HybridCodeAuthorship: A Benchmark Dataset for Line-Level Code Authorship Detection

Luke Patterson, Li Wang, Adam Faulkner
Card Intelligence, Capital One
{luke.patterson, li.wang, adam.faulkner}@capitalone.com

Abstract

Thanks to the rapid adoption of AI code assistants powered by large language models (LLMs), industry codebases are, increasingly, a hybrid of AI- and human-authored code. For risk management and productivity analysis purposes, it is crucial to enable fine-grained location detection of AI-generated code. To develop algorithms for this task, quality benchmarks are needed to assess performance. However, existing benchmarks tend to comprise academic, LeetCode-style problems and presume a code snippet is either completely human-authored or completely AI-authored, which is not reflective of the diverse intents and styles of industry codebases utilizing AI code assistants. To fill these gaps, we introduce HybridCodeAuthorship, a novel benchmark of Python code files with interleaved human- and AI-authored lines of code to simulate authentic utilization of AI code assistants. In this paper, we first present our dataset construction pipeline, which leverages CodeSearchNet, a massive collection of links to open sourced repositories on GitHub. We then benchmark the performance of two state-of-the-art AI-generated code detection algorithms at both the line- and chunk-level. Experimental results demonstrate that HybridCodeAuthorship is a challenging benchmark with a top-scoring algorithm, AIGCode Detector, obtaining a highest F1 score of 0.48 and 0.56 on line-level and chunk-level code detection tasks, respectively.

Keywords: AI-generated code, benchmark dataset, line-level detection

1. Introduction

Recent advances in generative AI are fundamentally reshaping the landscape of software development. AI code assistants powered by cutting-edge large language models (LLMs) are being rapidly adopted by industry to enhance developers' productivity.[1] A number of studies have been published that attempt to quantify the benefits of AI-based code assistants such as GitHub Copilot, Amazon CodeWhisperer, Google Cider, and Cursor. While most of the studies observed that developers who used AI code assistants experienced a 20% to 56% productivity increase (Deniz et al., 2023; Cui et al., 2025; Peng et al., 2023; Chatterjee et al., 2024; Paradis et al., 2025), one study has reported the opposite — experienced open-source contributors were decelerated by AI code assistants by ~19% (Becker et al., 2025). To provide a more fine-grained view of developer adoption of these tools, additional work has measured the acceptance rates of coding suggestions and lines of code generated by AI code assistants (Ziegler et al., 2024; Bakal et al., 2025).

With rapid industrial adoption of coding assistants, hybrid code authorship has been described as a new paradigm of software development, with contemporary codebases increasingly a hybrid of interleaved human- and AI-generated code. For example, a mixed-methods study investigated the practical use of AI coding assistants (Sergeyuk et al., 2025), and highlighted the numerous interactions and edits that users typically made to AI-generated code. Recognizing this new paradigm, commercial engineering management platforms are increasingly adopting AI impact measures.[2]

AI-generated code detection approaches typically involve extending existing AI-generated text detection approaches to the domain of code. This is sensibly motivated since distributional differences between human- and AI-generated natural language text, as measured by perplexity, have also been observed in human- versus AI-generated code (Xu and Sheng, 2024). In benchmarking code detection approaches, researchers have relied on academic, LeetCode-style problem sets that bear little stylistic resemblance to authentic code produced as part of a typical software development project. A more accurate measure of the effectiveness of these approaches requires a dataset that 1) more accurately reflects the range of styles and intents found in practical codebases and 2) accurately reflects how developers make use of AI code suggestions by interleaving AI- and human-authored code. To the best of our knowledge, no current AI-generated code detection benchmark

[1] A recent survey of developers from StackOverflow notes "76% of all respondents are using or are planning to use AI tools in their development process this year, an increase from last year (70%)." - https://survey.stackoverflow.co/2024/ai

[2] For examples, see: https://jellyfish.co/blog/ai-impact-framework/ and https://www.faros.ai/blog/how-much-code-is-ai-generated

Proceedings of the Fifteenth Language Resources and Evaluation Conference (LREC 2026), pages 1520–1532
11–16 May 2026. ©ELRA Language Resources Association (ELRA), 2026

datasets meet these two criteria. To fill this gap, we introduce HybridCodeAuthorship, a novel benchmark dataset of Python code files implementing common software engineering tasks with human- and AI-generated code interleaved throughout.

Our contribution is two-fold:

1. A benchmark dataset, HybridCodeAuthorship, composed of full code files with interleaved human-authored and AI-generated code.

2. Experimental results showing initial benchmark performance of adapting two state-of-the-art AI-generated code detection algorithms for both line- and chunk-level code detection using HybridCodeAuthorship.

As part of the first contribution, we created HybridCodeAuthorship [3] using a data construction pipeline that leverages CodeSearchNet (Husain et al., 2019). CodeSearchNet was released by GitHub and Microsoft Research, and is a massive collection of links to open-source repositories on GitHub. CodeSearchNet contains links to approximately 2 million code files believed to be human-authored in popular programming languages such as Python, Java, and Go. The current version of HybridCodeAuthorship is limited to Python code snippets with the other programming languages reserved for future extensions. We defined two phases for the pipeline: code testing and code interleaving. The code testing phase was developed to verify the correctness of both human-authored and AI-generated code files. During this phase, each human-authored file and its AI-generated counterpart were labeled to indicate whether they passed or failed the same unit tests, respectively. These labels are included in the dataset release and, in practice, will allow experimenters to filter out non-functional code if their experiment setting requires such filtering.

The code interleaving phase consists of three steps: code identification and masking, code marking, and, finally, code generation. In this latter phase, a specific portion of lines in each human-authored code file was selected, masked, and marked by an LLM with comments that could be recognized by an LLM for code generation. The resulting AI-generated code file was also validated for correctness with respect to Python syntax. The final dataset, in contrast to other benchmark datasets for AI code detection, accurately reflects real-world adoption of AI code assistants in software development by interleaving human-authored and AI-generated code relative to various acceptance rates.

