Can We Trust LLM-Generated Code? – A Quantitative Verification Study

E. Michael Bearss
Trideum Corporation
Huntsville, AL
mbearss@trideum.com

ABSTRACT

Recent advances in generative AI have positioned Large Language Models (LLMs) as potent tools for a variety of tasks, yet their widespread adoption raises important questions about reliability, quality, and trustworthiness. Building on previous work presented at I/ITSEC in 2024, which introduced verification and validation methods for code-generating LLMs, this paper significantly expands the dataset and the evaluation techniques. The limited sample size of the previous study has been expanded to over 10,000 code samples, including human-written code from public GitHub repositories and outputs from three distinct LLMs solving a suite of interview challenges.

This expanded dataset enables a more rigorous quantitative assessment of these models. Established code quality metrics, such as cyclomatic complexity, Halstead complexity, and maintainability index, are used to compare and contrast human-generated code with their LLM-generated counterparts. Additionally, an analysis of each program's abstract syntax tree was performed to uncover structural patterns uniquely characterizing LLM-generated code. Multiple machine learning models were also trained to distinguish LLM-generated code from human-written code, with a primary focus on explainability. By highlighting the most salient features guiding classification decisions, these models provide deeper insight into the elements that differentiate LLM-generated programs from those authored by humans.

This paper presents the results of this analysis and illustrates how LLM-generated code differs from human-written code, both in structural complexity and in measurable software engineering attributes. In addition to offering deeper insight into the reliability and clarity of LLM-generated code, these findings have broader implications for automated code review, compliance audits, and best practices in AI-assisted programming workflows. By systematically identifying and quantifying the strengths and weaknesses of LLM-generated code, this work enhances the understanding of how to effectively integrate AI into modern software development while preserving critical standards of quality, security, and maintainability.

ABOUT THE AUTHORS

E. Michael Bearss is a data scientist at Trideum Corporation, working at the Redstone Test Center in Huntsville, AL. He is a Certified Modeling and Simulation Professional and has multiple years of experience in both AI/ML and Live/Virtual/Constructive simulation. He received a Ph.D. in Computer Science from the University of Alabama in Huntsville in 2023.

Can We Trust LLM-Generated Code? – A Quantitative Verification Study

E. Michael Bearss
Trideum Corporation
Huntsville, AL
mbearss@trideum.com

INTRODUCTION

Large Language Models (LLMs) have become a popular tool for automated code generation. Their ability to translate natural language specifications into working source code promises to accelerate software development in numerous domains (Ashrafi, Bouktif, & Mediani, 2025). This trend is evident in the development of simulation and training systems, where engineers are beginning to leverage LLMs to quickly prototype scenarios, generate routine code modules, and assist in complex system integrations. However, with the growing adoption of LLM-generated code comes a fundamental concern: **can we trust the code produced by these models in operationally critical contexts?** In practice, LLM-generated code often appears syntactically correct and plausible, yet may conceal subtle bugs or security flaws. These hidden bugs, along with the rapid adoption of LLMs create an urgent need to establish confidence that LLM-produced code will behave reliably and safely when deployed in mission-critical training applications.

In 2024, a qualitative assessment was conducted to compare LLM-generated code to human generated code when solving coding tasks. The paper *"Evaluating the Trustworthiness of Large Language Models for Code Generation*," presented at I/ITSEC 2024, determined that for sufficiently complex examples, human-generated code is typically more correct and efficient. In addition, humans typically prefer code written by humans to code written by LLMs. However, LLMs are rapidly advancing, closing the gap significantly. In that paper, the more advanced LLM (ChatGPT 4) demonstrated greater coding ability, and generated code samples that were more understandable than older models like ChatGPT 3.5 (Bearss, 2024).

The previous study focused on the evaluation of LLMs on coding tasks using several methods including hidden unit tests, expert review (face validation), and evaluating samples using an ML classifier to determine the origin of the samples. The findings confirmed that while LLMs can produce functionally correct solutions in many cases, they also exhibit failures when confronted with novel topics. In the simpler problem (Fibonnacci Primes), ChatGPT 4 was able to generate code rivaling the best human expert, however, with the more complex problem (Collatz Numbers), both LLMs failed to produce a working solution. While the study produced some insights, the scope was limited. The previous study used a sample size of two problems, two LLMs, and eight human coders. In addition, the face validation used qualitative metrics to evaluate the two code provenances. These limitations left open questions about generalizability and failed to provide a detailed quantitative basis for trusting AI-generated code.

