JCBI
Journal of Computing & Biomedical Informatics
ISSN: 2710 - 1606
Volume 12 Issue 01
2026

Research Article
https://doi.org/10.56979/1201/2026/1645

ZeroTraceGPT: A Robust Framework for Identifying Human VS AI-Generated Code

Hamza Zaheer[1], Muhammad Kaleem[1], Muhammad Azhar Mushtaq[1*], and Sayyid Kamran Hussain[2]

[1]Department of Information Technology, Faculty of Computing & IT,University of Sargodha, Sargodha, 40100, Pakistan.
[2]Department of Computer Science, TIMES University, Multan, 60000, Pakistan.
[*]Corresponding Author: Muhammad Azhar Mushtaq. Email: azhar.mushtaq@uos.edu.pk

Received: June 04, 2026 **Accepted:** September 21, 2026

Abstract: Artificial intelligence tools such as ChatGPT emerged in the software development space as useful resources for effectively providing solutions to complex programming problems. These tools can ease and help with writing code; however, with the evolution of AI-generated source code, there is a growing need to identify whether code is written by a human or AI. To address this challenge, we proposed a novel model called ZeroTraceGPT. It has been built upon a new variant of GraphCodeBERT model called GBERT-EVO. Where GBERT-EVO is GraphCodeBERT Evolution that is the innovative variant model introduced in this research. Which uses the data flow patterns and structural integrity to trace the AI-code. ZeroTraceGPT model consists of four key components that contribute to the detection of AI and human-generated code, including: 1) Scraper, which extracts the code data initially, 2) Segmenter, tokenizes the code after the extraction task, 3) Train Guard, component evaluates and trains the model for the detection of the AI and human-written code, and 4) a Labeler, component assesses the train model results and classifies the code as either AI-written or human-written. During our evaluation, ZeroTraceGPT performed better than the previous state of the art GPTSniffer model, which performed poorly and had significant drawbacks. ZeroTraceGPT produced faster, accurate, and more reliable results and outperformed GPTSniffer with an average improvement of 11.75 percentage points across accuracy, precision, recall and F1-score under identical experimental conditions.

Keywords: AI Generated Code; Code Classification; AI Code Detection; CodeBERT; Pre-trained Models; Abstract Syntax Tree

1. Introduction

Large language models (LLMs) have made a large impact on software development [1] in recent years and in academic as well. These pretrained models, such as ChatGPT, which have been trained on large datasets [2], have helped reduce friction and gain efficiencies in the generation of code [3]. The process of writing code is revolutionized by LLMs [4]. LLMs can produce not only syntactically plausible but often semantically meaningful code snippets. LLMs have implications for all development-related tasks. Use of LLMs will reshape the methodology of coding for many software developers, but new challenges are also introduced by LLM development [5].

Even though these models offer so much potential, they have limitations as well [6]. The generated result quality varies based on the input prompt quality or the specific information in the prompt. They can produce good results, but have their training data constraints [7]. The usefulness and capabilities LLMs provide across different applications [8] make them completely adoptable not only in software industry, but increasingly in educational settings. The use of these LLM models has raised some serious concerns in the academic field [9]. Students have started using these tools regularly to finish program assignments, learn programming languages, and receive assistance with problems or projects with difficulty. In recent

years, AI-powered tools like ChatGPT have greatly impacted the academic sector [10]. ChatGPT and similar tools, used as intelligent tutoring systems, are unique to educational contexts. In digital education solutions like ChatGPT can offer consolidated resources to help students build understanding at their own pace [11].

So in academia, LLMs are now discussed frequently [12] and have done an incredible job of being able to generate quality code, which presents serious challenges. Students' competencies become a serious discussion among educators [13]. Students can now get fully functional code packages in seconds, and LLMs like ChatGPT are assisting students in programming assignments [14]. This complicates the process of assessing AI-generated code, which creates a risk for valid evaluation and accurate evidence of student learning outcomes. However, even with the benefits of AI for education, its adoption on a broad scale creates a different concern in academic integrity [15] for educators. Several universities have introduced their guidelines for the appropriate use of these LLMs [16] because their accessibility and use lead to the rise of academic dishonesty [17]. So the use of AI is not only bringing opportunities in academia, but challenges as well [18].

In industry, AI coding tools like ChatGPT can assist in software development [19]. LLM models like these in software engineering have become an important part [20] but they have risks. Developers may insert AI-generated code into production systems without a firm understanding of how it works or evaluating its robustness. While the code may get the job done in isolated testing, it may contain hidden bugs, and inefficient algorithms. So, ChatGPT generated code quality and reliability remain a potential threat and raise concerns [21]. Further complicating matters are the legal issues regarding AI-generated code [22]. Since an important element of LLMs is the datasets [23] and they are trained on large datasets often containing copyrighted repositories.

The majority of these repositories will have followed common coding standards, libraries, and designs. Therefore, the outputs of LLMs are often very similar to the style and formatting of open-source human-created code. Like human-written code, AI-generated code may also contain bugs [24]. This resemblance makes it hard to evaluate if a software code is AI-generated. We must also not set aside the ethical implications of using LLMs in development to save time and effort. While LLMs produce efficiency in the generation of code [25], their convenience may prompt unnecessary dependence on their use, particularly from entry-level developers or students who may gradually become familiar with substituting understanding with automation. This brings us into the larger discussion of responsible and ethical use of AI in education and the software engineering domain.

No doubt, LLMs are one of the remarkable advancements that have been achieved in NLP [26]. They have the potential to be tremendous educational tools and code productivity engines, but there are some implementation and ethical concerns [27]. Their application requires careful navigation of the empirical, ethical, and legal challenges LLMs create. While we may have made considerable advances in detecting LLM-generated text, which is the advanced form of text generated by a machine [28]. But there are no established equivalents in developing robust techniques for AI source code detection because it has different semantics and structure. In order to meet these challenges, an innovative and established new framework of transparency and accountability for the detection of AI-generated source code is needed.

There are continued limitations to our capacity to detect AI-generated code accurately. Most detection tools are based on text produced by large language models (LLMs) through natural language processing (NLP). While these LLM methods can analyze machine prose thoroughly, they do not respond to the formal structure of a programming language. The distinction of AI vs human-produced source code is more challenging. A prominent effort to address this problem was the development of GPTSniffer [29], a state-of-the-art model that operates to detect whether code was generated with AI or human-authored. Despite the progress made by GPTSniffer in this domain, the model has several limitations and weaknesses that decrease its prediction reliability and accuracy.

These limitations indicate that a better and more robust method is needed for AI and human-generated code classification. To overcome these concerns and to address GPTSniffer's limitation issues. We introduce a novel framework in this research called ZeroTraceGPT. It uses a novel variant of model introduced in this research called GBERT-EVO. Where GBERT-EVO means GraphCodeBERT Evolution because of its intelligent approach mechanism and evolve new variant implementation. GBERT-EVO is based on the GraphCodeBERT model [30] which is a graph-based pre-trained model, but has had

significant improvements from its foundation. GBERT-EVO model replaced the sequential text architecture and uses the novel structure aware architecture for better AI-code detection. So, ZeroTraceGPT is a fundamental model to maintain transparency while respecting and adhering to AI guidelines in the academic and software engineering fields.

The remainder of the paper is comprised of the following sections. Section 2 investigates the relevant prior literature surrounding AI-generated outputs and code detection methods. Section 3 outlines the methodology and describes in detail the working of ZeroTraceGPT. Section 4 describes the implementation setup, the datasets used, and the evaluation metrics for the performance of the model. Section 5 presents the results of the evaluation, including metric-wise comparison, error analysis, computational cost, ablation, generalization and robustness experiments, and comparison with recent detectors. Section 6 discusses the implications. In Section 7, concluding aspects of the research are presented, with future directions.

2. Related Work

The We examine existing research work in AI-driven text identification, as well as the detection of AI-generated source code. In the last few years, it has become increasingly common to investigate methods to distinguish between machine-generated and human-generated text and source code in the areas of academic, software development, and related fields. So, the research work [31] introduces a new way to detect machine-generated text called BENADV. The study [32] assesses five transformer-based models' performance for detecting LLM-generated text. Authors in [33] investigates machine learning models for identifying human or AI-generated text. Another study [34] assesses eight public LLM-generated text detectors using student submissions. The research [35] explored the relative performance of AI detectors and humans in identifying paraphrased AI-generated medical writings. The work presented shows that AI content detectors can misclassify human writers as AI-generated [36]. A recent study [37] investigates whether ChatGPT itself can be used as a zero-shot detector of AI-generated text. The survey [38] systematically discusses the current state-of-the-art methods for detecting AI-generated text.

The research project [39] examines the ability of academic staff and Turnitin's AI detector to recognize AI-generated submissions. Research carried out [40] examined the participants' ability to discern between human, AI, and internet-generated texts in educational writing and found limited success. The survey examines [41] recent progress in identifying LLM-generated text by surveying watermarking and statistical, and neural detectors. The authors [42] present MGTBench, the first validated, comprehensive benchmark for machine-generated text detection. Another benchmark is proposed called EvoBench [43], that benchmarks LLM-generated text detectors. Research in [44] demonstrated that current automatic detection tools vary significantly in the ability to detect ChatGPT-generated texts. The study [45] concludes that both novice teachers and experienced teachers were unable to identify essays generated by ChatGPT. Like that, authors at work [46] creates both linguistic and perplexity-based detectors for identifying ChatGPT-generated essays. In the project [47] a multimodal deep learning model, AI-Catcher, is presented, which uses MLP and CNN to detect the presence of ChatGPT-generated scientific texts.

