AICodeDetect: A Pipeline for Systematic Detection and Analysis of AI-Generated Code

1 Abstract

The surge in AI code generation tools has created an urgent need for standardized evaluation frameworks to assess detection methods. We introduce AICodeDetect, a comprehensive evaluation pipeline for AI-generated code detection that spans multiple programming paradigms and AI models. Our framework establishes testing protocols across imperative (Java, Python) and functional (OCaml) languages, incorporating code samples from GPT-3.5Turbo and GPT-4o to enable systematic comparison of detection approaches. The pipeline implements a progressive series of architectures: foundational methods (KNN, Neural Networks), sequence models (LSTM), traditional ML (Decision Trees, SVM), CNN-based approaches, transformer architectures (CodeBERT), and advanced multimodal fusion techniques. Through systematic ablation studies, we uncover key insights: (1) more sophisticated models like GPT-4o produce more detectable patterns than GPT-3.5Turbo, suggesting distinctive AI signatures rather than improved human mimicry, (2) detection efficacy varies across programming paradigms, with robust results in statically-typed languages, and (3) our novel CodeFusion architecture, which uses dual-stream processing contrastive learning with Vision Transformers, achieves perfect detection (F1 = 1.00) for GPT-4o code across all languages. The framework is designed to be extensible, enabling continuous evaluation as new models and detection methods emerge.

2 Introduction

The rise of AI-driven code generation models, such as OpenAI's ChatGPT and Codex, has fundamentally transformed software development while creating unprecedented challenges in education, software engineering, and cybersecurity. As these systems generate increasingly sophisticated code across multiple programming languages, the need for robust detection methods has become critical.

Despite the growing importance of AI code detection, the field lacks standardized evaluation methodologies that span different programming paradigms and AI model generations. Current detection approaches vary widely, from traditional machine learning methods treating code as token sequences to transformer-based models analyzing abstract syntax trees. However, without systematic evaluation frameworks, it remains unclear which approaches are most effective across different languages and scenarios.

To address this gap, we present AICodeDetect, a comprehensive pipeline for evaluating AI-generated code detection methods. Our framework provides testing protocols across both imperative (Python, Java) and functional (OCaml) programming paradigms, incorporating code samples from different language

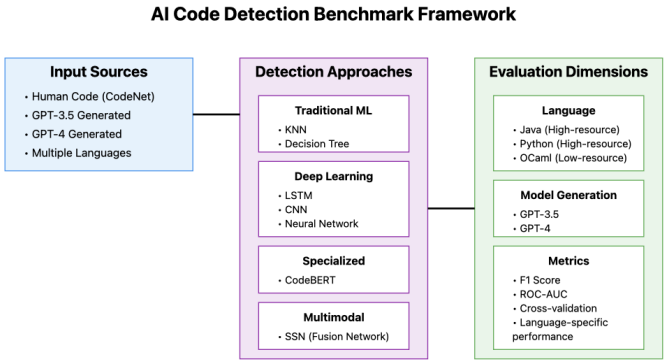


Fig. 1: Overview of AICodeDetect Pipeline.

model generations (GPT-3.5Turbo, GPT-4o). This modular pipeline enables systematic evaluation of various detection approaches and provides insights into their relative strengths and limitations.

The pipeline includes multiple evaluation stages that serve as an ablation study: First, it establishes results using traditional methods (CNN, SVM) to process raw code features. Next, it evaluates deep learning and transformer-based models on the same samples. Finally, it implements our novel CodeFusion approach, which combines CNN-based code analysis with Vision Transformer (ViT) processing of code's visual features. This systematic progression from basic models to our full CodeFusion architecture allows us to assess the contribution of each component to detection performance.

Through systematic application of our pipeline, we find that GPT-4o generated code is consistently more detectable than GPT-3.5Turbo code across all architectural configurations. More importantly, the ablation results reveal that while the CNN+SVM method achieves strong performance, the full CodeFusion architecture significantly improves detection accuracy, achieving perfect detection (F1 = 1.00) for GPT-4o generated code (Fig 6c). This progressive improvement demonstrates the value of integrating visual features into the detection pipeline.