Building on that foundation, this paper advances the study of LLM-generated code verification with greater scale and rigor. In particular, a quantitative verification study employing a substantially larger dataset and more comprehensive analytics is presented. Specifically, this work analyzes over *10,000* samples of code generated by state-of-the-art LLMs, enabling more robust statistical insights into model performance. In addition, rather than focusing on correctness, this study instead focuses on structural and software engineering quality metrics (e.g., complexity, maintainability, and static analysis checks) to better quantify attributes related to trustworthiness. Furthermore, an explainable machine learning classifier to automatically distinguish between LLM-generated and human-generated code samples was developed. The classifier was trained on a combination of software engineering metrics, Abstract Syntax Tree (AST)-derived structural features, and code embeddings, enabling code samples to be accurately distinguished based on both their structural complexity and semantic content. Human-interpretable explanations were generated using SHAP (Shapley Additive Explanations) analysis, allowing the most influential factors (e.g., unique subtree count and parse tree depth) to be identified. Through this approach, a more rigorous and structure-focused analysis of LLM-generated code was achieved, emphasizing how syntactic patterns and structural diversity contribute to the detection and evaluation of code trustworthiness.

Trust in software is a prerequisite for deploying automated solutions in high-stakes training and operational environments. This work provides a scalable, quantitative framework for evaluating the structural and functional reliability of LLM-generated code by directly comparing it to expert-authored human code. Through a combination of software engineering metrics and AST analysis, consistent and measurable differences have been identified, enabling robust classification of code origin and revealing characteristic patterns unique to LLMs. These findings offer timely guidance for organizations seeking to integrate AI coding assistants into the development of training simulations. By better understanding how LLM-generated code is different, development teams can make more informed decisions about review, integration, and oversight, ensuring more trustworthy systems.

BACKGROUND

This section provides background information on several key topics relevant to the analysis presented in this paper. It begins with a brief overview of LLMs in the context of code generation, followed by a detailed description of established software engineering metrics used to evaluate code quality. In addition, the role of AST analysis is introduced, highlighting its value in uncovering structural patterns within source code that may differentiate human-written and LLM-generated programs.

LLMs and Code Generation

LLMs are deep neural networks with billions (or even trillions) of parameters, typically built on the transformer architecture. These models are trained on massive text corpora (including source code) and learn to predict text sequences, enabling them to generate coherent natural language and code. Notably, decoder-only transformer LLMs like ChatGPT and its successors have been fine-tuned for programming tasks. For example, OpenAI's Codex (a GPT-3 derivative trained on GitHub code) can translate natural language prompts into executable code and powers tools such as GitHub Copilot (Chen et al., 2021). LLM-based code generators have demonstrated near-expert proficiency on certain benchmarks (Xu & Sheng, 2025), solving complex programming problems that earlier models could not. This performance makes LLMs highly applicable to software development, as they can rapidly produce boilerplate code, suggest algorithms, and even complete functions from comments or docstrings. However, their outputs are not guaranteed to be correct or optimal. Several studies have shown that even advanced models only solve a fraction of coding tasks without errors and may require multiple attempts or user guidance to produce a valid solution (Chen et al., 2021). Moreover, LLMs operate as black boxes and have no built-in verification; thus the code they generate cannot be inherently trusted to be correct or error-free (Cramer & McIntyre, 2025). Prior work has noted that LLM-generated code, while often syntactically correct, may contain logical bugs, inefficiencies, or security vulnerabilities that a human programmer might catch (Ma et al., 2024). While LLMs offer powerful new capabilities for automatic code generation, their architecture and training bring open questions about the reliability and safety of the code they produce.