The paper [48] addresses the large issue of detecting ChatGPT-generated "DeepFake" text. Another research work provides a mapping of research [49] on generative AI in education, revealing a focus on text-based generative AI, while multimodal generative AI has largely been ignored. CHEAT [50], a large dataset, is presented to facilitate the development of detection algorithms. Authors [51] introduce MIA-Tuner, an instruction-based approach that utilizes LLMs themselves to enhance membership inference attacks (MIA) used to determine if text was part of an LLM's training data. The research [52] provides datasets and evaluates detection methods for AI-paraphrased texts. A new zero-shot black-box approach [53] detects LLM-generated medical paper introductions using text reordering and similarity score. In [54] MLSDET is presented, a framework using varied large language models and unique metrics to detect AI-generated Chinese text. Another hybrid deep learning model [55] for the detection of AI-generated text is also proposed.

So, none of the mechanisms has been applied to either of these methodologies to identify AI-generated source code, as these approaches were designed specifically for detecting AI-generated text. So, we are particularly interested in newer methodologies developed to detect code produced from AI models such as ChatGPT. The research work [56] assesses existing tools for detecting AI code, but they often perform

poorly, lack robustness, and lack accuracy in the real world. In [57], authors examine seven of the leading AI code detectors on a new Python dataset, but the work is also limited. In another research [58] five detectors were evaluated, and all five performed poorly at differentiating AI-generated code from human-written code. The project [59] examines a range of methods for identifying ChatGPT-generated code in introductory-level CS assignments. An improved frameworks [60],[61] and a lightweight tool [62] for recognizing code vulnerabilities is also proposed. The research work [63] examines the AI-authored Python snippets from the DevGPT corpus.

RU-AI [64], a large-scale multimodal dataset was presented to improve the detection of machine-generated content. AIGCodeSet [65] is a dataset containing AI-generated Python codes and human-written Python codes to identify whether the code was from a human or an AI. A unique zero-shot synthetic code detector [66] is also proposed that utilizes the similarity between the original code and its LLM-rewritten versions. The study [67] examined the separation of code written by humans and code written by ChatGPT, and the study also outlines constraints when detecting AI code. The paper [68] presents an innovative approach to detecting AI-generated student code by applying masking perturbations to high-perplexity areas of code. Another research authors [69] assesses five approaches to zero-shot detection as well.

More recent work has moved beyond the early comparison with GPTSniffer. DetectGPT4Code [71] adapts probability-curvature zero-shot detection to code using a surrogate white-box model; the rewriting-based detector [66] measures the similarity between code and its LLM-rewritten variants; Codetector [69] provides a modular framework and benchmarks five zero-shot methods on 113,776 samples generated by 17 LLMs; Suh et al. [72] fine-tune LLMs and train classifiers over static code metrics and AST-based embeddings, reporting an F1-score of 82.55 together with a feature ablation; and CodeGPTSensor [73] applies contrastive learning for supervised detection. Table 01 summarizes how these methods differ from ZeroTraceGPT. Zero-shot methods need no labelled training data but depend on token-probability statistics, whereas supervised detectors, including GPTSniffer and ZeroTraceGPT, learn from labelled corpora. ZeroTraceGPT additionally builds on a pre-trained encoder that models data-flow relations explicitly. A quantitative comparison with these detectors under identical splits is not part of this study and is left for future work (Section 5.E).

Table 1. Qualitative comparison of AI-generated code detectors.

Method	Approach	Labelled training data	Main signal
GPTSniffer [29]	Fine-tuned CodeBERT classifier	Required	Token-level semantics
DetectGPT4Code [71]	Zero-shot probability curvature with a surrogate white-box model	Not required	Token probabilities
Rewriting detector [66]	Zero-shot similarity between code and its LLM rewrites	Not required	Rewriting similarity
Perplexity masking [68]	Masking perturbations on high-perplexity code regions	Not required	LLM perplexity
Codetector [69]	Modular framework evaluating five zero-shot methods	Not required	Token statistics
Suh et al. [72]	Fine-tuned LLMs; classifiers on static metrics and AST embeddings	Required	Code metrics, AST
CodeGPTSensor [73]	Contrastive-learning detector for program code	Required	Learned representations
ZeroTraceGPT (this work)	Fine-tuned GBERT-EVO (GraphCodeBERT-based) with Train Guard	Required	Data flow and structure

So many approaches to AI-generated code detection have been proposed, but each has serious limitations that make them very ineffective and frail. But GPTSniffer [29] which is considered the best-in-class model, is somehow better designed to distinguish between AI and the human-generated source code. GPTSniffer is built using a pretrained model called CodeBERT [70] that has a large amount of training in relation to natural language and programming languages. GPTSniffer is more focused on these challenges in classification. While GPTSniffer provides a better way to consider the human and AI-generated code classification problem, but is also limited in several ways that also limit the performance of the model itself.

GPTSniffer Weaknesses and Drawbacks:

- One of the major problems associated with GPTSniffer is that it does not have an evaluation plan within its training plan. Without an evaluation plan, it will not understand how it is performing across time, and it won't detect issues such as overfitting.
- In addition, it will not be able to pinpoint when the model performs best. A failure of a monitoring plan means that the model cannot understand performance improvements. The model will keep only a single final checkpoint, which is also usually not its best performance checkpoint. This means that the model's performance will always be fragile and unreliable in the intended space of use, such as for academic integrity checks or professional code audits.
- The training was fully employed with full precision only, but using a mixed precision classification training plan could be productive. Mixed precision can lessen the total compute cost needed and speed up training in comparison to not using mixed precision. When not utilizing mixed precision, GPTSniffer will be less efficient, consume more memory, and require longer training iterations.
- GPTSniffer does not actively adjust code length to account for variability, nor does it incorporate existing token-level attention methods. That may facilitate detecting different forms of more complex semantic relationships across code. This static structure hinders the model's ability to comprehend programming constructs over longer code blocks.
- Another significant limitation is how the code samples will be pre-processed to remove comments. GPTSniffer removes comments for specific programming languages, but what about other programming languages that are supported.
- The model uses the same batch size, thereby also limiting its ability to be flexible and efficient in terms of batch sizing. Adaptive batch sizes have the potential to optimize training by balancing compute load with model convergence, which GPTSniffer's training method ignores.
- The key measures of classification, such as accuracy, precision, recall, and F1-score, are imperative for estimating model performance. GPTSniffer computes those performance measures after the whole training process is completed, so the user has no progression feedback concerning their model, nor a learning reference to modify their training methods along the way.

These constraints undermine GPTSniffer's reliability and efficiency and ultimately its feasibility, so it is not mature enough to be implemented in high-stakes situations that require accurate identification of AI-generated code. Given these limitations, it is possible to visualize how GPTSniffer can be improved upon to create a more reliable, flexible, and effective tool for AI code detection. And this is accomplished through our novel ZeroTraceGPT model. ZeroTraceGPT uses novel Pre-Trained Transformers to counter GPTSniffer's limitations. More specifically, ZeroTraceGPT uses the GBERT-EVO pre-trained model, which is a highly optimized, fine-tuned novel version of the transformer architecture.

The domain-specific focus of ZeroTraceGPT is to not only overcome previous approaches' limitations but also consistently surpass GPTSniffer in multiple AI code detection tasks. With GBERT-EVO's added depth of knowledge of programming structures, ZeroTraceGPT is in a better position to distinguish between AI and human-written code in even the most nuanced circumstances. ZeroTraceGPT also mitigates the limitations of GPTSniffer with our use of dynamic training strategies, more sophisticated attention dimensions to improve semantic understanding, and supporting preprocessing strategies. With these limitations addressed, ZeroTraceGPT far outperforms GPTSniffer in terms of AI and human-generated source code detection.

3. Methodology

ZeroTraceGPT represents significant progress in the detection of AI-generated code by thoughtfully refining every element of the training process and implementation. It not only solves the limitations of GPTSniffer but also builds on cutting-edge approaches based on modern standards. This thoughtful reconstruction yields significant advances in performance in immediate evaluations and also provides a more flexible and elastic model for future usage. At its essence, ZeroTraceGPT is a new high-level machine learning framework that has been trained to tell apart code written by humans versus code written by large language models like ChatGPT. The purpose was to create a language-agnostic and extensible system able to reliably detect AI-written code across a large number of programming languages. The ZeroTraceGPT model uses GBERT-EVO, a novel finely-tuned variant of the GraphCodeBERT model [30].

GBERT-EVO allows the model to assess the relative positions in relation to the program's logical flow. Understanding the relationship of each token with other tokens or nodes in the program is important. GBERT-EVO excels in that it does not merely read the code's syntax; it also understands the structure and data flow of the code. This semantic-level view of code sits within the model architecture, allowing it to account for the logical and structural dependencies involved in programming code. Thus, the model can focus on distinguishing subtle lines of repetitive patterns that separate AI-generated from human-written code. GBERT-EVO for the richer context of code understanding uses the graph augmented representation. This novel variant makes use of its own adaptive training control for the code detection. It goes beyond the binary outcomes of the results to evaluate the model capability. It applies dynamic training based on metric aware. The model also makes use of the new adaptive batch sizing. With this new variant model ZeroTraceGPT capture the structure anomalies better. The model utilizes less computational resources due to its intelligent training approach.