Our main contributions include: (1) A modular pipeline architecture for evaluating AI code detection methods across multiple programming languages and AI model generations, (2) A systematic ablation study comparing detection approaches, from traditional ML methods to our novel CodeFusion architecture, (3) Empirical insights into detection effectiveness across programming paradigms and model generations, and (4) Integration of multimodal detection through CodeFusion, demonstrating how combining visual and textual features enhances detection performance.

3 Background

Prior work has explored various approaches to detecting AI-generated code. [7] conducted a comprehensive evaluation of existing AI code detectors against 5,069 Python solutions from Kaggle and LeetCode, testing 13 different problem variants with ChatGPT. Their study revealed limitations in current detection methods. Code stylometry has emerged as a promising detection approach. [4] achieved strong results distinguishing between GPT-4 and human-authored CodeChef solutions, with their classifier reaching an F1-score and AUC-ROC of 0.91. Their performance remained robust (0.89) even when excluding easily manipulated features like whitespace patterns, demonstrating consistent detection across problem difficulty levels. Similarly, [8] explored both lexical features and syntactic patterns from Abstract Syntax Trees on the NYU Lost-at-C dataset. Using standard classifiers on 58 C source files, they achieved accuracy up to 92% in distinguishing AI from human-generated code. Their study focused on a controlled programming assignment where both human developers and AI models implemented a shopping list using linked lists in C, providing a standardized basis for comparison. [5] and [3] have also contributed detection methods for text, rather than code, further expanding the field's understanding of distinguishing features between AI and human-written artifacts. [1] demonstrated that converting binary files to grayscale images enabled highly accurate malware classification through visual signatures. Their innovative approach of treating code as images achieved 98% accuracy in malware family classification, suggesting that visual representations can capture subtle patterns that traditional feature extraction might miss. We draw inspiration from this work in our multimodal methodology.

4 Methods

Our detection pipeline consists of three major components: (1) a multi-language data processing module, (2) a visual representation generator, and (3) a suite of detection methods that enable systematic ablation studies. Each component is designed to be modular and extensible for future enhancements.

4.1 Multi-Language Data Processing Module

This module implements three core design principles: language diversity, balanced representation, and real-world applicability. It processes code samples across Python, Java, and OCaml, deliberately incorporating both high-resource and low-resource programming languages to evaluate detection across varying syntax structures and ecosystem maturity levels.

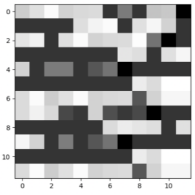
For each programming challenge from CodeNet [10], the module maintains a balanced composition: five human-written solutions from original repositories serve as authentic programming methods, complemented by ten AI-generated solutions (five each from GPT-3.5 Turbo and GPT-4). This distribution enables

comprehensive analysis across different generation capabilities while maintaining statistical validity.

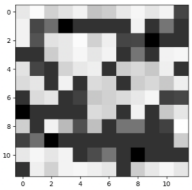
To ensure reproducibility, we implemented a standardized protocol using language-specific prompt templates for AI-generated samples. Our final dataset comprises approximately 8,000 samples per language. This three-way comparison enables thorough evaluation of detection methods across multiple dimensions of interest, from language-specific performance to cross-generation transfer capabilities. Our dataset analysis considers metrics such as code density, comment ratio, whitespace ratio, and line length distribution, as depicted in Figures 4a, 4b, 4c, and 4d.

4.2 Visual Representation Generation

We include visual representations in our benchmark dataset to capture structural patterns in code organization that may be independent of programming language or generation model. We transform source code into fixed-dimension visual representations through binary encoding: code files are tokenized into binary sequences, mapped to 34×34 dimensional matrices while preserving sequential order, and zero-padded for uniform dimensionality. The resulting matrices are rendered as grayscale images, where each cell's binary value determines its intensity (2).



(a) Human-written code as grayscale.



(b) AI-generated code as grayscale.

Fig. 2: Grayscale image representations of human-written and AI-generated code.

4.3 Detection Methods Module

Our detection module implements a systematic evaluation of multiple approaches for identifying AI-generated code, serving both as an ablation study and a comprehensive comparison of detection techniques. Each method is evaluated independently while supporting integration into an end-to-end detection pipeline.