Software Quality Metrics and Code Trustworthiness

Established software metrics provide an important capability to quantitatively evaluate the quality and maintainability of source code, two traits closely related to trustworthiness in software. Cyclomatic complexity, introduced by McCabe in 1976, measures the number of independent paths through a program's control flow graph (McCabe, 1976). Intuitively, this metric counts the branching points (such as loops or conditionals) in a section of code. A higher cyclomatic complexity indicates more branching logic, which is often associated with code that is harder to test and maintain (Squire et al., 2020). Another metric, Halstead complexity, takes a different approach, quantifying the abstract volume or effort of code based on its operators and operands (Halstead, 1977). Halstead's Volume, for instance, correlates with the size of an implementation by counting the occurrences of distinct and total symbols in the code. Both cyclomatic complexity and Halstead measures can be aggregated into composite metrics like the Maintainability Index, which combines these metrics with source lines of code (SLOC) to gauge how easy a software module is to modify or extend (Squire et al., 2020).

Generally, simpler code (lower complexity and smaller volume) is considered easier to understand, test, and maintain, thereby reducing the risk of defects. Traditional software engineering research has consistently linked high complexity to increased error-proneness and maintenance costs (Squire et al., 2020). While lower complexity often correlates with greater trustworthiness (simpler code is easier to understand, test, and maintain), it is not a guarantee. For instance, an overly simplistic solution might omit necessary functionality. Complexity metrics must therefore be interpreted in context—while simpler code is generally preferred for reliability and maintainability, it is essential to ensure that necessary functionality and robustness are not sacrificed in pursuit of simplicity.

In the context of LLM-generated code, these metrics reveal meaningful differences when compared to expert-written human code: Prior research presents mixed findings on the structural complexity and maintainability of LLM-generated code; while some studies report that LLM outputs are more complex and potentially harder to maintain (Abbassi et al., 2025), others suggest that such code tends to be simpler and more uniform than human-written counterparts (Takerngsaksiri et al., 2025). This variability highlights the importance of context-specific evaluation, particularly when assessing trustworthiness and review effort in operational settings. Even in cases where AI-generated solutions are functional, the presence of elevated complexity or excessive structural variation may undermine trust in their suitability for operational use. These quantitative measures, alongside structural analyses such as AST depth and subtree diversity, offer an initial basis for assessing whether LLM-generated code aligns with established human engineering practices or deviates in ways that may warrant caution.

Abstract Syntax Tree Analysis for Code Structure

While metrics are useful for condensing code into qualitative numbers, analyzing the AST of code offers a structural perspective. An AST is a tree representation of the source code's syntax, wherein each node corresponds to a language construct (e.g. expressions, statements, blocks). Unlike raw text, the AST explicitly captures the hierarchy and nesting of code constructs (for example, which statements are inside a loop or conditional). AST-based analysis is widely used in compilers and program analysis tools to examine code semantics and enforce coding standards. By examining AST patterns, it becomes possible to see structural patterns and irregularities that are often missed by inspecting the code line-by-line. For instance, an AST can reveal deeply nested logic, repeated subtrees (which might indicate copied boilerplate or recurring patterns), or unusual syntax usage, all of which characterize the code's design.

Using an AST for analysis is especially meaningful in assessing LLM-generated code because it allows direct comparison of structural patterns between code artifacts. The representation of code as an AST enables in-depth analysis of its structural characteristics, yielding insight into the code's inherent patterns and behaviors. Prior research has leveraged AST analysis in various domains (e.g. malware detection and code plagiarism detection) to effectively compare code structures (Rose et al., 2025). In the context of LLM outputs, AST analysis can highlight if the model tends to produce certain idiosyncratic structures – such as excessive chaining of conditions, repetitive try-catch blocks, or other signatures – which might affect readability or reliability. Moreover, ASTs are the foundation for applying graph-based machine learning (like tree kernels or GNNs) and rule-based checks on code. Thus, incorporating AST-based code analysis in a verification study provides a rigorous way to assess whether LLM-generated code conforms to expected structural norms or exhibit patterns that merit caution.

METHODOLOGY

This section outlines the construction of a curated dataset, the extraction of software metrics and structural features, and the development of a machine learning classifier to distinguish human-written from LLM-generated code. By combining static code analysis, AST-based features, and code embeddings, this methodology enables a rigorous and interpretable comparison of code quality and structure across both sources.