GBERT-EVO is better equipped to encapsulate the many facets of programming languages. It applies structural information along with semantic information to be much more efficient and effective for code understanding. Therefore, the combination of these two strategies collectively enables GBERT-EVO to lead in performance across all code-related purposes. ZeroTraceGPT's capability to analyze and learn such structural patterns enables it to accurately differentiate machine-generated code and has improved precision and recall as compared to a token analysis model. ZeroTraceGPT's training process is more intelligent and more flexible than GPTSniffer. ZeroTraceGPT uses a much more sophisticated and multifaceted approach, integrating a powerful novel model variant with a unique training and validation trait. ZeroTraceGPT also evaluates the information from the code's Abstract Syntax Tree (AST) and Control Flow Graph (CFG).

3.1. Novelty of GBERT-EVO and its architectural differences from GraphCodeBERT:

GBERT-EVO is a task-specific adaptation of GraphCodeBERT [30] and not a new pre-trained encoder. It starts from the pre-trained GraphCodeBERT encoder, whose input combines code tokens with data-flow variables and uses a graph-guided attention mask so that each variable node attends to the tokens it is derived from [30]. GBERT-EVO keeps this structure-aware input representation and differs from the original in four respects (Table 02): (i) a dedicated binary human/AI classification head replaces the task-agnostic output of pre-training; (ii) the Scraper normalizes the input so that surface style is suppressed and the encoder relies on structure; (iii) a metric-aware training controller, the Train Guard, validates the model after every epoch and selects the checkpoint by F1-score; and (iv) mixed-precision training [75] and adaptive batch sizing reduce computational cost. The contribution is therefore the integration of these elements for AI-code detection, not a new pre-training objective.

Table 2. GraphCodeBERT versus GBERT-EVO.

Aspect	GraphCodeBERT [30]	GBERT-EVO (this work)
Origin	Pre-trained encoder (masked language modelling, edge prediction, node alignment)	Fine-tuned from GraphCodeBERT weights; no new pre-training objective
Input	Code tokens plus data-flow variables with graph-guided attention mask	Same structure-aware input, after Scraper normalization
Output layer	Task-agnostic representation	Binary human/AI classification head
Training control	Fixed training schedule	Per-epoch validation; Train Guard (F1-based early stopping, best-checkpoint restore)
Numeric precision	Full precision	Mixed precision [75]
Batch size	Fixed	Adaptive batch sizing

Why data flow and structure should help detection. Surface cues such as identifier names, comments and formatting are easy to imitate or alter, whereas data-flow edges (where each variable's value originates) and syntactic structure reflect how a solution is organized. GraphCodeBERT showed that adding data flow improves code-understanding tasks [30]. For detection, a component-level ablation that removes the data-flow input is not part of this study (Section 5.E), so the benefit of data flow here is supported by [30] and by the overall comparison in Section 5, not by an isolated test.

ZeroTraceGPT improves upon many of the weaknesses of GPTSniffer and incorporates some major improvements. One key improvement is the inclusion of an evaluation phase after each training epoch.

This emphasizes robustness during training and helps mitigate against overfitting, such that the model will generalize much more appropriately to unseen data. The training mechanism of ZeroTraceGPT is also much more stable and efficient. By employing larger batch sizes, the model encounters more stable optimization. Continuing, mixed precision training is included for greater speed and improved memory utilization. The result is faster training times, with no degradation in accuracy or other model characteristics. In this way, ZeroTraceGPT is more resource-efficient and has the capacity to scale across larger datasets and increasingly complex codebases. ZeroTraceGPT's architecture is modular and well-designed, as it has four key components that work in order to process and classify code snippets as shown in Figure 01.

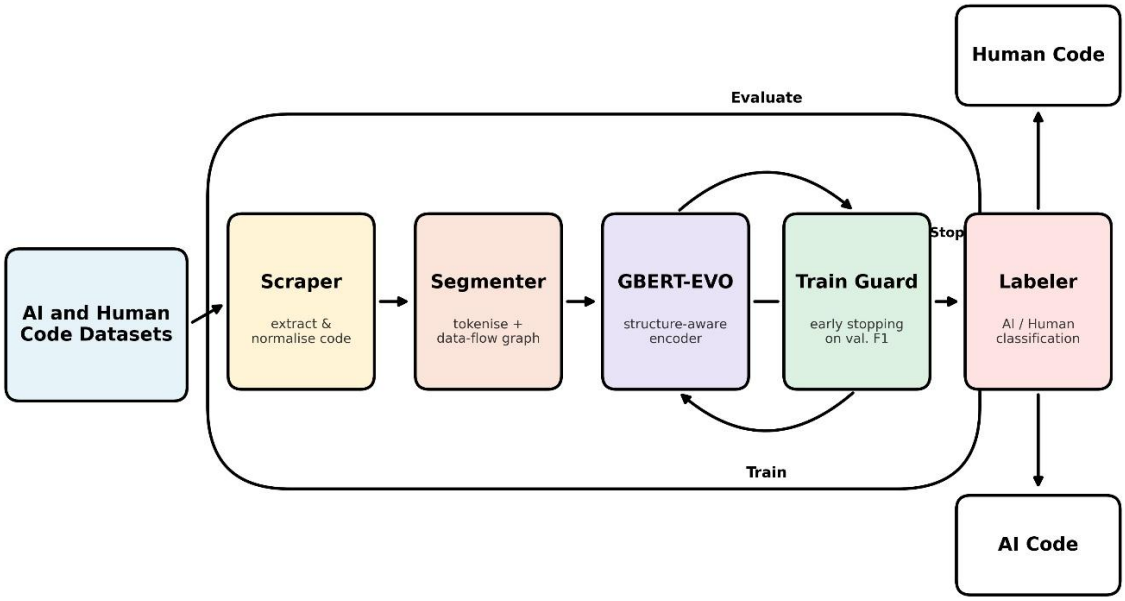


Figure 1. ZeroTraceGPT components.

3.2. Scraper

The Scraper module is an integral part of the ZeroTraceGPT framework. The Scraper module uses sophisticated means of extracting code, converting it into usable code for the model to evaluate. Specifically, the Scraper applies more advanced preprocessing techniques and tunable transformation rules that expand the diversity and representation of the training data. In software engineering, human authoring can be indicated by stylistic fingerprints in variable naming, inline comments, and import statements. By concentrating on these style aspects, Scraper hopes to help the model learn to detect the difference between human and AI authorship.

Scraper is not only using new rules, but also many rewriting rules are derived from the state-of-the-art model but modified with a more sophisticated approach. In summary, the point of transformation is to avoid introducing more noise into the detection model by changing levels of irrelevant surface detail and to allow the model to work on more abstract semantic and structural ideas in the code. Specifically, Scraper minimizes noise from stylistic features. In doing so, Scraper creates more robust and generalizable features for the detection model. More importantly, the detection model will learn to differentiate between human-written code and AI-written code even better than previously.

3.3. Segmenter

Once the source code has been extracted and preprocessed, it is passed to the Segmenter module, which is an essential part of the ZeroTraceGPT model. The Segmenter's job is to convert raw code snippets into token sequences that satisfy the input constraints of the GBERT-EVO model. Tokenization provides the link between the raw text of unstructured source code to structured contextual information or semantic content. That is how the model understands the content of the code. To its simplest function, the Segmenter takes the code and divides it into the smallest meaningful semantic unit, which is a token. The strategy of the tokenization is better aligned with structural and semantic representation of code.

This tokenization process is quite different than simply segmenting the text. It allows the model to not only identify discrete pieces of the code but also to understand the more abstract structural relationships and semantic meaning within the snippet. Both humans and AI can write code that performs similar tasks; they often differ in how they express that logic, conventions, and overall stylistic features, and tokenization is the first step that allows the model to find those differences. Additionally, as the Segmenter moves from a segment of code to tokens with context, it allows the model to recognize not just explicit programming structures, but also implicit coding behaviors or patterns that elicit a coding style by the author. The Segmenter is an essential component of the overall purpose of ZeroTraceGPT, which is to reliably classify code as human-generated or AI-generated. Without the explicitly defined tokenization process, the model would not have the necessary understanding to make accurate distinctions.

3.4. Train Guard

The next and most important component is the Train Guard, which is based on the GBERT-EVO. GBERT-EVO makes use of the Segmenter output, which becomes its input for training. Once the GBERT-EVO model is trained, the training results become inputs to the Train Guard components. The Train Guard is actually an early stopping module to reduce overfitting and improve performance. This module actively evaluates whether the model's performance, on an evaluation dataset, improves during training. It achieves this functionality by monitoring the F1-score on the evaluation dataset after every epoch is completed. A patience value is defined, which indicates that if the F1-score does not improve over defined consecutive epochs, the training of the model will be automatically stopped. In this case, avoiding additional epochs is beneficial because it can avoid wasting time and resources that do not improve the ability of the model. The model is first trained, then evaluation is done by Train Guard, and if it is performing better, it proceeds to further training, and if no performance improvement, the Train Guard stops the process, so this is how this loop cycle works.

The use of this component provides several significant advantages, which enable ZeroTraceGPT to outperform. First, it Prevents Overfitting, which is the main advantage. The model stops training once the best model has been found, so it isn't overly fitted to the training data. Second, the Saving of Resources is achieved as training deep learning models can be time-consuming and costly. Early stopping can end the training process when it is unlikely that there will be any further improvements, thus saving time, energy, and resources. Third, it provides the Best Model Selection, ensuring that even if training stops occur after some non-improving epochs, the final model that was saved is the model that achieved the overall best F-1 score. Through this intelligent and automated learning component, ZeroTraceGPT will be a more efficient and effective classifier because ZeroTraceGPT is built to have optimal performance without overfitting.

