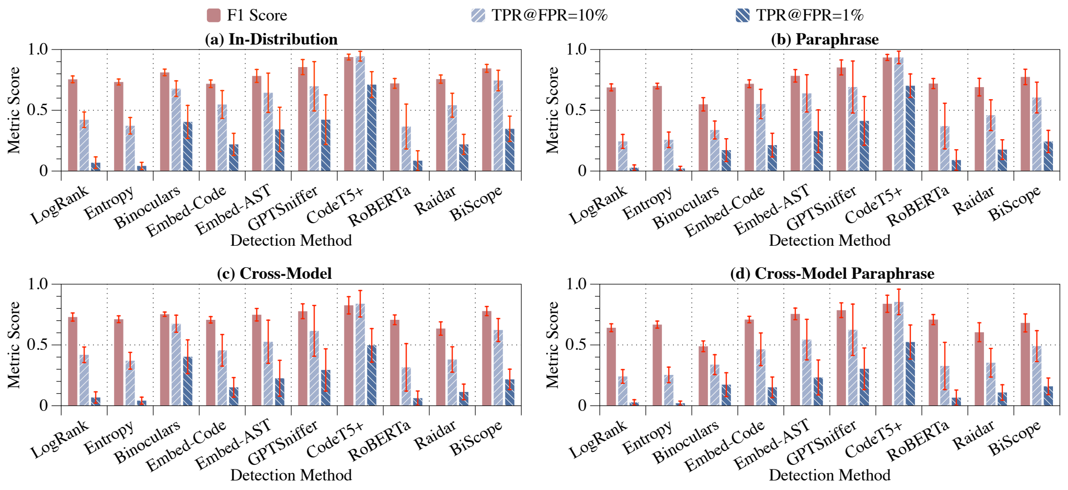

Figure 6: **Comparison Between Different Evaluation Metrics.** The bar charts illustrate the average F1 scores of baseline detectors on different LLMs across programming languages.

classifier for detection. In ***CodeMirage***, we employ the latest *CodeXEmbed-2B* [46] model as the embedding model and use a *Random Forest* [6] classifier as the downstream detector.

GPTSniffer* [52, 53].** *GPTSniffer* is a state-of-the-art fine-tuning-based detector that leverages the code-related capability of *CodeBERT* [19]. It is fine-tuned on a labeled dataset consisting of both human-written and AI-generated code samples, and evaluated on unseen test data. In ***CodeMirage, we adopt training hyperparameters consistent with prior work [58]: 5 training epochs, a learning rate of 3e-4, weight decay of 1e-3, and a warmup ratio of 0.1. We train *GPTSniffer* on ***CodeMirage***'s training set and evaluate on ***CodeMirage***'s test set.

CodeT5+* [84] & *RoBERTa* [47].** These two fine-tuning-based detectors follow the same training pipeline as *GPTSniffer*, but utilize different backbone models: the latest *CodeT5+* [84] and the classic *RoBERTa* [47]. In ***CodeMirage, we use the same training hyperparameters and evaluation settings as those employed for *GPTSniffer* to ensure a fair comparison.

Raidar* [48].** *Raidar* is based on the observation that LLMs tend to modify a greater proportion of human-written content compared to AI-generated content. It hence uses multiple prompts to instruct an LLM to rewrite the input and then computes a set of numerical features (*e.g.*, Bag-of-Words edit distance and Levenshtein score). These features are used to train a downstream classifier as the final detector. In ***CodeMirage, we adopt the latest *GPT-4.1-nano*[4] as the rewriting model, which is stronger than the original *GPT-3.5-Turbo* used in *Raidar*. We also follow the official implementation[5] to extract features and train the detection model.

BiScope* [27].** *BiScope* is a state-of-the-art detector that leverages a pre-trained LLM to extract bi-directional entropy features, which are then used to train a lightweight downstream classifier. The bi-directional entropy is designed to capture both next-token prediction (forward entropy) and previous-token memorization (backward entropy) from the model's output logits. In ***CodeMirage, we use *Llama-3.2-3B-Instruct*[6] as the feature extractor for *BiScope*, consistent with the scoring model used in *LogRank* and *Entropy*. A *Random Forest* [6] classifier is employed as the downstream detector.