Dataset Construction

A similar dataset was created to build upon the previous verification study (Bearss, 2024). Two complementary code datasets were curated, one human-written and one LLM-generated, to enable a rigorous comparison. All code samples are exclusively Python. Selecting a single language eased parsing logic and ensured a consistent syntax across all samples. The human-written corpus consists of approximately 6,000 *s*amples collected from 20 public GitHub repositories that contain solutions to LeetCode programming problems. These repositories were selected based on their reputation (e.g., number of stars) to target high-quality code that conformed to best practices. To avoid any contamination from LLM-assisted commits, only code samples from commits dated before January 1, 2022 (a cutoff chosen to precede the widespread adoption of code-generating LLMs) were included. This filtering helped to ensure that the human-written set reflects genuine human coding practices of that era, uncontaminated by generative AI tools.

The LLM-generated code corpus comprises roughly 6,000 Python solutions produced by querying three state-of-the-art LLMs specialized in code generation. In particular, three open-source models with high performance in coding tasks were selected:

- Code LlaMA 34B
- Qwen 2.5 Coder 32B
- WizardCOder 33B

These models were chosen because they are among the strongest publicly available at code generation ability (Xu & Sheng, 2025), and can be run locally, facilitating reproducibility and cost-effectiveness. The first 400 LeetCode problems that were publicly available and listed at a low or medium difficulty were selected (problem IDs 1 through 542) as the prompt set, covering a broad range of algorithmic challenges. For each problem, each of the three models was prompted five times to generate a solution. To encourage diversity of responses, a `temperature` of 0.5 and a `top_p` of 0.9 were used. The `temperature` parameter controls sampling randomness; a value of 0.5 introduces moderate variability while still favoring high-probability tokens. The `top_p` parameter restricts sampling to the most likely tokens whose cumulative probability exceeds 0.9, reducing unlikely outputs while allowing diversity. This setup helped ensure varied outputs across the five samples per model, yielding up to 15 candidate solutions per problem (5 per model) and about 5,800 LLM-generated functions in total (not all model executions produced runnable code). In addition to these local models, a small dataset was also created using ChatGPT 4.1 (an updated model with coding improvements introduced in April 2025). To reduce costs, only 400 samples (1 sample per problem) were generated. This dataset was kept separate and is explicitly annotated in later sections.

Code Analysis

To quantitatively evaluate the quality and complexity of each code sample, a suite of software engineering metrics were computed using the Radon library (Radon, n.d.). Radon is a static analysis tool that parses Python source code into an AST and computes well-known code metrics. In this study, three metrics were chosen*:* cyclomatic complexity*,* Halstead complexity measures*,* and the maintainability index. Cyclomatic complexity is defined as the number of linearly independent paths through a program's source code (roughly, the count of decision points + 1). This metric reflects the complexity of the control flow: a higher cyclomatic complexity indicates more branching logic, which can correlate with harder-to-test, less maintainable code. In addition, Radon computes Halstead's complexity metrics (including measures such as operator count, operand count, program length, vocabulary, volume, difficulty, and effort) which quantify complexity based on lexical properties of the code. These Halstead metrics capture a different aspect of complexity by analyzing the code's tokens and can highlight differences in how verbosely or concisely the code is written.

In addition, the Maintainability Index, a composite score introduced by Oman and Hagemeister that combines cyclomatic complexity, Halstead volume, and lines of code, was used to assess code quality. In Radon's implementation of maintainability index, a scaled value (with higher values indicating more maintainable code) is generated. By collecting these metrics for every function in both datasets, a quantitative profile of code complexity and maintainability is created. This allows the comparison of human-written versus LLM-generated code on objective grounds. For example, prior studies have suggested that AI-generated code can be more complex or "over-engineered" than human solutions (Licorish et al., 2025), which can be directly tested via these metrics. All metric calculations were executed in an identical manner for both corpora to ensure fairness.

While aggregate metrics are useful, they do not capture deeper structural patterns in code. An in-depth AST analysis was performed to analyze the structural characteristics of the code. First, basic structural features from each AST were extracted in an interpretable way. These features include the tree depth (maximum AST node nesting level), the distribution of node types (frequency of different syntactic constructs such as loops, conditionals, function calls, etc.), and counts of unique subtrees. Tree depth serves as a proxy for nesting complexity. For instance, deeply nested code (high AST depth) may indicate complex logic or multiple layers of loops/conditionals. The node type distribution provides a fingerprint of the code's structure (e.g., how many `If` nodes or `For` loops are present relative to the total nodes). Unique subtrees in the code were also counted as a way to gauge structural repetition or diversity. A high count of unique subtrees suggests the code has many distinct structural patterns (potentially indicating more complex or handcrafted logic), whereas a lower variety with repeated subtrees might indicate templated or boilerplate segments. These AST-derived features are language-agnostic structural indicators that complement the numeric metrics described earlier. LLM-generated code is expected to exhibit distinctive structural signatures such as deeper ASTs and more regular, uniform patterns in loops, spacing, and assignments, which can be captured by these features (Orel, Azizov, & Nakov, 2025).