[3]The complete benchmark is available at https://github.com/CapitalOne-Research/c1-hybrid-code-authorship

Our interleaving algorithm is directly informed by recent mixed-methods research (Sergeyuk et al., 2025) which highlights the frequent edits and fine-grained interactions developers have with AI-generated suggestions. We ask LLMs to first select distinct, atomic parts of human-authored code files and replace them with descriptive summaries of their functionality. These summaries serve as proxies for user-provided prompts, reflecting a developer's specific intent for what a code block should achieve within a larger codebase. This approach mimics the authentic interaction of a user guiding a coding assistant to generate code that fulfills a specific requirement while maintaining the surrounding human-authored context. To ensure the realism of the resulting hybrid code, we use complex, real-world repositories from CodeSearchNet as our foundation. We further validate the authenticity of these interleaved files by requiring them to pass the original project unit tests and maintain syntactical correctness, ensuring they represent functional software rather than just superficial code fragments.

Our second contribution, experimental results utilizing two state-of-the-art AI-generated code detection algorithms, demonstrate that line- and chunk-level code detection is a challenging task with a top-scoring approach, AIGCode Detector (Xu and Sheng, 2024), obtaining a highest F1 score of 0.48 and 0.56 on line-level and chunk-level code detection tasks, respectively. Line-level code detection treats each line of code as a distinct unit for authorship classification. Chunk-level first creates "chunks" by concatenating consecutive lines of code that share the same author type, and treats those chunks as the units for classification.

The rest of the paper is organized as follows: Section 2 reviews previous work on benchmark datasets and relevant algorithms for AI code detection. Section 3 describes the two core phases powering the data construction pipeline while Section 4 describes the resulting benchmark dataset, HybridCodeAuthorship. Section 5 presents experiment results with two selected approaches to AI-generated code detection. Section 6 discusses the limitations of our work. Finally, Section 7 provides a summary of our work and outlines our plans for future research. We hope that our work will accelerate the development of new algorithms and inspire future studies that explore topics beyond validating productivity enhancement driven by AI code assistants.

2. Related Work

In recent years, numerous benchmark datasets have been proposed to evaluate the performance of AI code detection algorithms. Alam et al. (2023)

developed GPTCloneBench, a large dataset of AI-Human clone pairs generated with GPT-3 (Brown et al., 2020). Demirok and Kutlu (2024) created AIGCodeSet, containing human-authored and AI-generated pairs using a variety of LLMs. Several other datasets have been proposed in recent years (Bulla et al., 2024; Idialu et al., 2024; Pan et al., 2024; Shi et al., 2025; Xu and Sheng, 2024; Choi and Mohaisen, 2025; Demirok et al., 2025). Recently, newer datasets have expanded the number of languages, models, and code files. Guo et al. (2025) introduced a dataset of over 200,000 sample code files including 10 different languages and uses 10 different LLMs to generate code. Furthermore, Orel et al. (2025) introduced an extensive resource suite including, DroidCollection, a large-scale dataset of over one million code samples across 7 programming languages, 43 language models and 3 coding domains.

However, the construction of these datasets remains similar: the authors start with some set of prompts and human-authored solutions. Then, the authors ask an LLM or LLMs to produce AI-generated code for the same prompt. In almost all existing benchmark datasets, code snippets are either entirely human-authored or entirely AI-generated. These benchmarks are designed for the task of binary authorship classification on code snippets. Orel et al. (2025) proposed machine-refined samples to simulate three coding scenarios of human-AI interaction in the real world: human-to-LLM continuation, gap filling and code rewriting. However, the machine-refined samples still target coarse-grained classification tasks with sample-level classes. To reflect the evolving trend of hybrid authorship in code, detection algorithms must be tested at a more finer-grained unit of analysis, such as the line-level.

There are now numerous commercially available tools dedicated to the task of AI-generated text detection (Copyleaks, 2023; GPTZero, 2023; OpenAI, 2023; Writer, 2023) and both classification (Guo et al., 2023; Solaiman et al., 2019) and perturbation-based approaches (Mitchell et al., 2023; Xu and Sheng, 2024) to the task have been proposed in the literature. Line-level code authorship attribution remains an under-researched task with a handful of approaches existing in the stylometry and authorship attribution literature predating contemporary LLMs. Çalışkan İslam et al. (2015) developed a random forest stylometric classifier with features derived from abstract syntax trees of code, Wang et al. (2018) introduced an approach that uses features generated dynamically from code execution to detect authorship, and Abuhamad et al. (2020) implemented an RNN-based authorship verification model for this task. Recently, Orel et al. (2025) trained a suite of Transformer encoders, DroidDetect, using a supervised contrastive objective over DroidCollection after removing suspicious AI code. Xu and Sheng (2024) developed AIGCode Detector, which is an adaptation of DetectGPT. It identifies AI-generated code by comparing the log probabilities of original and perturbed versions, utilizing CodeBERT for perturbations and a composite scoring system based on perplexity, standard deviation, and burstiness.

This concept of hybrid authorship also applies to the domain of natural language text, and several benchmark datasets of text have already been constructed. Kadiyala et al. (2025) produced a large-scale dataset with 2.4 million hybrid texts in 23 languages from 12 different models. Similarly, Su et al. (2025) introduced HACo-Det, a dataset generated via an automatic pipeline that provides word-level attribution labels. Other notable datasets for detecting hybrid prose include MixSet (Zhang et al., 2024), RAID (Dugan et al., 2024) and Beemo (Artemova et al., 2024). In this paper, we extend this process to the production of hybrid authorship Python code.

3. Benchmark Construction

We provide an illustration of the data construction pipeline for our benchmark dataset, HybridCodeAuthorship, in Figure 1. The pipeline processes code files sampled from CodeSearchNet in two phases: code testing and code interleaving. Notably, both phases comprise multiple steps and code testing is invoked twice to run the same unit tests against each human-authored code file and its AI-modified counterpart.