3.5. Labeler

Finally, after the model has been trained, the Labeler component is responsible for making classification decisions. The Labeler indexes already-trained code snippet inputs and uses the trained model to determine which pieces of code were created by a human and which were created by an AI. The Labeler component makes use of Train Guard outputs to provide accurate and efficient classification for reliable results with different code samples. Labeler is supervised in the training process while using labeled datasets of human and AI-written code snippets

During the course of training, the model will iteratively improve to capture subtle underlying patterns and stylistic cues that will distinguish human developers from AI generators, even when there is no syntactical or substantive difference. Distinctions in style and structure allow the Labeler to be effective. The Labeler received a lot of its capacity through Train Guard pretrained knowledge as a result of the pretraining on a large and diverse programming corpus. This pretrained foundation allows the Labeler to interpret code snippets with a deeper understanding of its logic rather than superficial indicators. To make a better understanding of how the ZeroTraceGPT framework model works, this algorithm explains its working mechanism.

Novel contributions versus implementation modules:

Table 3. Role and novelty of each ZeroTraceGPT component.

Component	Role	Status	Basis
Scraper	Extracts code and normalizes it to reduce stylistic noise	Adapted module	Preprocessing and rewriting rules

Segmenter	Converts code into token sequences and data-flow input for GBERT-EVO	Implementation module	derived from prior work [29] and modified Tokenization aligned with the GraphCodeBERT input format [30]
Train Guard	Per-epoch validation, F1-based early stopping, best-checkpoint restore	Novel as an integrated controller; early stopping itself is standard	Metric-aware control combined with adaptive batch size and mixed precision
Labeler	Runs the trained model on test code and assigns the AI/Human label	Implementation module	Classification head and labelling of the test set

In summary, the methodological contribution lies in the Train Guard and in the integration of the components around GBERT-EVO. The Scraper, Segmenter and Labeler are implementation modules that make the pipeline complete and reproducible.

We can separate this algorithm working into three different phases: PREPARE_DATA, TRAIN_MODEL, and LABEL_CODE. The first phase, PREPARE_DATA, encompasses the different actions of the pipeline. First, the Scraper takes an unstructured raw dataset and converts it into a usable format. The Segmenter takes scraped code and segments it into meaningful chunks. Finally, the prepared data will be partitioned into three sets of training, validation, and testing. The second step, TRAIN_MODEL the learning process whereby the GBERT-EVO and Train Guard will both be doing their job. The GBERT-EVO model is trained on the training set by going through the training data iteratively for a specified number of times. After each epoch, the model's performance on the separate validation set would be assessed. This is where the EarlyStopCallback, the Train Guard, becomes relevant.

The Train Guard is an important part of preventing overfitting. The primary responsibility of the Train Guard is it store the reference to the best performance they have seen so far. When the performance on the validation set does not improve by at least 'min_delta' for a certain number of epochs (patience = n), the Train Guard concludes that it is not seeing adequate learning happening. Therefore, it should stop training the model. When the Train Guard identifies that the training process has ended, the Model will save the best performing model to be passed into the final phase of the pipeline, LABEL_CODE. In the LABEL_CODE phase, the Labeler will take the trained model and make predictions on the independent test set of data. The predictions will then be labeled and classified as either "AI Code" or "Human Code" based on what the model learnt during training.

ZeroTraceGPT Algorithm 1

```
FUNCTION PREPARE_DATA (raw_datasets):     // 1. Data Preparation Phase
    scraped_data = SCRAPE_CODE (raw_datasets)        // Scraper
    segmented_data = SEGMENT_CODE (scraped_data)        // Segmenter
    training_set, validation_set, test_set = SPLIT_DATA (segmented_data)        // Split data
    RETURN training_set, validation_set, test_set
FUNCTION TRAIN_MODEL (model, training_data, validation_data, hyperparameters):     // 2. Model Training
    model_instance = INITIALIZE_GBERT_EVO (model, hyperparameters)        // Initialize GBERT-EVO
    early_stop_callback = EARLY_STOP_CALLBACK (min_delta=0.001, patience=n)        // EarlyStopCallback
    best_validation_loss = INFINITY
```

```
epochs_without_improvement = 0
FOR each epoch in total_epochs:
    current_loss = TRAIN  (model_instance, training_data)  // Train the model
```

4. Experimental Setup

To assess the performance and efficiency of ZeroTraceGPT, we need to directly compare ZeroTraceGPT against state-of-the-art model. Although ZeroTraceGPT has been tested against several different datasets of human-written and AI-generated code, a fair assessment of ZeroTraceGPT against GPTSniffer must be conducted with the same source code datasets used by GPTSniffer [29]. Those datasets of source code contain code samples from human authors and AI-generated code. The human-written code was sourced from public GitHub repositories and had multiple programming styles, genres, and levels of coding experience. In the case of code generated by AI, the dataset was created by multiple systematic prompts of ChatGPT, and was trying to capture the most AI coding styles from these prompts.

Dataset Construction and Labelling:

The experiments use the dataset released with GPTSniffer [29], so that both detectors are compared on identical data. Human-written samples were collected from public GitHub repositories, and AI-generated samples were produced by systematically prompting ChatGPT with comparable programming tasks. Labels are assigned by provenance: a sample is labelled human if it was taken from a repository and AI if it was returned by ChatGPT for a prompt. No manual re-annotation was performed by the authors. Precision, recall and F1-score are support-weighted averages over the two classes, which is consistent with weighted recall being equal to accuracy in Table 04. Per-sample class counts follow the release of [29].

Evaluation Settings:

The evaluation for both ZeroTraceGPT and GPTSniffer was done under the same conditions in terms of configuration, training, specific datasets, and evaluation settings. The evaluation is carried out using the computational resources available to truly make the comparison as valid and unbiased as possible. The dataset was divided into three groups for evaluation: 80% for training, 10% for validation, and 10% for testing. The predictions were compared against the true and false labels of human-written or AI-generated code. From these predictions, we derived the classification metrics of Accuracy, Precision, Recall, and F1-Score. By using these metrics, we can quantitatively elaborate on how good ZeroTraceGPT is in comparison to GPTSniffer.

Data Leakage Controls:

Overlap between training and test partitions can inflate results. The data are inherited from [29], and the released metadata do not allow us to rule out overlap of repositories, authors or generation prompts between the training and test partitions. No repository-level or prompt-level audit was performed in this study, so leakage is treated as a threat to validity: the reported scores may be optimistic if related samples appear on both sides of the split. A re-split grouped by repository and by prompt, together with removal of near-duplicate code, is the appropriate check and is left for future work.

Statistical Protocol:

Both detectors were trained and evaluated on the same partitions with the same evaluation code, as described above. Differences are quantified with 95% Wilson confidence intervals and an exact test on correct and incorrect counts (Section 5.A). Multi-seed variance and paired tests such as McNemar's test [74] require retained per-sample predictions and repeated training runs, and are left for future work.

5. Results and Evaluation

The evaluation results indicate and compare the performance of ZeroTraceGPT and GPTSniffer directly. This comparison is imperative to understand the effectiveness of ZeroTraceGPT and how well it performs relative to GPTSniffer. So, both ZeroTraceGPT and GPTSniffer are trained, validated, and tested in multiple evaluations on the datasets to explore how each model operates under different assessment conditions. One of the complete evaluation result visualizations of both models is discussed, which shows the strengths and weaknesses of each of the two models.

A. Classification Metrics:

The bar graph shown in Figure 02 provides a comparison of the two models, GPTSniffer and ZeroTraceGPT, over the four-performance metrics: Accuracy, F1-Score, Precision, and Recall. The paired bar chart has red bars for GPTSniffer and blue bars for ZeroTraceGPT. Each of the scores for each metric is clearly marked above the bar to facilitate comparison. As shown in the Figure, ZeroTraceGPT has a much greater Accuracy at 0.9231 compared to GPTSniffer at 0.8022; thus, ZeroTraceGPT has been more successful at classifying both the positive and negative cases correctly. Accuracy is a reliable indicator when the classes are balanced.

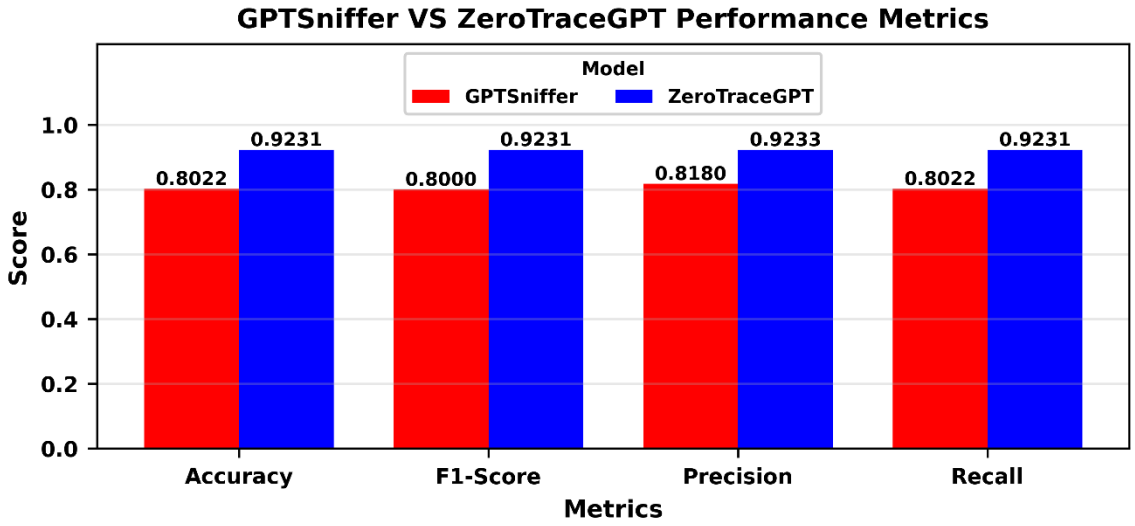


Figure 02. Performance metrics comparison.