G Evaluation Results of Additional Metrics

The results appear in Figure 6, where the x-axis lists the detection methods and the y-axis shows their metric values. As before, each bar reflects the mean performance across ten programming languages

[4]https://platform.openai.com/docs/models/gpt-4.1-nano
[5]https://github.com/cvlab-columbia/raidarllmdetect
[6]https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct

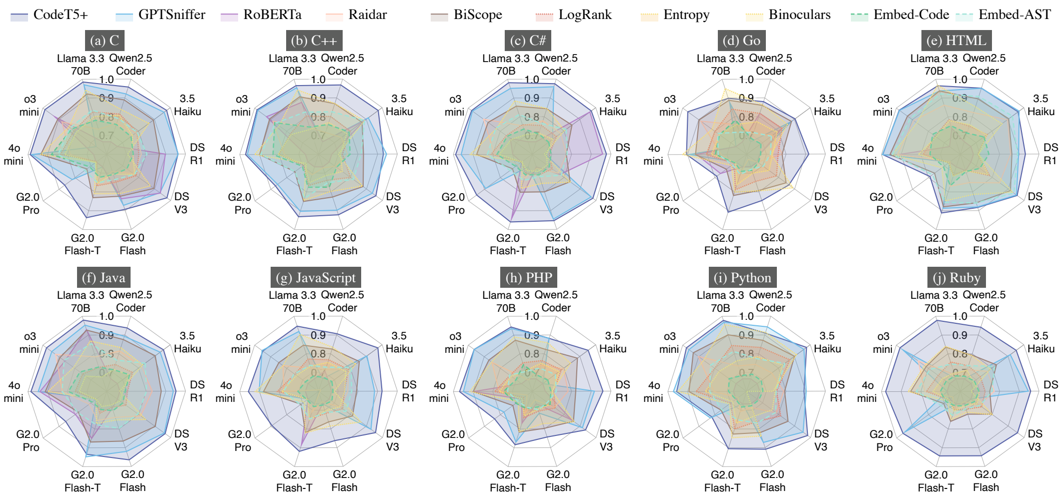

Figure 7: Complete F1 scores of all baseline detectors across various LLMs and programming languages under the in-distribution configuration.

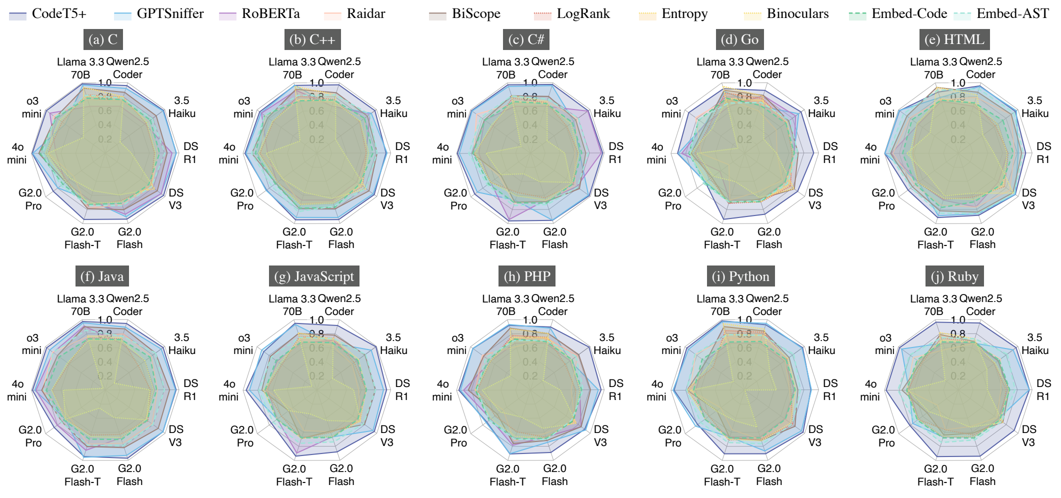

Figure 8: Complete F1 scores of all baseline detectors across various LLMs and programming languages under the paraphrase configuration.

and ten LLMs, with error bars indicating one standard deviation. The figure is divided into four panels, each corresponding to a different evaluation configuration. Despite decent F1 scores across the board, all detectors suffer a dramatic drop in true-positive rate once the false-positive rate is constrained (e.g., TPR@FPR=1% is generally lower than 0.3), showing that they fail to catch enough positives under realistic, low-alarm requirements and are therefore impractical.

H Additional Evaluation Results

In this section, we present the complete F1 scores of all baseline detectors evaluated across different LLMs and programming languages. Specifically, Figure 7 shows the results under the in-distribution configuration, while Figure 8 reports the scores under the paraphrase configuration. Figure 9 illustrates the results under the cross-model configuration, and Figure 10 presents the scores under the cross-model paraphrase configuration.

These comprehensive results are consistent with the trends discussed in section 4, further validating the key findings derived from the ***CodeMirage*** evaluation.