Baseline Methods We establish baseline performance using classical machine learning approaches optimized for code analysis. A **K-Nearest Neighbors**

(KNN) classifier evaluates similarity between code samples using TF-IDF vectorization, capturing both lexical patterns and keyword distributions characteristic of human and AI-generated code. A **Decision Tree** learns hierarchical rules by recursively splitting on the most discriminative code features, providing an interpretable model that reveals key differences between human and AI code patterns.

Deep Learning Module Our deep learning approaches capture increasingly complex code representations. A feed-forward **Neural Network** learns nonlinear feature combinations from the code's lexical structure. The **CNN** architecture processes code as image data through three convolutional layers with batch normalization, enabling detection of visual patterns in code formatting and structure that may distinguish AI generation. A **LSTM** network with BERT embeddings models the sequential nature of code, capturing long-range dependencies and contextual relationships between code elements. We also adapt **CodeBERT**, which pre-trains BERT's architecture specifically for code understanding tasks, leveraging its knowledge of programming language syntax and semantics for detection.

Fusion Module (CodeFusion) We develop fusion approaches that combine textual and visual code representations to capture complementary features. The **TF-IDF CNN** merges statistical text patterns with visual code structure by combining TF-IDF features with CNN representations, enabling detection of inconsistencies between code content and formatting. A **CNN-SVM** hybrid leverages CNNs for automatic feature extraction from code images, feeding these learned representations into an SVM for robust classification. Our primary contribution, **CodeFusion**, processes code through parallel visual and textual pathways with contrastive learning to capture both structural and semantic patterns:

Visual Pipeline: Processes input code images $x \in \mathbb{R}^{H \times W \times C}$ through patch-based transformation:

Given an input code image $x \in \mathbb{R}^{H \times W \times C}$, we divide it into fixed-size patches $x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the image resolution and $N = HW/P^2$ is the number of patches:

$$z_0 = [x_{class}; x_p^1 E; x_p^2 E; ...; x_p^N E] + E_{pos} \tag{1}$$

where E is the patch embedding projection and E_{pos} are learnable position embeddings.

Textual Pipeline: The BERT-based pipeline processes tokenized input sequences $t = [t_1, ..., t_n]$ to generate contextual embeddings:

$$h = \text{BERT}(t) \tag{2}$$

where $h \in \mathbb{R}^{n \times d}$ and d is the embedding dimension.

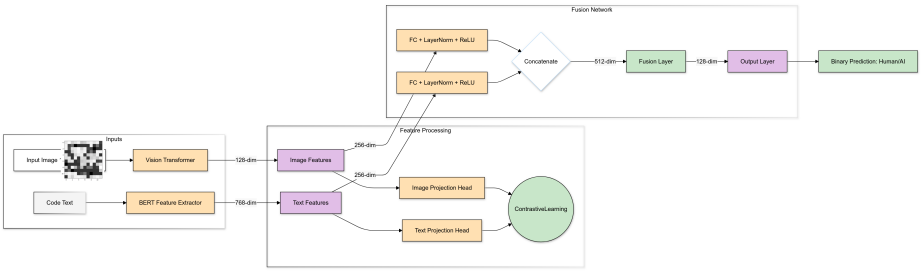


Fig. 3: Overview of CodeFusion model, showing the visual and textual processing streams and their fusion through contrastive learning.

Contrastive Fusion We project both modalities into a shared space:

$$p_v = f_v(v), \quad p_t = f_t(h_{[CLS]}) \tag{3}$$

The model optimizes a combined objective, $\mathcal{L} = \mathcal{L}_{BCE} + \lambda \mathcal{L}_{cont}$

where $\mathcal{L}_{BCE}$ is the binary cross-entropy loss and $\mathcal{L}_{cont}$ is the contrastive loss computed as:

$$\mathcal{L}_{cont} = -\log \frac{\exp(sim(p_v, p_t)/\tau)}{\sum_{i=1}^{N} \exp(sim(p_v, p_t^i)/\tau)} \tag{4}$$

5 Pipeline Evaluation and Analysis

We evaluate our AICodeDetect pipeline through three progressive stages: structural analysis, visual representation assessment, and comprehensive detection performance evaluation. Through this systematic evaluation, we demonstrate how each component of the pipeline contributes to detection performance while revealing key insights about AI-generated code patterns.