ZeroTraceGPT has an F1-score of 0.9231 compared to GPTSniffer's 0.8000. This again suggests that ZeroTraceGPT is indeed not only accurate for overall correctness, but it also has a better balance of Recall and Precision. The graph indicates that ZeroTraceGPT has a higher Precision of 0.9233 compared to GPTSniffer's 0.8180. This means that when ZeroTraceGPT predicts a case as positive, it is more often correct. ZeroTraceGPT has a Recall score of 0.9231, which is also greater than GPTSniffer with a Recall of 0.8022, meaning ZeroTraceGPT is much better at finding all instances of AI-generated code than GPTSniffer. There is no question that ZeroTraceGPT outperformed the GPTSniffer on every measure of performance metrics.

Table 04 gives the metric-wise results. The reported average improvement of 11.75% is the mean of the absolute gains in percentage points for accuracy (12.09), F1-score (12.31), precision (10.53) and recall (12.09). In relative terms the gains range from 12.9% to 15.4%. GPTSniffer was trained and evaluated on the same partitions with the same evaluation script as ZeroTraceGPT.

Table 4. Metric-wise comparison of GPTSniffer and ZeroTraceGPT.

Metric	GPTSniffer	ZeroTraceGPT	Gain (pp)	Relative gain (%)
Accuracy	0.8022	0.9231	12.09	15.07
Precision	0.8180	0.9233	10.53	12.87
Recall	0.8022	0.9231	12.09	15.07
F1-score	0.8000	0.9231	12.31	15.39
Mean	-	-	11.75	-

Statistical significance. Because the test partition is small (the accuracies of 0.9231 and 0.8022 correspond to 84 and 73 correct predictions out of 91 test samples), sampling uncertainty is large. The 95% Wilson confidence interval for accuracy is 0.923 [0.850, 0.962] for ZeroTraceGPT and 0.802 [0.709, 0.871] for GPTSniffer. An unpaired Fisher exact test on correct and incorrect counts gives p = 0.030. The intervals overlap slightly, so the evidence of improvement is suggestive but not conclusive; paired McNemar tests [74] and multi-seed results are needed for a definitive statement (Section 5.E).

B. Confusion Matrix:

A direct comparison of the confusion matrices for GPTSniffer and ZeroTraceGPT is shown in Figure 03, which provides tangible insights into the successes and shortcomings in classification performance. The confusion matrix for GPTSniffer shows significant asymmetry in performance. It has a high true positive rate of 91% for code correctly assessed as generated by ChatGPT. However, it has poor performance on code authored by humans. The confusion matrix indicates a steep 30% false positive rate, which means that it misclassified human-written code and classified it as generated by ChatGPT. This represents a disadvantage as it can lead to distrust in the system and inaccurate results in a real-world environment. The true negative rate is 70%, which is less than expected.

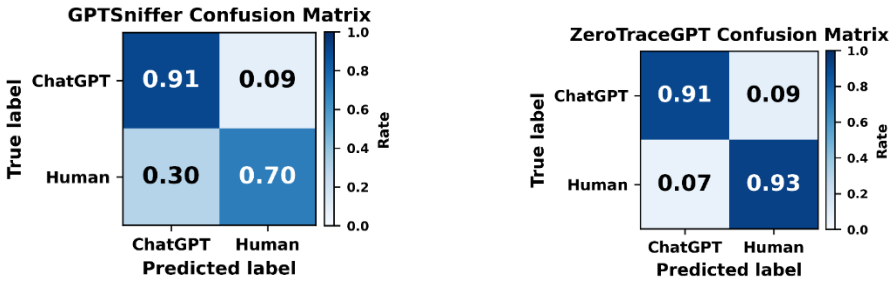


Figure 3. Confusion matrices.

In comparison to it, the confusion matrix of ZeroTraceGPT illustrates a more balanced and superior performance. It mimics GPTSniffer in its identification of ChatGPT-generated content with a 91% accuracy, indicating similarly capable performance in that area. The true strength of ZeroTraceGPT is in its identification of human-written code. It maintains a false positive rate of only 7%, which is impressive when compared with the 30% false positive rate for GPTSniffer, which decreases the likelihood of falsely classifying human-written code as AI-generated code. ZeroTraceGPT maintains a True Negative rate for human-generated code at 93%, which represents a superior gain from GPTSniffer.

False-positive and false-negative analysis. For ZeroTraceGPT the false-positive rate (human code labelled AI) is 7% and the false-negative rate (AI code labelled human) is 9%; for GPTSniffer they are 30% and 9%. The two detectors therefore miss the same share of AI code, and the improvement comes from fewer false accusations of human authors. In an academic-integrity setting this is the costlier error: roughly 7 of 100 human-written submissions would still be flagged, so the output should be treated as evidence for review and never as proof of misconduct, and the decision threshold of the Labeler should be set conservatively. With a small test partition, each misclassified sample shifts these rates by about two percentage points. A qualitative inspection of individual misclassified samples is not part of this study and is left for future work.

C. ROC Curve:

A ROC (Receiver Operating Characteristic) Curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The Area Under the Curve (AUC) provides a single scalar metric to describe the performance of the model being evaluated. The ROC curves for the two models are shown in Figure 04. The dashed line provides context for a random classifier.

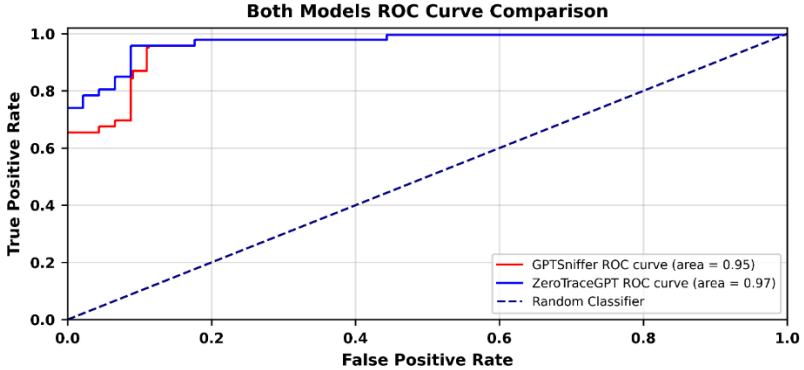


Figure 4. ROC curve comparison.

The red curve shows the performance of the GPTSniffer model and shows that it has an AUC of 0.95. Whereas the blue curve for ZeroTraceGPT has an AUC of 0.97, this is higher than GPTSniffer's AUC. This means that ZeroTraceGPT was a better classifier overall than GPTSniffer. A higher AUC indicates a better ability of the ZeroTraceGPT model to make positive classifications with the fewest false positives. The ZeroTraceGPT curve is always higher than or equal to the GPTSniffer curve over the entire plot. Since the true positive rate is always greater with ZeroTraceGPT for any given false positive rate, the ROC shows that ZeroTraceGPT is the superior model.

D. Training and Inference Time:

GPTSniffer and ZeroTraceGPT perform very differently as far as training runtime and inference time are concerned. Figure 05 shows the training time comparison for both models. GPTSniffer takes about 1203.48 seconds to train the model, which is the length of time that the model took to train from the source dataset. Now, comparing that to ZeroTraceGPT, which trained faster than GPTSniffer, training at 1177.89 seconds. ZeroTraceGPT, therefore, trained 25.59 seconds (2.1%) faster than GPTSniffer. This difference is small and comes from a single run, so it should not be attributed to the training strategy with confidence; the clearer efficiency difference is at inference time.

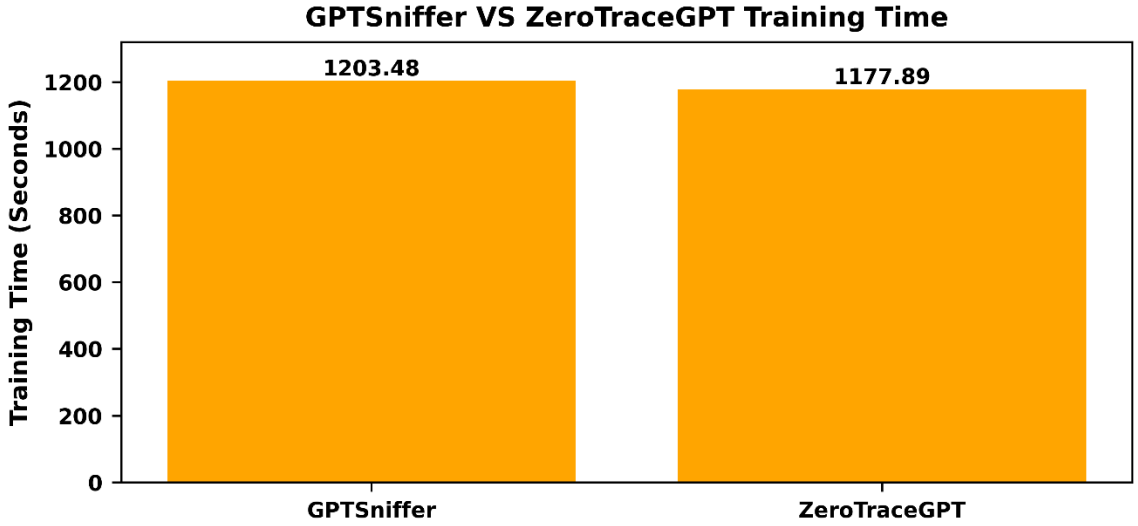


Figure 5. Training time comparison.