In addition to looking at only these structural features, additional analysis was done to compare the code structure holistically. Ideally, each sample's AST could be compared to compute a tree edit distance, however, this pairwise comparison is intractable for such large sample sizes. Instead, each AST was transformed into a vector representation (an embedding) that encapsulates its overall shape. A pre-trained code encoder with AST-awareness was used to vectorize the code. The UniXcoder model, which learns a joint representation of code from both the source text and the AST context, was leveraged (Orel, Azizov, & Nakov, 2025). Through the use of these advanced representations, subtler patterns were captured – for example, whether the code exhibits an idiomatic structure commonly used by humans or an unusual but correct structure that might be produced by an LLM.

Classification and Explainability

Finally, a binary classifier was developed using a combination of software engineering metrics, AST-derived structural features, and code embeddings to differentiate between human-written and LLM-generated code. Specifically, each code sample was represented by a feature vector that included cyclomatic complexity, Halstead volume, maintainability index, maximum AST depth, unique subtree count (normalized per line of code), and a 768-dimensional embedding generated by the UniXcoder model. All features were standardized, and the dataset was partitioned into a stratified 80/20 training and test split to preserve the class distribution. An XGBoost classifier was trained on the training data, with hyperparameters selected and validated using 5-fold cross-validation.

To ensure interpretability and transparency in the classifier's decisions, SHAP analysis was performed. SHAP values quantified the contribution of each feature, revealing that the unique subtree count per line of code was the most influential predictor, strongly distinguishing LLM-generated code. AST depth also contributed meaningfully, though to a lesser extent. The UniXcoder embedding dimensions provided additional signal but had lower individual impact compared to the structural features. This interpretable modeling approach yielded a highly accurate classification mechanism while offering clear insights into the structural and stylistic differences between human and AI-generated code.

RESULTS

This section presents the results of the empirical evaluation comparing human-written and LLM-generated code. It includes analyses of code complexity metrics, structural patterns derived from abstract syntax trees (ASTs), semantic embedding visualizations, and the performance of a machine learning classifier trained to distinguish between the two code sources.

Comparison of Code Complexity Metrics

The cyclomatic complexity analysis reveals significant differences between human-written and LLM-generated Python solutions. The LLM-generated code artifacts consistently exhibit higher complexity relative to human-authored code. The mean cyclomatic complexity values are shown in Table 1. These results were tested statistically to confirm significance. Both groups (human and LLM) showed significant departures from normality (Shapiro-Wilk, $p < 0.001$ for each group). Thus, non-parametric testing was appropriate. The Mann–Whitney U test comparing the complexity of human and LLM code yielded a highly significant result ($U = 14773053.5$, $p = 1.529e-80$), confirming a consistent and statistically robust difference in complexity distributions. Additionally, Welch's t-test (robust to unequal variances) produced a similar outcome ($t = 9.15$, $p = 7.289e-20$), reinforcing the statistical significance of this complexity difference.

The analysis of Halstead complexity metrics reveals significant quantitative differences between human-written and LLM-generated code. Specifically, the Halstead Volume—a measure representing the total amount of information (in bits) in the program's code—was substantially lower for LLM-generated solutions compared to the human-authored solutions. The mean values for Halstead Volume are also shown in Table 1. Statistical tests confirmed the significance of these differences. Both distributions (human and LLM-generated) significantly deviated from normality (Shapiro-Wilk, $p \leq 0.001$). Therefore, the non-parametric Mann–Whitney U test was applied, showing a significant difference in Halstead Volume between human LLM code (U = 8594984, p = 2.393e-130). Welch's t-test similarly produced highly significant results (t = -11.08, p = e.406e-28). Thus, despite LLM-generated code exhibiting higher cyclomatic complexity, it is quantitatively simpler in terms of token variety and total tokens used.