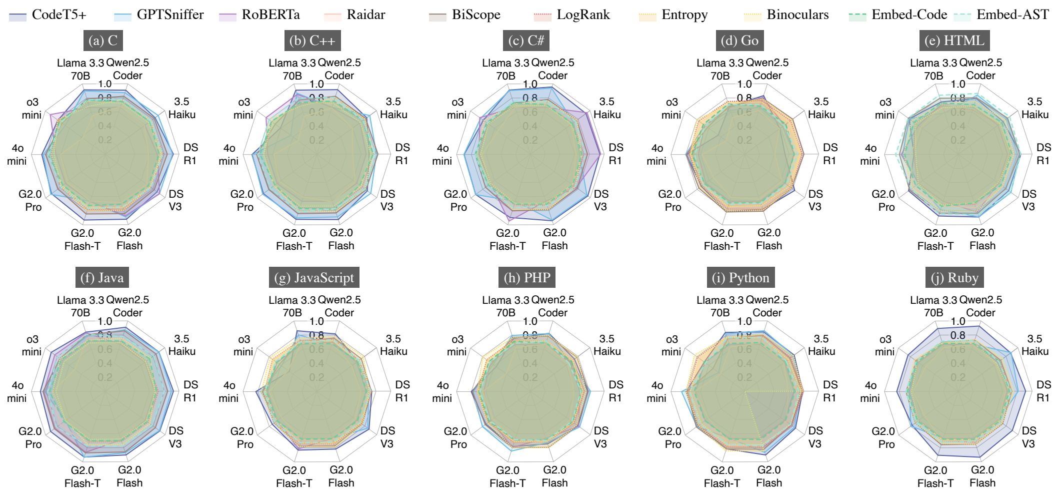

Figure 9: Complete F1 scores of all baseline detectors across various LLMs and programming languages under the cross-model configuration.

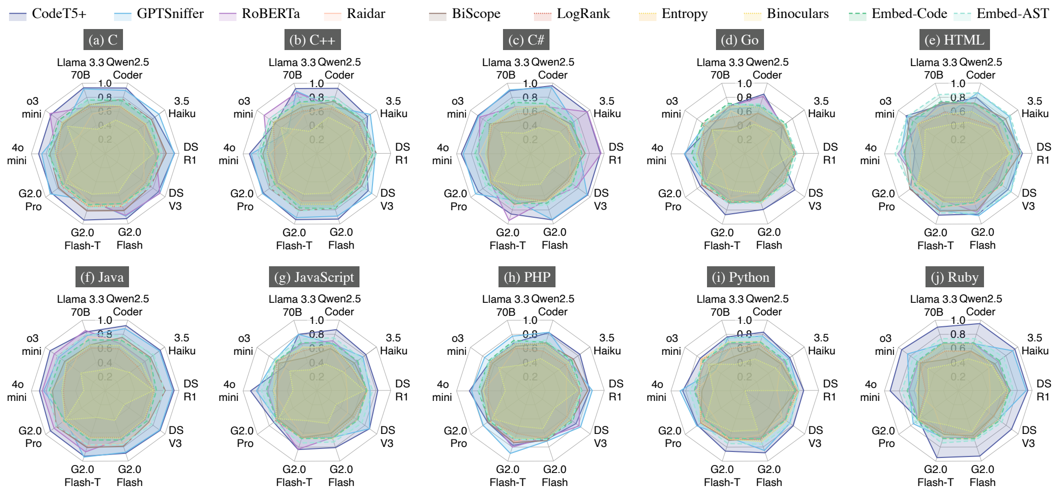

Figure 10: Complete F1 scores of all baseline detectors across various LLMs and programming languages under the cross-model paraphrase configuration.

I Limitations and Future Work

While ***CodeMirage*** represents a significant step toward a more comprehensive evaluation of AI-generated code detectors, several limitations remain and could be addressed in future work.

First, though ***CodeMirage*** includes a broad set of programming languages, LLMs, and detectors, it does not exhaustively cover all possibilities. Additional languages, particularly those less commonly used in mainstream software development but still important in specific domains, remain unexplored. Similarly, many emerging LLMs and detection techniques are not included in the current benchmark. Future work could expand ***CodeMirage*** to incorporate these newly emerged models and underrepresented languages, enabling broader and more inclusive evaluations.

Second, ***CodeMirage*** focuses exclusively on prompt-based paraphrasing attacks in its adversarial setting, given their practicality and prevalence in real-world coding. However, a wider spectrum of adversarial techniques, especially those designed for the natural language domain, could be explored. Future efforts could adapt adversarial strategies against natural language detection to the code domain or propose novel code-specific attack paradigms to more rigorously evaluate detector robustness.

Third, ***CodeMirage*** centers on document-level detection where AI-generated code files are fully generated by LLMs. In practice, however, AI coding assistants often generate partial code completions embedded in human-written code. Evaluating detection methods under mixed-authored code with different granularities could be an important direction for future benchmarks and detection methods.

Despite these limitations, ***CodeMirage*** advances the field by offering a more comprehensive and realistic evaluation benchmark compared to prior work [75, 59, 14, 62, 58, 87]. We believe the insights obtained and evaluation platform established by ***CodeMirage*** will serve as a strong foundation for developing more robust and generalizable AI-generated code detectors.