3.1. CodeSearchNet

CodeSearchNet (Husain et al., 2019), released by GitHub and Microsoft Research, is a massive collection of links to open-source repositories on GitHub. It contains approximately 2 million links to code files written in 6 programming languages (i.e., Go, Java, JavaScript, PHP, Python and Ruby). The Python subset of links in CodeSearchNet points to more than 450,000 Python code files from over 13,000 unique repositories with diverse levels of complexity and intent. Each file is identified by its metadata including GitHub URL, repository name, file path, function name, docstring, function code and commit hash. As CodeSearchNet was originally proposed to facilitate model training for semantic code search, the code files were further split into training, validation and test sets.

Released in 2019 before the emergence of LLMs, CodeSearchNet is generally believed to represent real-world code written by humans and is considered a standard benchmark for evaluating algo-

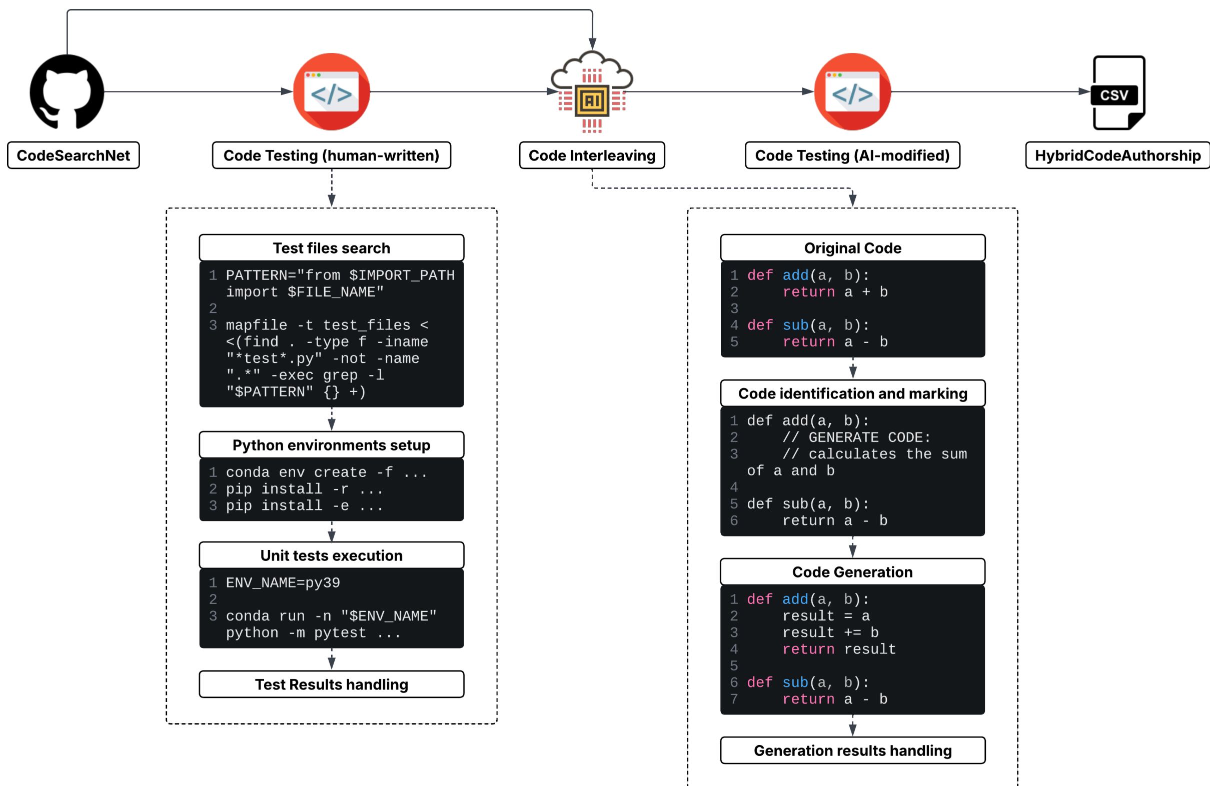


Figure 1: Overview of the data construction pipeline for HybridCodeAuthorship. First, human-authored code files sampled from CodeSearchNet were validated during a code testing phase, which involves test files search, Python environment setup, and unit test execution. Second, all human-authored files were processed by the code interleaving phase in a parallel step that performed code identification, code marking (i.e., instructing an LLM to provide a comment for the downstream LLM to write code that fulfills the intent of the masked code) and, finally, code generation using an LLM. Third, the AI-modified code files were verified via another code testing phase. Finally, both human-authored and AI-modified code files were collected and annotated at the line-level as either "Human" or "AI", resulting in a final benchmark dataset.

rithms for code generation and detection (Feng et al., 2020; Wang et al., 2021; Lu et al., 2021; Wang et al., 2023; Li et al., 2023; Xu et al., 2025). In addition to being human-authored, CodeSearchNet was selected as a representative sample of GitHub repositories implementing software solutions for real-world tasks.

To construct HybridCodeAuthorship, we selected a subset of 4,814 Python code files from CodeSearchNet that are still available on GitHub.

3.2. Code Testing

As part of our broader goal of creating a dataset that reflects real-world coding tasks, we first assessed the correctness of the code as code correctness is generally viewed as a prerequisite of functioning software. To provide information about the validity of both human-authored and AI-modified code in HybridCodeAuthorship, we developed a code testing phase that verifies code files by running relevant unit tests from their repositories using `Bash`. The same steps were followed to test both human-authored and AI-modified code.