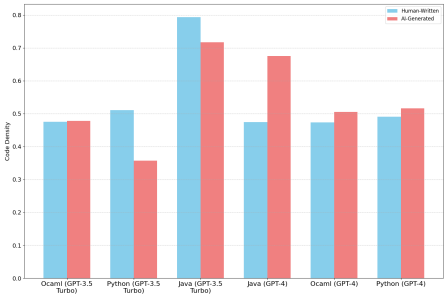
5.1 Stage 1: Structural Analysis Module

Our pipeline's structural analysis component examines four key metrics that reveal distinctive patterns between human and AI-generated code. The code density analysis (Figure 4a) reveals model-specific signatures across languages. In Java, GPT-3.5Turbo produces lower density code compared to human-written code, while Python shows higher density in AI-generated samples. GPT-4o generated code shows more consistent density levels across languages, though these patterns differ significantly from human baselines.

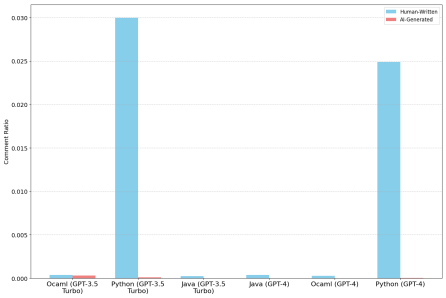
Comment distribution analysis (Figure 4b) provides the strongest discriminative signal in the pipeline, with human Python code exhibiting substantially higher comment density, while AI-generated code maintains consistently low comment ratios across all languages and models. The whitespace pattern analysis (Figure 4c) further identifies model-specific formatting characteristics, where

Java code from GPT-4o exhibits higher whitespace ratios than human code, while GPT-3.5Turbo-generated code shows more similar whitespace patterns to human samples across languages. This suggests that newer models may develop distinct formatting preferences rather than simply mimicking human patterns.

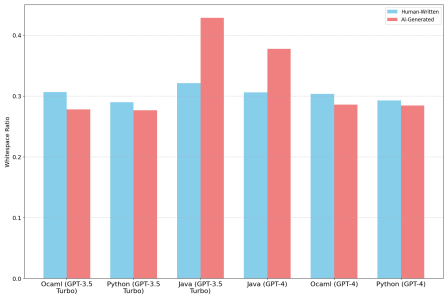
Line length analysis (Figure 4d) reveals that human-written OCaml code consistently has longer lines than AI-generated versions, with less pronounced differences observed in Python and Java. These structural metrics provide fundamental signals that inform the pipeline's detection mechanisms.



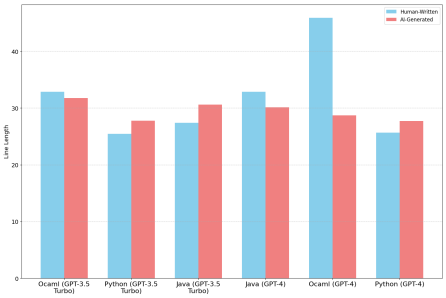
(a) Code density across all samples.



(b) Comment ratio across all samples.



(c) Whitespace ratio across all samples.



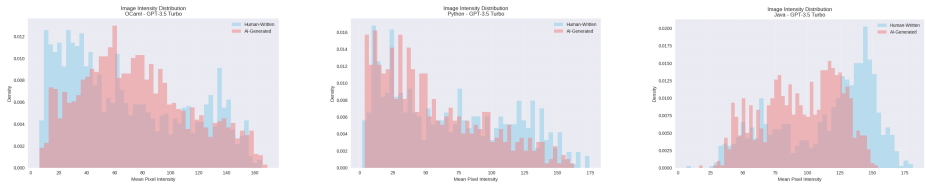
(d) Line length distribution.

Fig. 4: Analysis of code structure metrics across all samples. Human-written Python samples show the highest comment density, reflecting Python's emphasis on readability and documentation.