This pattern could reflect inherent stylistic differences in the samples. Human-authored solutions from reputable repositories often use sophisticated or idiomatic Python constructs, yielding richer and denser information per code unit. Conversely, LLM-generated code might tend toward simpler, more repetitive patterns with fewer distinct constructs, resulting in lower information density per line or function.

The maintainability index similarly supports these observations. The human-written code scores higher on the maintainability index compared to the LLM-generated solutions. Statistical tests confirmed the significance of this difference between human and LLM samples (Mann–Whitney U = 10351025.5, p = 1.662e-34; Welch's t-test, t = -10.12, p = 6.192e-24). Despite having lower Halstead complexity, the LLM-generated solutions' higher cyclomatic complexity negatively impacts their maintainability index, resulting in lower scores. Nonetheless, the mean maintainability values remain relatively high across all groups, indicating generally good code readability and maintainability. The small but statistically significant decrease in maintainability for LLM-generated code highlights subtle yet meaningful distinctions in the structural and stylistic qualities of LLM-generated versus highly optimized human-written solutions.

Table 1 Software Engineering Metrics

	Cyclomatic Complexity	Halstead Volume	Maintainability Index
Human	2.943656	134.333262	74.513353
Local LLMs	3.445247	72.869653	71.599194
ChatGPT 4.1	3.445590	86.913086	69.584948

These quantitative findings reveal a critical insight: although LLM-generated code has lower lexical complexity, it tends toward more complex logical branching, resulting in mixed implications for maintainability and cognitive readability. Thus, organizations employing LLM-based code-generation tools must weigh these trade-offs carefully, ensuring adequate reviews and validation processes to maintain high standards of code quality, clarity, and maintainability in operational software environments.

AST Structural Analysis

In addition to traditional software metrics, an analysis of AST structures was conducted to explore deeper structural distinctions between human-written and LLM-generated code. Each Python code sample was parsed into its corresponding AST using Python's built-in `ast` library. Three primary structural characteristics were extracted:

- AST Depth: Maximum nesting depth within the code, serving as an indicator of structural complexity.
- Unique Subtree Count: Reflecting structural diversity, measured by counting distinct AST subtree patterns within each sample.
- Node Type Distribution: Frequency and variety of syntax elements present in the code.

The computed AST metrics revealed subtle yet meaningful differences between human-authored and LLM-generated code samples. Human-written code exhibited a slightly lower average AST depth compared to LLM-generated code. Although the numerical differences in AST depth were modest, the consistency of this result indicated that LLM-generated solutions tended toward marginally deeper nesting structures, consistent with the higher cyclomatic complexity observed previously.

The unique subtree counts demonstrated a pronounced and significant contrast. Human-written solutions possessed notably fewer unique AST subtree patterns on average, reflecting concise and highly optimized code structures. In contrast, LLM-generated solutions contained dramatically higher counts of unique AST subtrees. This stark difference suggests that LLM-generated solutions frequently introduced redundant or unnecessarily complex structural variations, possibly indicating a less efficient or less optimized solution style compared to human expert-authored solutions.

Table 2 AST Analysis

	AST Depth	Unique subtrees / loc
Human	18.988707	4506.647903
Local LLMs	19.541204	18522.593408
ChatGPT 4.1	20.000000	17315.515107

To complement manual AST feature analysis, vectorized code embeddings were generated using the pre-trained UniXcoder transformer model, capturing deeper semantic and structural nuances. Subsequently, dimensionality reduction was performed via t-distributed Stochastic Neighbor Embedding (t-SNE) to visualize potential clustering patterns within the high-dimensional embedding space.