3.2.1. Test Files Search

Given a code file (either human-authored or AI-modified) and its repository, code testing started with a search for all test files that import the target code file within the associated commit and the `main/master` branch. Ancestor directories of the target file were first scanned for non-hidden folders containing the `test` handle. If one or more test folders were found, our code testing framework then searched for all test files in each test folder that contain common import statements mentioning the target code file using a regular expression:

```
'\b(import|from)\b[^#]*\b($PARENT_DIR|
    $FILE_NAME)\b'
```

Notably, `PARENT_DIR` is included to handle the edge case where `FILE_NAME` is `__init__.py`. For example, given a target file https://github.com/keon/algorithms/blob/4d656946

4a62a75c1357acc97e2dd32ee2f9f4a3/algorithms/queues/priority_queue.py, all the file paths under /algorithms/algorithms/queues/priority_queue.py were scanned for common import statements like:

```
import priority_queue
import queues.priority_queue
import algorithms.queues.priority_queue
from priority_queue import
from queues.priority_queue import
from algorithms.queues.priority_queue import
```

This approach to selectively executing unit tests, rather than the entire suite, minimizes false positive results that occur when the code file passes tests that are unrelated to its functionality and boosts the efficiency of the entire pipeline.

3.2.2. Python Environment Setup

Following a successful search of eligible test files, code testing proceeded with multiple Python environments whose versions were tied to the code being tested. We made the following key decisions related to environment setup:

- Python versions (2.7, 3.6 and 3.12) were aligned to the time range in which code files of CodeSearchNet were created.
- Compatible versions of common libraries, such as numpy, scipy, mock, etc., were preinstalled as part of the "default" environment.
- The open-source testing framework, pytest, which contains built-in support of other testing libraries such as unittest and nose, was used to execute all unit tests.

To minimize import errors during unit tests and mitigate false negative results, we also installed the repository and its required libraries through setup.py, pyproject.toml and requirements.txt files. Crucially, each repository was installed in a clean virtual environment to avoid dependency conflicts.

3.2.3. Test Execution and Correction

To evaluate the functional correctness of both human-authored and AI-modified code, we implemented an iterative testing methodology. This strategy systematically filtered out the code snippets that passed all tests in any eligible test file during a given run, allowing subsequent runs to focus on only those that failed all tests with upgraded testing code based on failure logs. Furthermore, the iterative filtering in the strategy prevents redundant test executions and optimizes the utilization of computation resources.

With a large number of code files, it is practically infeasible to test each single function in a code file with the hope to pass all tests across multiple test files, without exaggerating the likelihood of false negative results. By adopting a greedy heuristic classifying a code snippet as a success if it passes all tests in any eligible test file, we believe we effectively balanced the trade-off between false negatives and false positives.

Although required dependencies were mostly captured by the environment setup stage, pytest could still raise ModuleNotFound error during run time. To further mitigate false negative results, we developed an adaptive correction mechanism that attempted to install missing libraries based on the error messages from pytest during a testing run. For libraries whose names cannot be inferred from their import statements (e.g., scikit-learn), we maintained a hard-coded list of missing libraries throughout the iterations.

3.2.4. Test Results Handling

We added two columns HumanCodeTier and AICodeTier in HybridCodeAuthorship to indicate the validity of each human and AI code file, respectively. Each column comprises of three validity levels: "Unit Test Passed", "AST parsable" and "Unparsable". If a code file passed any test running within a Python environment and a repository state, it was annotated with "Unit Test Passed". Otherwise, if the code file was still AST (abstract syntax tree) parsable, it received the annotation "AST parsable". Because AST is dependent on Python version, we also included the version in the "AST parsable" label for completeness (e.g., "AST parsable: py36").

Unit tests associated with a target code file could fail for a variety of reasons, especially for legacy repositories. The most common reasons for both human-authored and AI-modified code failure were:

- Tests Not Found: No test files matched the common import patterns or the test files were created in a fork (i.e., copy of the repository).
- Import Error: Required modules did not exist in available libraries or the import paths of custom modules were broken.
- Module Not Found Error: Required libraries were missing for unit tests or could not be installed.
- Syntax Error: Code in the target code file or test files violated Python grammar.

- `Repo Not Found`: Repositories became missing or inaccessible or could not be installed.

Notably, AI-modified code tended to produce new errors not observed with human-authored code, including `Attribute Error`, `Type Error`, `Name Error` and `Value Error`. Please refer to Appendix A for complete error analysis.

3.3. Code Interleaving

We attempted code interleaving on a set of 4,814 human-authored code files. Segments in each file were identified, masked, and subsequently replaced with AI-generated code that preserved the original functionality and intent of the masked code. This process of *AI code interleaving* was performed in three steps for each file. These steps were repeated for three different LLMs, `Llama3.3-70B`, `Llama-4-Scout` (Dubey et al., 2024), and `GPT-OSS-120b` (Agarwal et al., 2025). These steps are visualized in Figure 1. 4,196 files successfully completed our pipeline for at least one LLM.

3.3.1. Code Identification

In the first, code identification step, the LLM was prompted to identify lines of meaningful, atomic code for replacement. To modulate the extent of code modification, the prompt included a target replacement percentage, denoted as p. This parameter was intended to control the density of AI-generated code within the output interleaved file. Over all code files, the value of p was systematically varied, sampling uniformly from the discrete set $\{10\%, 20\%, \ldots, 100\%\}$. While LLM outputs did not always adhere to this request, we empirically observed that inclusion of this instruction helped increase overall variance in AI code density among code files. We also left a random sample of files (10%) completely unchanged and marked all lines as `Human`. This increased the difficulty of the AI-generated code detection task and improves the robustness of the resulting benchmark where code detection algorithms cannot rely on file-level signal potentially correlated with the existence of AI-generated line of code in the file.

3.3.2. Code Marking

In the second, code marking step, the LLM was instructed to remove the original code identified lines in step 1 and to replace the code with a placeholder comment starting with the string `GENERATE CODE:`. This clearly identified the section of code to be generated in step 3. Additionally, the LLM was instructed to include a comment that summarizes the intent of the deleted code in enough detail to allow plausible reconstruction with similar functionality, though not necessarily with the same exact syntax. See Appendix B for the code marking prompt.