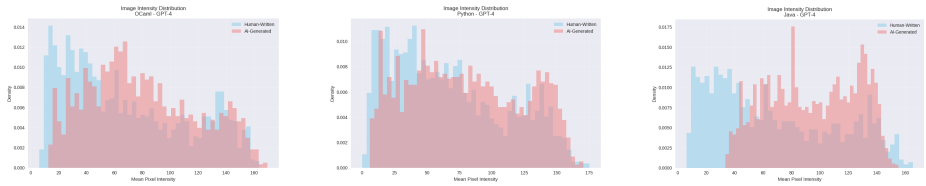
5.2 Stage 2: Visual Analysis Module

The pipeline's visual analysis component processes code into standardized image representations, enabling detection of spatial and structural patterns that may not be apparent in textual analysis alone. Image intensity (5) distributions reveal clear separation between human and GPT-4o code in Java, while OCaml shows more similar patterns between human and AI code, potentially due to strict formatting conventions. Python demonstrates intermediate differentiation, with GPT-4o showing more distinctive patterns than GPT-3.5Turbo. These vi-

sual signatures provide complementary signals that enhance the pipeline's fusion modules.



(a) OCaml GPT-3.5Turbo (b) Python GPT-3.5Turbo (c) Java GPT-3.5Turbo



(d) OCaml GPT-4o (e) Python GPT-4o (f) Java GPT-4o

Fig. 5: Image intensity analysis across languages and models.

5.3 Stage 3: Detection Performance Evaluation

Cross-Model Performance Analysis Our evaluation framework reveals progressive improvements across architectural complexity. Traditional methods like CNN and SVM show notable improvement when detecting GPT-4o code compared to GPT-3.5Turbo, with performance gains of up to 15% in Python. The baseline CNN achieves F1 scores of 0.92, 0.79, and 0.79 for Java, Python, and OCaml respectively on GPT-4o code.

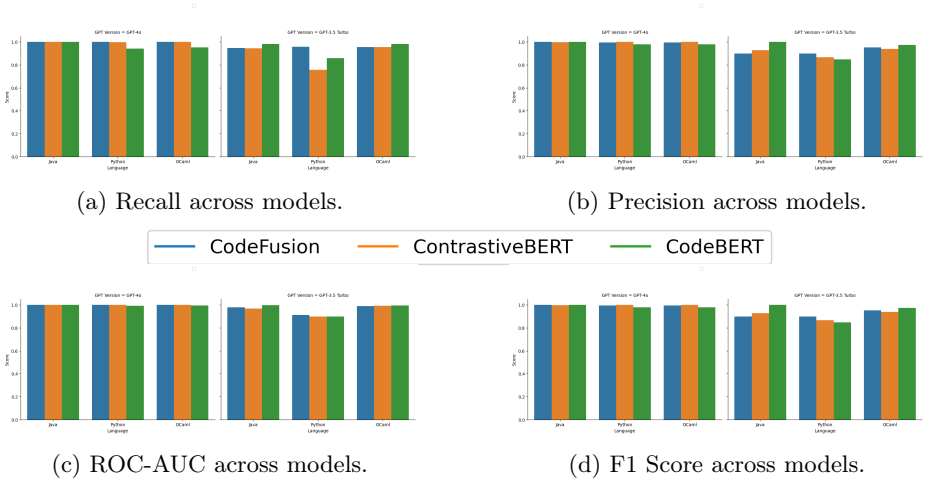
CodeBERT demonstrates exceptional capability with particularly high accuracy on GPT-4o generated code, achieving perfect detection in Java ($F1 = 1.00$) and near-perfect results in Python ($F1 = 0.99$) and OCaml ($F1 = 0.98$). On GPT-3.5Turbo code, CodeBERT maintains strong performance with F1 scores of 0.98, 0.85, and 0.96 for Java, Python, and OCaml respectively.

Our novel CodeFusion architecture extends these capabilities further, achieving perfect or near-perfect detection ($F1 \geq 0.99$) for GPT-4o code across all languages, while maintaining competitive performance on GPT-3.5Turbo code (F1 scores: Java 0.92, Python 0.81, OCaml 0.95). The ROC-AUC scores for CodeFusion reach 1.00 across all languages for GPT-4o detection.

Language-Specific Analysis The pipeline reveals varying effectiveness across programming languages. In Java, all advanced methods (CodeBERT and CodeFusion) achieve strong performance ($F1 \geq 0.95$), suggesting Java's structured

nature provides clear detection signals. Python's flexibility presents unique challenges, particularly for GPT-3.5Turbo code, though CodeFusion's multimodal architecture demonstrates particular strength here, achieving near-perfect detection for GPT-4o code.