If we look at the inference time comparison of the two models shown in Figure 06, it can be seen that GPTSniffer takes 0.00498 seconds to predict an inference for the data we feed it. Whereas ZeroTraceGPT inference time was intensely lower at 0.00199 seconds per inference. This is a 60.0% reduction in prediction time relative to GPTSniffer, which means ZeroTraceGPT is very practical and efficient in modeling predictions post-training. ZeroTraceGPT is not only training faster than GPTSniffer, but it's also taking less time to make the final prediction about the source code.

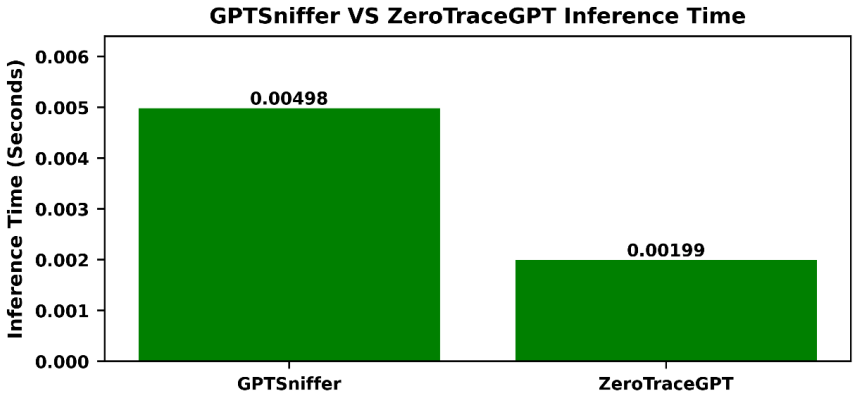


Figure 06. Inference time comparison.

Table 05 summarizes the computational requirements. Both detectors use a base-size encoder of about 125 million parameters (CodeBERT [70] and GraphCodeBERT [30]), so the speed difference is unlikely to

come from model size. Mixed-precision training can lower memory use and training time [75], and the Train Guard may stop training early. The inference gap should be verified with warm-up runs and repeated measurements on the same hardware.

Table 5. Computational requirements.

Item	GPTSniffer	ZeroTraceGPT
Training time (s)	1203.48	1177.89
Inference time per sample (s)	0.00498	0.00199
Trainable parameters	about 125 M (CodeBERT-base)	about 125 M (GraphCodeBERT-base)

E. Scope and Limitations of the Evaluation:

The evaluation in this study is a controlled head-to-head comparison with GPTSniffer on the dataset of [29] under identical conditions. Several further evaluations are not part of this study, and the corresponding claims are therefore not made: a component-level ablation of GBERT-EVO; testing on independent datasets such as AIGCodeSet [65]; testing on code from unseen LLM versions and families and in other programming languages; robustness to obfuscation, paraphrasing, manual editing of AI code and mixed human and AI files [56]; a quantitative comparison with the recent detectors in Table 01 under identical splits; and multi-seed variance with paired significance tests [74]. The results thus support the statement that, on this dataset and for ChatGPT-generated code, ZeroTraceGPT classifies more accurately than GPTSniffer and makes fewer false accusations of human authors. They do not establish robustness or generality, and the small test partition limits the precision of every estimate. These experiments are the main directions for future work.

6. Discussion

The overall comparative evaluation of GPTSniffer and ZeroTraceGPT performance using multiple metrics shows that ZeroTraceGPT is determined to be the better of the two options. Looking at the overall reported scores for each model, as presented in the comparative visualizations, provides the necessary context for understanding the two models' performance. We can clearly see the pattern of consistently higher performance, which suggests ZeroTraceGPT is a more robust and reliable classification option. The final integration of all analyses leads us to the conclusion that ZeroTraceGPT achieves greater overall reliability by minimizing false positives while maintaining a high true positive rate.

ZeroTraceGPT presents an opportunity on this front, offering a more consistent way to determine whether programming code comes from a human author or is AI-generated than current tools like GPTSniffer [29]. Results from this study show that ZeroTraceGPT outperforms GPTSniffer across many critical measures, and this is valuable in providing transparency to and maintaining the integrity of academic and software development practices. It is important to understand that this tool is not anti-AI but is meant as a mechanism to support a meaningful and responsible use of generative AI technology. While the results are promising, further evaluation is needed to see how ZeroTraceGPT performs on a wider array of programming languages. As LLMs evolve to create code that looks more like code written by a human, ZeroTraceGPT may need fine-tuning in order to recognize AI-generated code effectively.

Dataset bias and detector reliability. All results come from a single dataset in which human code was collected from public GitHub repositories and AI code was generated by ChatGPT, evaluated on a small test partition. A detector trained on such data may learn dataset-specific artefacts, for example prompt phrasing, task type, comment conventions or code length, instead of general properties of AI-generated code, and detectors may degrade on newer LLMs and other settings [56], [72]. The reported accuracy therefore applies to the evaluated distribution, and cross-dataset and cross-model experiments, which are not part of this study (Section 5.E), are needed to measure this gap.

Edited and mixed code. Real submissions are rarely pure: AI-generated code is often edited, refactored or combined with human-written code, which blurs the stylistic and structural cues. ZeroTraceGPT classifies a whole snippet and returns a binary label, so it cannot indicate which part is AI-written, and its behaviour on edited or hybrid code has not been evaluated (Section 5.E). Obfuscation and LLM paraphrasing are plausible evasion strategies [56].

Practical use. With a false-positive rate of 7%, a detector flag is a reason for human review and not proof of authorship. Institutions should combine it with other evidence, define appeal procedures, and avoid automated sanctions.

7. Conclusion

Since the launch of ChatGPT, it has become increasingly apparent that ChatGPT has been capable of assisting with a range of software engineering tasks, especially in generating source code. With the availability of AI-generated code, there is growing concern about distinguishing human-written and AI-generated code, leading to a need for tools like ZeroTraceGPT. This research introduces ZeroTraceGPT, an advanced, practically enhanced novel tool to assist in AI-generated source code detection. It uses novel GBERT-EVO model architectural structure, which incorporates unique classification layers and stylistic analysis to differentiate human-generated code from that which was generated from AI. ZeroTraceGPT model consists of four key components, including 1) Scraper, which performs data extraction tasks, 2) Segmenter, which does tokenization, 3) Train Guard performs the continued evaluation and trains the model, and 4) a Labeler component, which classifies the code as human or AI-generated. To evaluate the performance of ZeroTraceGPT, we performed numerous assessments in which ZeroTraceGPT outperformed state-of-the-art GPTSniffer with an average improvement of 11.75 percentage points across accuracy, precision, recall and F1-score on the evaluated dataset. These results indicate that, on the evaluated dataset, ZeroTraceGPT offers a more consistent and stable methodology for detecting AI code. While results are favorable, we acknowledge their limits: the evaluation uses a single dataset and a small test partition, and component-level ablation, generalization to other datasets, LLMs and languages, and robustness to obfuscation and code editing remain to be evaluated. More sophisticated machine learning techniques and a variety of datasets can be utilized to enhance and enrich the detection performance of the model.