3.3.3. Code Generation

In the final step, the marked source file was presented to the LLM for code generation. During this phase, the model was asked to parse the file, identify all instances of the `GENERATE CODE:` placeholders and, for each one, to generate a new code segment that met the requirements described in the associated comment. See Appendix C for the code generation prompt.

3.4. Determining Line-Level Authorship

For each interleaved code file, we must reconcile which lines are AI-generated and which are human-authored. To do this, we used the Python `diff` library to track which lines were preserved from the original code (marked as `Human`) versus which lines were newly introduced in the interleaving process (marked as `AI`). This provides the ground-truth labeling needed for benchmark evaluation.

To focus evaluation on substantive code portions, we also classified lines as either `Trivial` or `Non-trivial`. This allows experimenters to vary the segments targeted by their AI-generated code detection algorithms depending on the requirements of their experiments. We used `regex` matching logic to identify lines with minimal semantic content:

```
r'^[a-zA-Z_][a-zA-Z0-9_]*\s*=\s*(?:True|
    False|\d+|"[^"]*"|\'[^\']*\')$'
```

A line is also marked as `Trivial` if it consists of whitespace, simple comments, docstrings, `pass` statements, or standalone brackets.

4. HybridCodeAuthorship Dataset

The HybridCodeAuthorship dataset comprises 10,488 records derived from 4,196 Python code files. Each file was independently rewritten by multiple LLMs, resulting in several modified versions per source file. Of the 10,488 file records, 39% (4,103) of human-authored code files passed unit tests. In comparison, 29% (3,000) of the 10,488 AI-interleaved records passed unit tests. Within these files, there is a total of 2,827,938 lines of code. 17% (488,896) of lines are AI-generated, while the rest are human-authored. 69% (1,943,728) of lines are deemed nontrivial. The schema for the dataset, detailing its columns and data types, is presented in Table 1. The distributions of file counts and average proportions of AI-generated lines across file sizes and proportions of nontrivial lines are visualized in Figure 2. The top two plots overview the

Column Name	Data Type	Description
ModelId	String	Identifier for which LLM was used for code generation.
RecordId	String	A unique identifier for each code sample, serving as the primary key when combined with ModelId.
Language	String	The programming language of the code sample.
GitHubUrl	String	The URL to the original GitHub repository and file.
HumanCode	String	The original, unmodified human-authored code.
HumanCodeTier	String	The validity tier of the human-authored code ("Unit Test Passed", "AST Parsable", "Unparsable").
AICode	String	The final code containing interleaved AI-generated content.
AICodeTier	String	The validity tier of the AI-generated code ("Unit Test Passed", "AST Parsable", "Unparsable").
AICodeLines	List of String	AICode string as a list with each line its own item.
LineNumber	List of Int	List of line numbers for AICode. Used as index for lists in AICodeLines, Attribution, and Triviality columns.
Attribution	List of String	List of attribution labels for AICode lines ("AI" or "Human").
Triviality	List of String	List of triviality labels for AICode lines ("Trivial" or "Nontrivial").
AILineProportion	Float	The actual proportion of lines attributed to AI in the AICode.

Table 1: HybridCodeAuthorship schema.

entire benchmark and the bottom two zoom into the subset where both human-authored and AI-generated code files passed unit tests. Regarding data utilization, we recommend taking additional data processing steps to balance the proportions of AI-generated lines across files with different sizes.

5. Experimental Results

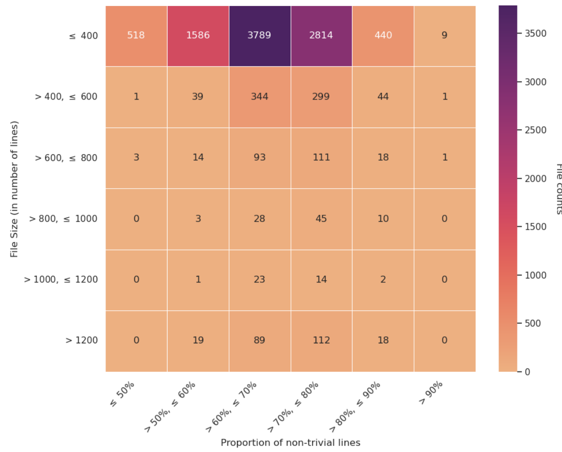
While HybridCodeAuthorship is intended to benchmark the performance of AI-generated code detection approaches at the line-level, for completeness we also report the results of chunk-level experiments. Chunk-level detection involves concatenating consecutive lines of code with the same author, either human or AI. We benchmarked the task of both line- and chunk-level AI-generated code detection on the HybridCodeAuthorship dataset using two state-of-the-art approaches to automated AI-generated code detection. The first, DroidDetect (Orel et al., 2025) [4] is a model-based AI-generated code detector that uses ModernBERT (Warner et al., 2025) as a base-model and is fine-tuned on a collection of human- and AI-generated code. The second, AIGCode Detector (Xu and Sheng, 2024), is a modified version of the DetectGPT (Mitchell et al., 2023) algorithm, which introduced a perturbation-based approach to AI-generated text detection: the target text is perturbed using another pre-trained model, such as T5, and the log probability of the perturbed and non-perturbed variants of the text are compared. This approach leverages the observation that machine-generated text tends to have a lower log probability after minor perturbations, whereas human-authored text is less sensitive to these changes. AIGCode Detector's modifications include the use of CodeBERT (Feng et al., 2020), rather than T5, to perturb the target code, with the proportion of targeted code determined by the perplexity score of the target code rather than the fixed percentage utilized by DetectGPT. Additionally, rather than using the "perplexity disparity" between perturbed and unperturbed code to assign a human vs. AI score, a composite score is calculated which combines the perplexity of the target code, the standard deviation of the perplexity scores calculated over multiple perturbed versions of the code, and a burstiness score.