Results in OCaml highlight the importance of capturing both semantic and structural patterns in functional programming languages. CodeFusion's perfect performance on GPT-4 code ($F1 = 1.00$) compared to CodeBERT's 0.978 validates the multimodal approach. Traditional methods maintain reasonable performance, with SVM achieving F1 scores above 0.90 across all languages for GPT-4o detection.



(a) Recall across models.

(b) Precision across models.

(c) ROC-AUC across models.

(d) F1 Score across models.

Fig. 6: Comparison of model performance metrics. Left: 4o, Right: 3.5Turbo

5.4 Model Evolution Impact Analysis

Our pipeline reveals an unexpected trend in AI code generation evolution: despite GPT-4o's more advanced capabilities, its code is consistently more detectable than GPT-3.5Turbo's across all languages and detection methods. Traditional approaches show marked improvement in detecting GPT-4o code, with performance gains of 10-15% compared to GPT-3.5Turbo detection. This pattern suggests that as language models become more sophisticated, they may develop their own distinctive coding patterns rather than simply mimicking human conventions.

The superior performance of CodeFusion on GPT-4o generated code (achieving $F1 \geq 0.99$ across all languages) suggests that multimodal analysis becomes particularly valuable for newer generation models. This may be because these models produce more distinctive structural and visual patterns, as evidenced

by our structural analysis module's findings on code density and whitespace distribution.

Quantitative analysis through our pipeline components reveals specific patterns in model evolution:

- Traditional methods (CNN, SVM) show consistent improvement in GPT-4o detection compared to GPT-3.5Turbo across all languages (average improvement: Java 13%, Python 15%, OCaml 12%)
- Transformer-based methods achieve near-perfect detection rates on GPT-4o code while showing more variation on GPT-3.5Turbo samples (CodeBERT F1 difference: Java 0.02, Python 0.14, OCaml 0.02)
- Multimodal fusion approaches demonstrate the most robust performance across both model generations, with CodeFusion maintaining F1 scores above 0.90 for most language-model combinations

These findings highlight the importance of evolving detection strategies alongside generation capabilities, potentially focusing more on characteristic patterns that emerge from advanced models rather than deviations from human conventions. The pipeline's modular design enables continuous adaptation to these evolving patterns while maintaining systematic evaluation capabilities.

Model	Java 4o		Python 4o		OCaml 4o	
	F1	ROC	F1	ROC	F1	ROC
CodeFusion	1.00	1.00	1.00	1.00	0.98	1.00
CodeBERT	1.00	1.00	0.99	0.99	0.98	0.99
Contrastive BERT CNN	1.00	1.00	1.00	1.00	0.98	1.00
TFID-CNN	1.00	1.00	0.91	0.99	0.97	1.00
SVM	0.96	0.99	0.93	0.98	0.96	0.99
LSTM	0.94	0.97	0.92	0.98	0.97	0.99
Decision Tree	0.94	0.94	0.91	0.92	0.94	0.93
Neural Network	0.83	0.90	0.89	0.89	0.84	0.86
KNN	0.83	0.81	0.84	0.91	0.85	0.92
CNN	0.92	0.98	0.79	0.88	0.79	0.89
CNN SVM	0.86	0.92	0.75	0.83	0.76	0.82

Table 1: Model 4o Results

6 Discussion

Our pipeline evaluation demonstrates immediate practical applications across software development contexts, from automated code review systems to educational institutions assessing student submissions. The modular nature of our framework enables adaptation to language-specific characteristics while maintaining consistent evaluation protocols.