References

1. C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying LLM-Based Software Engineering Agents," Proc. ACM Softw. Eng., vol. 2, no. FSE, pp. 801–824, Jun. 2025, doi: 10.1145/3715754.
2. C. Zhou et al., "A comprehensive survey on pretrained foundation models: a history from BERT to ChatGPT," Int. J. Mach. Learn. Cybern., Nov. 2024, doi: 10.1007/s13042-024-02443-6.
3. M. Hamza, D. Siemon, M. A. Akbar, and T. Rahman, "Human-AI Collaboration in Software Engineering: Lessons Learned from a Hands-On Workshop," in Proceedings of the 7th ACM/IEEE International Workshop on Software-intensive Business, Lisbon Portugal: ACM, Apr. 2024, pp. 7–14. doi: 10.1145/3643690.3648236.
4. D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help With Code Understanding," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. doi: 10.1145/3597503.3639187.
5. X. Chen, C. Gao, C. Chen, G. Zhang, and Y. Liu, "An Empirical Study on Challenges for LLM Application Developers," ACM Trans. Softw. Eng. Methodol., vol. 34, no. 7, pp. 1–37, Sep. 2025, doi: 10.1145/3715007.
6. E. Sanu, T. K. Amudaa, P. Bhat, G. Dinesh, A. U. Kumar Chate, and R. K. P, "Limitations of Large Language Models," in 2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS), Bengaluru, India: IEEE, Nov. 2024, pp. 1–6. doi: 10.1109/CSITSS64042.2024.10817070.
7. F. Nuzhat, K. Shivashankar, and N. Steinmetz, "Enhancing Question Answering Systems with Generative AI: A Study of LLM Performance and Error Analysis," in Knowledge Graphs and Semantic Web, vol. 15459, S. Tiwari, B. Villazón-Terrazas, F. Ortiz-Rodríguez, and S. Sahri, Eds., in Lecture Notes in Computer Science, vol. 15459. , Cham: Springer Nature Switzerland, 2025, pp. 242–258. doi: 10.1007/978-3-031-81221-7_17.
8. Z. Jiang et al., "L4: Diagnosing Large-scale LLM Training Failures via Automated Log Analysis," in Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, Clarion Hotel Trondheim Trondheim Norway: ACM, Jun. 2025, pp. 51–63. doi: 10.1145/3696630.3728531.
9. I. Adeshola and A. P. Adepoju, "The opportunities and challenges of ChatGPT in education," Interact. Learn. Environ., vol. 32, no. 10, pp. 6159–6172, Nov. 2024, doi: 10.1080/10494820.2023.2253858.
10. Y. Albadarin, M. Saqr, N. Pope, and M. Tukiainen, "A systematic literature review of empirical research on ChatGPT in education," Discov. Educ., vol. 3, no. 1, p. 60, May 2024, doi: 10.1007/s44217-024-00138-2.
11. H. Uğraş, M. Uğraş, S. Papadakis, and M. Kalogiannakis, "ChatGPT-Supported Education in Primary Schools: The Potential of ChatGPT for Sustainable Practices," Sustainability, vol. 16, no. 22, p. 9855, Nov. 2024, doi: 10.3390/su16229855.
12. R. Khojah, M. Mohamad, P. Leitner, and F. G. De Oliveira Neto, "Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice," Proc. ACM Softw. Eng., vol. 1, no. FSE, pp. 1819–1840, Jul. 2024, doi: 10.1145/3660788.
13. V. Grande, N. Kiesler, and M. A. Francisco R., "Student Perspectives on Using a Large Language Model (LLM) for an Assignment on Professional Ethics," in Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, Milan Italy: ACM, Jul. 2024, pp. 478–484. doi: 10.1145/3649217.3653624.
14. C. A. G. D. Silva, F. N. Ramos, R. V. De Moraes, and E. L. D. Santos, "ChatGPT: Challenges and Benefits in Software Programming for Higher Education," Sustainability, vol. 16, no. 3, p. 1245, Feb. 2024, doi: 10.3390/su16031245.
15. I. M. García-López, C. S. González González, M.-S. Ramírez-Montoya, and J.-M. Molina-Espinosa, "Challenges of implementing ChatGPT on education: Systematic literature review," Int. J. Educ. Res. Open, vol. 8, p. 100401, Jun. 2025, doi: 10.1016/j.ijedro.2024.100401.
16. G. F. N. Mvondo, B. Niu, and S. Eivazinezhad, "Exploring the ethical use of LLM chatbots in higher education," J. Bus. Res., vol. 200, p. 115575, Nov. 2025, doi: 10.1016/j.jbusres.2025.115575.
17. M. B. Mutanga, J. Msane, T. N. Mndaweni, B. B. Hlongwane, and N. Z. Ngcobo, "Exploring the Impact of LLM Prompting on Students' Learning," Trends High. Educ., vol. 4, no. 3, p. 31, Jun. 2025, doi: 10.3390/higheredu4030031.
18. D. Weber-Wulff et al., "Testing of detection tools for AI-generated text," Int. J. Educ. Integr., vol. 19, no. 1, Dec. 2023, doi: 10.1007/s40979-023-00146-z.
19. A. I. Champa, M. F. Rabbi, C. Nachuma, and M. F. Zibran, "ChatGPT in Action: Analyzing Its Use in Software Development," in Proceedings of the 21st International Conference on Mining Software Repositories, Lisbon Portugal: ACM, Apr. 2024, pp. 182–186. doi: 10.1145/3643991.3645077.
20. A. Rajbhoj, A. Somase, P. Kulkarni, and V. Kulkarni, "Accelerating Software Development Using Generative AI: ChatGPT Case Study," in Proceedings of the 17th Innovations in Software Engineering Conference, Bangalore India: ACM, Feb. 2024, pp. 1–11. doi: 10.1145/3641399.3641403.
21. Liu et al., "Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 5, pp. 1–26, Jun. 2024, doi: 10.1145/3643674.
22. W. Ma, Y. Song, M. Xue, S. Wen, and Y. Xiang, "The 'Code' of Ethics: A Holistic Audit of AI Code Generators," IEEE Trans. Dependable Secure Comput., vol. 21, no. 5, pp. 4997–5013, Sep. 2024, doi: 10.1109/tdsc.2024.3367737.
23. F. Du et al., "A Survey of LLM Datasets: From Autoregressive Model to AI Chatbot," J. Comput. Sci. Technol., vol. 39, no. 3, pp. 542–566, May 2024, doi: 10.1007/s11390-024-3767-3.

24. F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: an empirical study," Empir. Softw. Eng., vol. 30, no. 3, p. 65, May 2025, doi: 10.1007/s10664-025-10614-4.
25. K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation," in Proceedings of the 21st International Conference on Mining Software Repositories, Lisbon Portugal: ACM, Apr. 2024, pp. 167–171. doi: 10.1145/3643991.3645074.
26. A. Adel, A. Ahsan, and C. Davison, "ChatGPT Promises and Challenges in Education: Computational and Ethical Perspectives," Educ. Sci., vol. 14, no. 8, p. 814, Jul. 2024, doi: 10.3390/educsci14080814.
27. S. Rawas, "ChatGPT: Empowering lifelong learning in the digital age of higher education," Educ. Inf. Technol., vol. 29, no. 6, pp. 6895–6908, Apr. 2024, doi: 10.1007/s10639-023-12114-8.
28. S. Fariello, G. Fenza, F. Forte, M. Gallo, and M. Marotta, "Distinguishing Human From Machine: A Review of Advances and Challenges in AI-Generated Text Detection," Int. J. Interact. Multimed. Artif. Intell., vol. 9, no. 3, p. 6, 2025, doi: 10.9781/ijimai.2024.12.002.
29. [P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT," J. Syst. Softw., vol. 214, p. 112059, Aug. 2024, doi: 10.1016/j.jss.2024.112059.
30. [D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," 2020, arXiv. doi: 10.48550/ARXIV.2009.08366.
31. Z. Wang, C. Zhang, and M. Ren, "A Novel Benford's Law-Driven Approach for Detecting Machine-Generated Text," ACM Trans. Inf. Syst., p. 3748305, Jul. 2025, doi: 10.1145/3748305.
32. Z. Lai, X. Zhang, and S. Chen, "Adaptive Ensembles of Fine-Tuned Transformers for LLM-Generated Text Detection," in 2024 International Joint Conference on Neural Networks (IJCNN), Yokohama, Japan: IEEE, Jun. 2024, pp. 1–7. doi: 10.1109/ijcnn60899.2024.10651296.
33. K. Hayawi, S. Shahriar, and S. S. Mathew, "The imitation game: Detecting human and AI-generated texts in the era of ChatGPT and BARD," J. Inf. Sci., Feb. 2024, doi: 10.1177/01655515241227531.
34. M. S. Orenstrakh, O. Karnalim, C. A. Suárez, and M. Liut, "Detecting LLM-Generated Text in Computing Education: Comparative Study for ChatGPT Cases," in 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), Osaka, Japan: IEEE, Jul. 2024, pp. 121–126. doi: 10.1109/compsac61105.2024.00027.
35. J. Q. J. Liu et al., "The great detectives: humans versus AI detectors in catching large language model-generated medical writing," Int. J. Educ. Integr., vol. 20, no. 1, p. 8, May 2024, doi: 10.1007/s40979-024-00155-6.
36. M. A. Flitcroft et al., "Performance of Artificial Intelligence Content Detectors Using Human and Artificial Intelligence-Generated Scientific Writing," Ann. Surg. Oncol., vol. 31, no. 10, pp. 6387–6393, Oct. 2024, doi: 10.1245/s10434-024-15549-6.
37. A. Bhattacharjee and H. Liu, "Fighting Fire with Fire: Can ChatGPT Detect AI-generated Text?," ACM SIGKDD Explor. Newsl., vol. 25, no. 2, pp. 14–21, Mar. 2024, doi: 10.1145/3655103.3655106.
38. K. C. Fraser, H. Dawkins, and S. Kiritchenko, "Detecting AI-Generated Text: Factors Influencing Detectability with Current Methods," J. Artif. Intell. Res., vol. 82, pp. 2233–2278, Apr. 2025, doi: 10.1613/jair.1.16665.
39. M. Perkins, J. Roe, D. Postma, J. McGaughran, and D. Hickerson, "Detection of GPT-4 Generated Text in Higher Education: Combining Academic Judgement and Software to Identify Generative AI Tool Misuse," J. Acad. Ethics, vol. 22, no. 1, pp. 89–113, Mar. 2024, doi: 10.1007/s10805-023-09492-6.
40. R. Kumar and M. Mindzak, "Who Wrote This?: Detecting Artificial Intelligence–Generated Text from Human-Written Text," Can. Perspect. Acad. Integr., vol. 7, no. 1, Jan. 2024, doi: 10.55016/ojs/cpai.v7i1.77675.
41. J. Wu, S. Yang, R. Zhan, Y. Yuan, L. S. Chao, and D. F. Wong, "A Survey on LLM-Generated Text Detection: Necessity, Methods, and Future Directions," Comput. Linguist., vol. 51, no. 1, pp. 275–338, Mar. 2025, doi: 10.1162/coli_a_00549.
42. X. He, X. Shen, Z. Chen, M. Backes, and Y. Zhang, "MGTBench: Benchmarking Machine-Generated Text Detection," in Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, Salt Lake City UT USA: ACM, Dec. 2024, pp. 2251–2265. doi: 10.1145/3658644.3670344.
43. X. Yu et al., "EvoBench: Towards Real-world LLM-Generated Text Detection Benchmarking for Evolving Large Language Models," in Findings of the Association for Computational Linguistics: ACL 2025, Vienna, Austria: Association for Computational Linguistics, 2025, pp. 14605–14620. doi: 10.18653/v1/2025.findings-acl.754.
44. V. Bellini, F. Semeraro, J. Montomoli, M. Cascella, and E. Bignami, "Between human and AI: assessing the reliability of AI text detection tools," Curr. Med. Res. Opin., vol. 40, no. 3, pp. 353–358, Mar. 2024, doi: 10.1080/03007995.2024.2310086.
45. J. Fleckenstein, J. Meyer, T. Jansen, S. D. Keller, O. Köller, and J. Möller, "Do teachers spot AI? Evaluating the detectability of AI-generated texts among student essays," Comput. Educ. Artif. Intell., vol. 6, p. 100209, Jun. 2024, doi: 10.1016/j.caeai.2024.100209.
46. Y. Jiang, J. Hao, M. Fauss, and C. Li, "Detecting ChatGPT-generated essays in a large-scale writing assessment: Is there a bias against non-native English speakers?," Comput. Educ., vol. 217, p. 105070, Aug. 2024, doi: 10.1016/j.compedu.2024.105070.
47. B. Alhijawi, R. Jarrar, A. AbuAlRub, and A. Bader, "Deep learning detection method for large language models-generated scientific content," Neural Comput. Appl., vol. 37, no. 1, pp. 91–104, Jan. 2025, doi: 10.1007/s00521-024-10538-y.
48. A. Boutadjine, F. Harrag, and K. Shaalan, "Human vs. Machine: A Comparative Study on the Detection of AI-Generated Content," ACM Trans. Asian Low-Resour. Lang. Inf. Process., vol. 24, no. 2, pp. 1–26, Feb. 2025, doi: 10.1145/3708889.