F1 scores for the best-performing algorithm relative to the LLM, dataset split (trivial vs nontrivial) and granularity (line- vs chunk-level) have been boldfaced in Table 2. AIGCode Detector easily out-performed DroidDetect across all experiment variants. Chunk-level detection emerged as the more challenging task which aligns with the original DetectGPT authors' (Mitchell et al., 2023) observation of a degradation in DetectGPT's performance as a function of text length. As expected, performance on the task of detection in trivial segments, which contain minimal semantic content, was significantly worse than that of nontrivial segments for all variants, with the exception of GPT-OSS-120b.

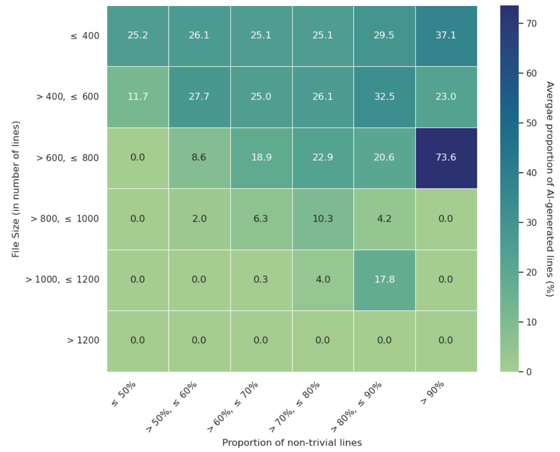
6. Limitations

Although HybridCodeAuthorship is currently the only available dataset that simulates hybrid AI and human authorship at the fine-grained line level, there are some limitations to discuss.

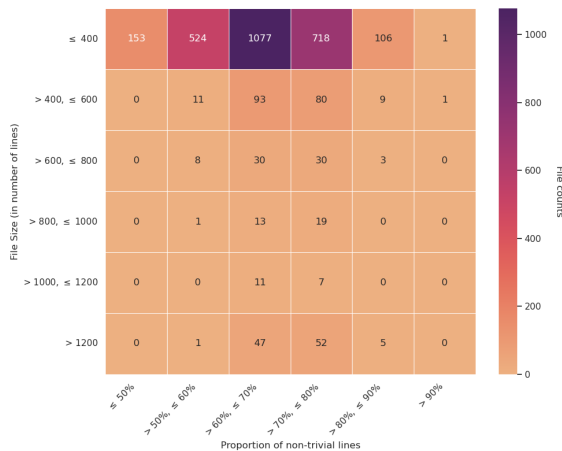
[4] https://huggingface.co/project-droid/DroidDetect-Large-Binary



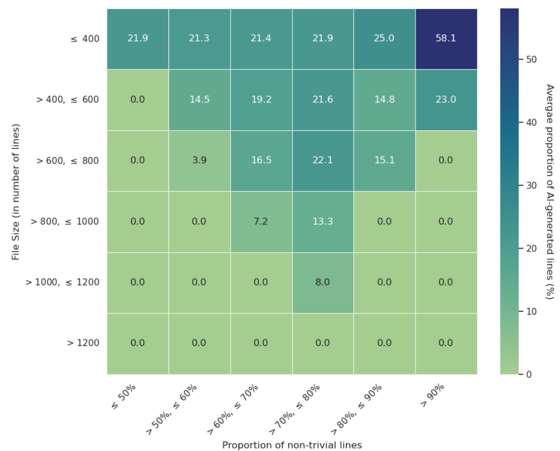
(a) Number of files by file size and proportion of nontrivial lines for the entire benchmark.



(b) Average proportion of AI-generated lines by file size and proportion of nontrivial lines for the entire benchmark.



(c) Number of files by file size and proportion of nontrivial lines for the subset where both human-authored and AI-generated files passed unit tests.



(d) Average proportion of AI-generated lines by file size and proportion of nontrivial lines for the subset where both human-authored and AI-generated files passed unit tests.

Figure 2: Overview of HybridCodeAuthorship with respect to the distributions of file counts and proportion of AI-generated lines over file size and proportion of nontrivial lines. HybridCodeAuthorship demonstrates sufficient variation in data segments with high concentration in code files that have fewer than 1000 total lines and over 50% nontrivial lines.

First, the code testing phase did not succeed in perfectly handling the incompatibility of all legacy code files in CodeSearchNet, which led to a number of false negatives in unit test results.

Second, the code interleaving phase faced challenges in code generation. Some code samples were not successfully processed by all three LLMs, which resulted in fewer samples in the final benchmark. The challenges mainly came from the finite context length and inconsistent instruction compliance during prompting. There are observable characteristics that are strongly correlated with pipeline failure (for example, code files longer than the LLM context window will always fail). This could impact the real-world representativeness of the sample of code files that make it into HybridCodeAuthorship from CodeSearchNet. Using more advanced LLMs could resolve some of these challenges in future work. Additionally, because we needed to be sure that no AI code was present in the original code we interleaved, we selected CodeSearchNet, which contains code that is 6 years old or older. As a result, HybridCodeAuthorship does not contain recently developed or popularized code libraries.

7. Conclusion

In this paper, we present a novel benchmark dataset that will allow the development of algorithms to distinguish AI-generated versus human-

LLM	Dataset Split	Granularity	Metric	Detection Algorithm	
				DroidDetect	AIGCode Detector
GPT-OSS-120b	Trivial	Line-level	Precision	0.197	0.560
			Recall	0.986	0.570
			F1	0.328	**0.560**
		Chunk-level	Precision	0.101	0.540
			Recall	0.856	0.560
			F1	0.181	**0.480**
	Nontrivial	Line-level	Precision	0.266	0.540
			Recall	0.994	0.550
			F1	0.419	**0.530**
		Chunk-level	Precision	0.144	0.510
			Recall	0.673	0.510
			F1	0.237	**0.440**
Llama-3.3-70b	Trivial	Line-level	Precision	0.054	0.520
			Recall	0.980	0.560
			F1	0.102	**0.460**
		Chunk-level	Precision	0.032	0.510
			Recall	0.744	0.520
			F1	0.062	**0.390**
	Nontrivial	Line-level	Precision	0.200	0.530
			Recall	0.986	0.550
			F1	0.333	**0.510**
		Chunk-level	Precision	0.131	0.500
			Recall	0.531	0.500
			F1	0.211	**0.440**
Llama-4-Scout	Trivial	Line-level	Precision	0.017	0.500
			Recall	0.983	0.530
			F1	0.034	**0.430**
		Chunk-level	Precision	0.020	0.500
			Recall	0.891	0.490
			F1	0.040	**0.370**
	Nontrivial	Line-level	Precision	0.064	0.510
			Recall	0.993	0.560
			F1	0.121	**0.430**
		Chunk-level	Precision	0.080	0.510
			Recall	0.778	0.540
			F1	0.145	**0.410**