The structural analysis module reveals key insights into why GPT-4o code is more consistently detectable than GPT-3.5Turbo across our pipeline stages. While GPT-3.5Turbo exhibits variable patterns, GPT-4o shows pronounced and

Model	Java 3.5		Python 3.5		OCaml 3.5	
	F1	ROC	F1	ROC	F1	ROC
CodeFusion	0.92	0.98	0.81	0.91	0.95	0.99
CodeBERT	0.98	1.00	0.85	0.94	0.96	0.99
Contrastive BERT CNN	0.91	0.97	0.81	0.90	0.94	0.99
TFID-CNN	0.90	0.97	0.81	0.89	0.90	0.96
SVM	0.93	0.98	0.84	0.93	0.91	0.96
LSTM	0.90	0.96	0.81	0.90	0.92	0.97
Decision Tree	0.90	0.90	0.76	0.77	0.86	0.86
Neural Network	0.70	0.81	0.81	0.80	0.91	0.92
KNN	0.83	0.90	0.67	0.85	0.85	0.91
CNN	0.92	0.98	0.66	0.75	0.76	0.82
CNN SVM	0.88	0.94	0.66	0.72	0.72	0.80

Table 2: Model 3.5 Turbo Results

consistent deviations from human code: near-zero comment ratios across all languages (Figure 4b), distinctive whitespace distributions particularly in Java (Figure 4c), and consistent line length patterns. These systematic differences suggest GPT-4o has developed its own consistent coding style rather than mimicking human patterns.

6.1 Architectural Performance Analysis

Our pipeline's progressive evaluation of detection architectures reveals distinct performance patterns. Traditional methods establish baseline performance levels, while transformer-based approaches like CodeBERT achieve substantially stronger results, particularly on GPT-4o code. The pipeline demonstrates that static languages like Java consistently yield higher detection rates ($F1 > 0.97$) compared to dynamic languages like Python ($F1 \approx 0.85$).

The CodeFusion architecture (Fig 3) leverages contrastive learning to align visual code structure with semantic understanding, helping identify discrepancies that often characterize AI-generated code. The dual-stream approach preserves modality-specific features while ensuring their complementary signals contribute to the final detection decision.

The complete CodeFusion architecture, which integrates all pipeline components including structural and visual features, achieves near-perfect detection ($F1 > 0.99$) of GPT-4o code across all languages. This progression in performance through our pipeline stages demonstrates the value of combining multiple analysis modalities.

6.2 Model Evolution Insights

A key finding from our pipeline evaluation is that GPT-4o's output is more consistently detectable than GPT-3.5's across all languages and detection methods, with the gap particularly pronounced in Python (F1 improvement > 0.15). This

reveals an unexpected trend in language model evolution: as models become more sophisticated, they may optimize for internal consistency and efficiency rather than human mimicry.

Current language models are initially trained to imitate human code through techniques like RLHF, but our pipeline analysis suggests they may optimize beyond simple mimicry as they advance. This evolution toward distinctive AI patterns, rather than closer human imitation, could reflect these systems developing their own efficient approaches through reinforcement learning over long trajectories. Such a trend has significant implications for detection strategies, as future detection systems may need to focus on identifying model-specific patterns rather than deviations from human code. It also improves our understanding of AI capabilities: models may develop novel coding approaches that prioritize efficiency over human-like characteristics. It also lays the foundation for future frameworks, which must maintain flexibility to adapt to evolving AI coding patterns.

References

1. L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in Proceedings of the 8th international symposium on visualization for cyber security, 2011, pp. 1–7.
2. N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 2785–2799.
3. A. Bhattacharjee and H. Liu, "Fighting fire with fire: can ChatGPT detect AI-generated text?" ACM SIGKDD Explorations Newsletter, vol. 25, no. 2, pp. 14–21, 2024.
4. O. J. Idialu, N. S. Mathews, R. Maipradit, J. M. Atlee, and M. Nagappan, "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, 2024, pp. 394–406.
5. V. S. Sadasivan, A. Kumar, S. Balasubramanian, W. Wang, and S. Feizi, "Can AI-generated text be reliably detected?" arXiv preprint arXiv:2303.11156, 2023.
6. Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot generated code in github," arXiv preprint arXiv:2310.02059, 2023.
7. W. H. Pan, M. J. Chok, J. L. S. Wong, Y. X. Shin, Y. S. Poon, Z. Yang, C. Y. Chong, D. Lo, and M. K. Lim, "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, 2024, pp. 1–11.
8. S. Bukhari, B. Tan, and L. De Carli, "Distinguishing AI-and Human-Generated Code: a Case Study," in Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, 2023, pp. 17–25.
9. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
10. Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., et al. (2021). Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.