49. V. Heilala, R. Araya, and R. Hämäläinen, "Beyond Text-to-Text: An Overview of Multimodal and Generative Artificial Intelligence for Education Using Topic Modeling," in Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, Catania International Airport Catania Italy: ACM, Mar. 2025, pp. 54–63. doi: 10.1145/3672608.3707764.
50. P. Yu, J. Chen, X. Feng, and Z. Xia, "CHEAT: A Large-Scale Dataset for Detecting CHatGPT-writtEn AbsTracts," IEEE Trans. Big Data, vol. 11, no. 3, pp. 898–906, Jun. 2025, doi: 10.1109/TBDATA.2025.3536929.
51. W. Fu, H. Wang, C. Gao, G. Liu, Y. Li, and T. Jiang, "MIA-Tuner: Adapting Large Language Models as Pre-training Text Detector," Proc. AAAI Conf. Artif. Intell., vol. 39, no. 26, pp. 27295–27303, Apr. 2025, doi: 10.1609/aaai.v39i26.34939.
52. A. Kartelj, M. Mladenović, and S. Vujičić Stanković, "Comparison of algorithms for the recognition of ChatGPT paraphrased texts," J. Big Data, vol. 12, no. 1, p. 28, Feb. 2025, doi: 10.1186/s40537-025-01082-0.
53. J. Sun and Z. Lv, "Zero-shot detection of LLM-generated text via text reorder," Neurocomputing, vol. 631, p. 129829, May 2025, doi: 10.1016/j.neucom.2025.129829.
54. D. Mao, D. Zhang, A. Zhang, and Z. Zhao, "MLSDET: Multi-LLM Statistical Deep Ensemble for Chinese AI-Generated Text Detection," in ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Hyderabad, India: IEEE, Apr. 2025, pp. 1–5. doi: 10.1109/ICASSP49660.2025.10888686.
55. J. Blake, A. S. M. Miah, K. Kredens, and J. Shin, "Detection of AI-Generated Texts: A Bi-LSTM and Attention-Based Approach," IEEE Access, vol. 13, pp. 71563–71576, 2025, doi: 10.1109/ACCESS.2025.3562750.
56. S. Pordanesh, S. Bukhari, B. Tan, and L. De Carli, "Hiding in Plain Sight: On the Robustness of AI-Generated Code Detection," in Detection of Intrusions and Malware, and Vulnerability Assessment, vol. 15748, M. Egele, V. Moonsamy, D. Gruss, and M. Carminati, Eds., in Lecture Notes in Computer Science, vol. 15748. , Cham: Springer Nature Switzerland, 2025, pp. 44–64. doi: 10.1007/978-3-031-97623-0_3.
57. J. Esteban Cuellar Argotty and R. Manrique, "AI-Generated Code Detection: An Examination of Current Tools in Education," in Generative Systems and Intelligent Tutoring Systems, vol. 15723, S. Graf and A. Markos, Eds., in Lecture Notes in Computer Science, vol. 15723. , Cham: Springer Nature Switzerland, 2026, pp. 192–201. doi: 10.1007/978-3-031-98281-1_15.
58. W. H. Pan et al., "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, Lisbon Portugal: ACM, Apr. 2024, pp. 1–11. doi: 10.1145/3639474.3640068.
59. M. Hoq et al., "Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models," in Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, Portland OR USA: ACM, Mar. 2024, pp. 526–532. doi: 10.1145/3626252.3630826.
60. L. Huynh et al., "Detecting Code Vulnerabilities using LLMs," in 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Naples, Italy: IEEE, Jun. 2025, pp. 401–414. doi: 10.1109/DSN64029.2025.00047.
61. A. Mechri, M. A. Ferrag, and M. Debbah, "SecureQwen: Leveraging LLMs for vulnerability detection in python codebases," Comput. Secur., vol. 148, p. 104151, Jan. 2025, doi: 10.1016/j.cose.2024.104151.
62. D. Cotroneo, R. De Luca, and P. Liguori, "DeVAIC: A tool for security assessment of AI-generated code," Inf. Softw. Technol., vol. 177, p. 107572, Jan. 2025, doi: 10.1016/j.infsof.2024.107572.
63. M. F. Rabbi, A. I. Champa, M. F. Zibran, and M. R. Islam, "AI Writes, We Analyze: The ChatGPT Python Code Saga," in Proceedings of the 21st International Conference on Mining Software Repositories, Lisbon Portugal: ACM, Apr. 2024, pp. 177–181. doi: 10.1145/3643991.3645076.
64. L. Huang, Z. Zhang, Y. Zhang, X. Zhou, and S. Wang, "RU-AI: A Large Multimodal Dataset for Machine-Generated Content Detection," in Companion Proceedings of the ACM on Web Conference 2025, Sydney NSW Australia: ACM, May 2025, pp. 733–736. doi: 10.1145/3701716.3715306.
65. B. Demirok and M. Kutlu, "AIGCodeSet: A New Annotated Dataset for AI Generated Code Detection," in 2025 33rd Signal Processing and Communications Applications Conference (SIU), Sile, Istanbul, Turkiye: IEEE, Jun. 2025, pp. 1–4. doi: 10.1109/SIU66497.2025.11112334.
66. T. Ye et al., "Uncovering LLM-Generated Code: A Zero-Shot Synthetic Code Detector via Code Rewriting," Proc. AAAI Conf. Artif. Intell., vol. 39, no. 1, pp. 968–976, Apr. 2025, doi: 10.1609/aaai.v39i1.32082.
67. M. Oedingen, R. C. Engelhardt, R. Denz, M. Hammer, and W. Konen, "ChatGPT Code Detection: Techniques for Uncovering the Source of Code," AI, vol. 5, no. 3, pp. 1066–1094, Jul. 2024, doi: 10.3390/ai5030053.
68. Z. Xu and V. S. Sheng, "Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models," Proc. AAAI Conf. Artif. Intell., vol. 38, no. 21, pp. 23155–23162, Mar. 2024, doi: 10.1609/aaai.v38i21.30361.
69. N. Adham, H. Duwe, and H. H. Hoos, "Codetector: A Framework for Zero-shot Detection of AI-Generated Code," in Learning and Intelligent Optimization (LION 2025), Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2025, doi: 10.1007/978-3-032-09192-5_11.
70. Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, Online: Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.
71. X. Yang, K. Zhang, H. Chen, L. Petzold, W. Y. Wang, and W. Cheng, "Zero-Shot Detection of Machine-Generated Codes," arXiv, 2023, doi: 10.48550/arXiv.2310.05103.

72. H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu, and I. Ahmed, "An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?," in Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025, arXiv:2411.04299.
73. X. Xu et al., "Distinguishing LLM-generated from Human-written Code by Contrastive Learning," arXiv:2411.04704, 2024.
74. T. G. Dietterich, "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," Neural Comput., vol. 10, no. 7, pp. 1895–1923, Oct. 1998, doi: 10.1162/089976698300017197.
75. P. Micikevicius et al., "Mixed Precision Training," in International Conference on Learning Representations (ICLR), 2018, arXiv:1710.03740.