Table 2: AI-generated code detection results for HybridCodeAuthorship dataset including model-specific and dataset split results

authored lines of code, built by a novel data processing pipeline that could be reused for expansion of HybridCodeAuthorship. Future work could expand HybridCodeAuthorship to include other popular programming languages such as Java, JavaScript, and Go. This would ensure greater representativeness across the landscape of software development. Another enhancement could be adding more examples from additional LLMs, which may offer greater diversity of AI-generated coding patterns and greater generalizability of benchmark results. Using LLMs that specialize in code generation and handle longer context than the three LLMs used could lead to more successfully generated interleaved results. Looking ahead, this line of work opens several promising avenues for future research. Future efforts could focus on designing algorithms specifically for the line-level attribution task, possibly by integrating features from code stylometry and other related work.

8. Bibliographical References

Mohammed Abuhamad, Tamer Abuhmed, Dae-Hun Nyang, and David Mohaisen. 2020. Multi-χ: Identifying multiple authors from source code files. *Proceedings on Privacy Enhancing Technologies*.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, et al. 2025. gpt-oss-120b & gpt-oss-20b model card.

Ajmain I Alam, Chanchal K Roy, and Kevin A Schneider. 2023. GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE.

Ekaterina Artemova, Jason Lucas, Saranya Venkatraman, Jooyoung Lee, Sergei Tilga, Adaku Uchendu, and Vladislav Mikhailov. 2024. Beemo: Benchmark of expert-edited machine-generated outputs. ArXiv preprint:2411.04032.

Gal Bakal, Ali Dasdan, Yaniv Katz, Michael Kaufman, and Guy Levin. 2025. Experience with github copilot for developer productivity at zoominfo. *arXiv preprint arXiv:2501.13282*.

Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the impact of early-2025 ai on experienced open-source developer productivity. *arXiv preprint arXiv:2507.09089*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Luana Bulla, Alessandro Midolo, Misael Mongiovì, and Emiliano Tramontana. 2024. Ex-Code: A robust and explainable model to detect AI-generated code. *Information*, 15(12):819.

Aylin Çalışkan İslam, Richard Davison, Richard Liu, Rachel Greenstadt, and Arvind Narayanan. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270.

Sayan Chatterjee, Ching Louis Liu, Gareth Rowland, and Tim Hogarth. 2024. The impact of ai tool on engineering at anz bank an empirical study on github copilot within corporate environment. *arXiv preprint arXiv:2402.05636*.

Soohyeon Choi and David Mohaisen. 2025. Attributing ChatGPT-generated source codes. *IEEE Transactions on Dependable and Secure Computing*.

Copyleaks. 2023. AI Content Detector. `https://copyleaks.com/ai-content-detector`.

Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2025. The effects of generative ai on high-skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566*.

Basak Demirok and Mucahid Kutlu. 2024. AIGCodeSet: A new annotated dataset for AI generated code detection. In *2025 33rd Signal Processing and Communications Applications Conference*, pages 1–4.

Basak Demirok, Mucahid Kutlu, and Selin Mergen. 2025. MultiAIGCD: A comprehensive dataset for ai generated code detection covering multiple languages, models, prompts, and scenarios. *arXiv preprint arXiv:2507.21693*.

Begum Karaci Deniz, Chandra Gnanasambandam, Martin Harrysson, Alharith Hussin, and Shivam Srivastava. 2023. Unleashing developer productivity with generative ai. *McKinsey Digital*, 7.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Liam Dugan, Alyssa Hwang, Filip Trhlik, Josh Magnus Ludan, Andrew Zhu, Hainiu Xu, Daphne Ippolito, and Chris Callison-Burch. 2024. RAId: A shared benchmark for robust evaluation of machine-generated text detectors. ArXiv preprint:2405.07940.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547.

GPTZero. 2023. GPTZero. `https://gptzero.me/`.

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arXiv:2301.07597*.

Hanxi Guo et al. 2025. Codemirage: A multi-lingual benchmark for detecting ai-generated and paraphrased source code from production-level llms. *arXiv preprint arXiv:2506.11059*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Mei Nagappan. 2024. Whodunit: Classifying code as human authored or GPT-4 generated-a case study on CodeChef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 394–406.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Ram Mohan Rao Kadiyala et al. 2025. Robust and fine-grained detection of AI generated texts. ArXiv preprint:2504.11952.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International conference on machine learning*, pages 24950–24962. PMLR.

OpenAI. 2023. AI Text Classifier. https://platform.openai.com/ai-text-classifier.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025. Droid: A resource suite for ai-generated code detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31251–31277.

Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. Assessing AI detectors in identifying AI-generated code: Implications for education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, pages 1–11.

Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2025. How much does ai impact development speed? an enterprise-based randomized controlled trial. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 618–629. IEEE.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.

Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. 2025. Beyond synthetic benchmarks: Evaluating llm performance on real-world class-level code generation. *arXiv preprint arXiv:2510.26130*.

Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, 178:107610.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2025. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 51–62. IEEE Computer Society.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*.

Zhixiong Su, Yichen Wang, Herun Wan, Zhaohan Zhang, and Minnan Luo. 2025. HACo-Det: A study towards fine-grained machine-generated text detection under human-AI coauthoring. ArXiv preprint:2506.02959.

Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65.

Ningfei Wang, Shouling Ji, and Ting Wang. 2018. Integration of static and dynamic code stylometry analysis for programmer de-anonymization. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 3–14.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, et al. 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 (Volume 1: Long Papers)*, pages 2526–2547.

Hao Wen, Yueheng Zhu, Chao Liu, Xiaoxue Ren, Weiwei Du, and Meng Yan. 2024. Fixing function-level code generation errors for foundation large language models. *arXiv preprint arXiv:2409.00676*.

Writer. 2023. AI Content Detector. `https://writer.com/ai-content-detector`.

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 and Methodology*, 34(4):1–31.

Zhenyu Xu and Victor S Sheng. 2024. Detecting ai-generated code assignments using perplexity of large language models. In *Proceedings of the aaai conference on artificial intelligence*, volume 38, pages 23155–23162.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173.

Qihui Zhang, Chujie Gao, Dongping Chen, Yue Huang, Yixin Huang, Zhenyang Sun, Shilin Zhang, Weiye Li, Zhengyan Fu, Yao Wan, et al. 2024. LLM-as-a-coauthor: Can mixed human-written and machine-generated text be detected? In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 409–436.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63.

A. Test Error Analysis

We categorized different errors raised in testing both human-authored and AI-modified code. The error distributions for the two scenarios are visualized in Figure 3 and Figure 4, respectively. Both distributions share the most common errors as listed in Section 3.2.4, Test Results Handling. Specific error definitions include `Multiple Errors` (two or more concurrent errors), `Django Config Error` (`ImproperlyConfigured` exceptions), and `Connection Error` (HTTP/API access failures). Furthermore, AI-modified code by the open-source LLMs in our case tended to produce new errors not observed with human-authored code: `Attribute Error`, `Type Error`, `Name Error` and `Value Error` are the top errors unique to AI-modified code while the percentage of `Syntax Error` is magnified. Notably, this observation coincide with the findings in previous literature (Rahman et al., 2025; Tambon et al., 2025; Wen et al., 2024; Yin et al., 2023; Jimenez et al., 2023).

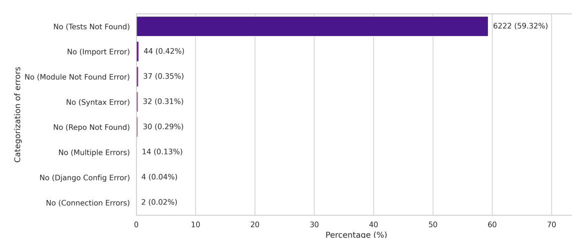


Figure 3: Error distribution in testing human-authored code.

B. Code Marking Prompt

This is the prompt fed to the LLM for the **code identification and marking tasks**.

```
Code Identification and Marking
Prompt
```

```
You will receive a code snippet.
```

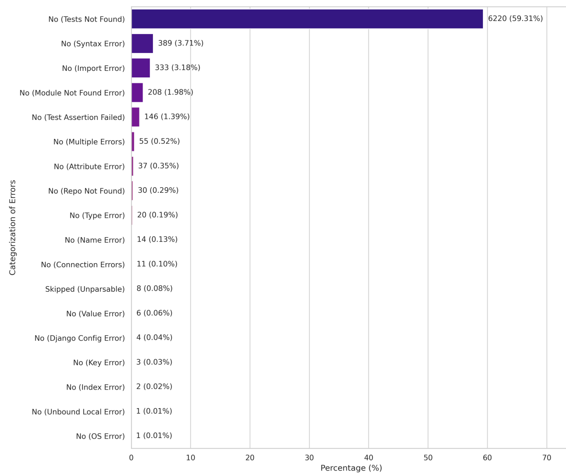


Figure 4: Error distribution in testing AI-modified code.

```
Your task is to remove some parts
of the code, and replace it with
placeholder comments that de-
scribe what the section you re-
moved must do.

Some criteria for replacement:

1. the sections of code must be
   clearly atomic and logically
   separable from surrounding
   code, such that the selected
   code can be cleanly described
   (for example, don't stop the
   deleted section in the middle
   of a for loop).

2. clearly distinguish comments
   added in this manner from
   other comments in the code
   by starting your comment with
   "GENERATE CODE:"

3. Your comments need to be de-
   tailed enough that someone
   without reference to the
   original code could plausibly
   reconstruct the code section
   with similar functionality
   (though not necessarily with
   the same exact syntax)

4. Try to be not super specific
   about the exact implemen-
   tation, the goal is not to
   describe it so specifically
   that there is only one pos-
   sible code string that meets
   the specified requirements

5. Do not replace non-
   significant, non-meaningful
   lines of code. Examples of
   these are:

   • Blank lines
   • Comment-only lines
   • Docstrings
   • Simple print statements
   • Simple brackets/parenthe-
     ses
   • Simple pass statements
   • Setting variables to some
     string, bool or int lit-
     eral

Remember, you are to REMOVE some
parts of the code, and REPLACE
the removed code with your com-
ments.  So the file may not be
a valid, executable code file,
and that's ok.  Return nothing
else besides the edited code
file.  You'll also receive the
approximate percentage of signif-
icant, meaningful lines of code
that should be replaced in this
manner.

[Approximate Percentage In-
serted]
[Code File Inserted]
```

C. Code Writing Prompt

This is the prompt fed to the LLM for the **code writing task**.

Code Writing Prompt

```
We are creating code snippets
that have human-written and AI-
written code mixed together.  You
are responsible for creating
the AI-written portions of the
code snippets.  I will give you
a completely human-written code
snippet, with some sections re-
moved.  the functionality of the
removed sections are described
in inline comments that start
with "GENERATE CODE:".  Your task
is to create code that meets the
requirements described in these
comments.

Return the completed version of
the entire code file, including
both the original contributions
and your contributions.  Remove
all "GENERATE CODE:" comments
from your output, but preserve
other comments.  Return nothing
else besides this complete code
file.

[Code File Inserted]
```