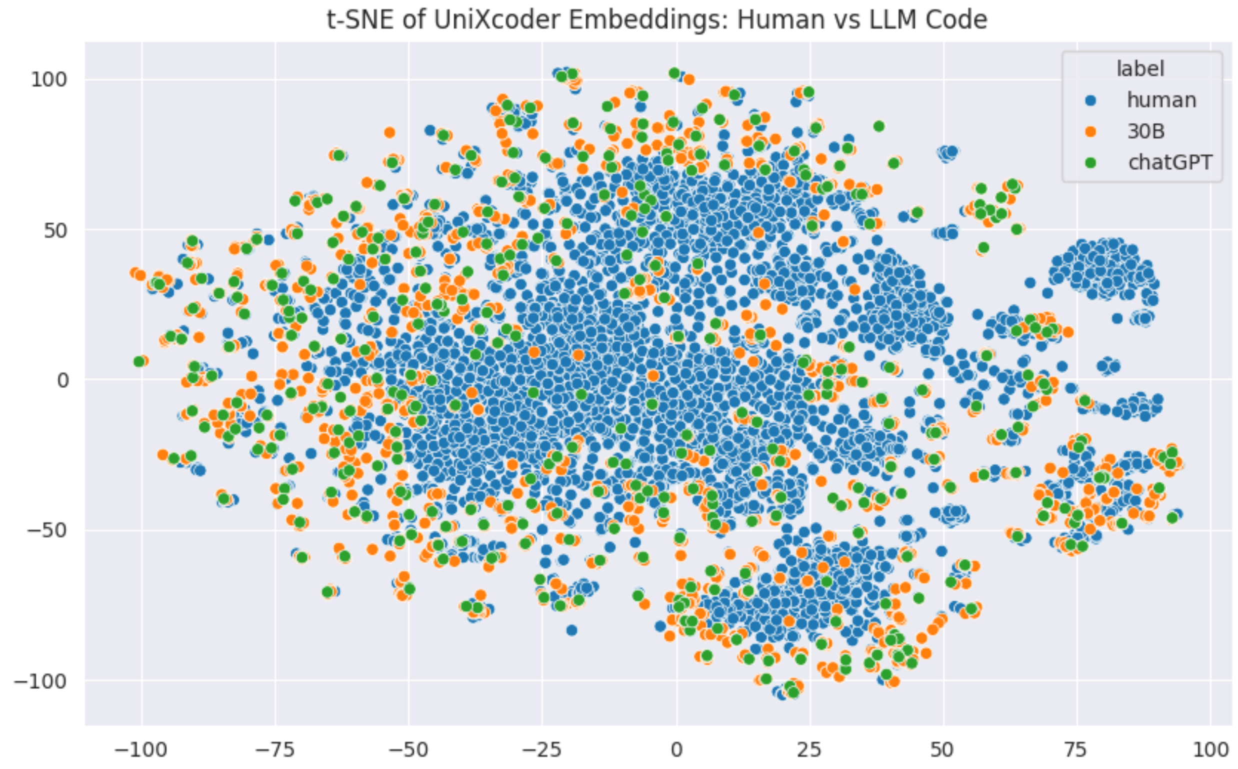

Figure 1 t-SNE Visualization of UniXcoder-Generated Embeddings of Samples

Upon visual inspection of the resulting t-SNE plot (Figure 1), clear but not strictly separable clusters emerged. Human-generated code distinctly clustered in certain regions, separate from LLM-generated code, although substantial overlap remained. Minor clustering patterns between different LLM models (ChatGPT and local models) were also present but subtle. Efforts to algorithmically separate human and LLM-generated clusters using K-Means clustering proved unsuccessful, as meaningful linear or cluster-based separations were not achievable.

The AST-based analysis revealed quantifiable and meaningful structural differences between human-written and LLM-generated code, complementing earlier complexity metric findings. Human solutions were structurally concise, demonstrating modest nesting depth and fewer unique AST patterns indicative of optimized and idiomatic coding practices. In contrast, LLM-generated code exhibited substantially greater structural diversity and slightly increased nesting depth, often reflecting redundant or overly elaborate solution patterns. While these structural distinctions were visually apparent via t-SNE embeddings, automated clustering methods such as K-Means struggled to differentiate the groups clearly, highlighting the nuanced nature of these differences. Ultimately, these results emphasize the critical value of structural AST analyses in evaluating LLM-generated code, particularly for applications within high-stakes or operationally critical software development environments.

Machine Learning Classification of Code Authorship

To robustly differentiate between human-authored and LLM-generated code, an XGBoost classifier was trained using a diverse set of features, including traditional software engineering metrics (cyclomatic complexity, Halstead volume, and maintainability index), AST-based structural metrics (AST depth and unique subtrees per line of code), and high-dimensional embeddings from the UniXcoder model. A stratified 80/20 train-test split ensured balanced representation across classes and varying code lengths.

The classifier exhibited exceptional predictive performance, achieving 99% accuracy with precision, recall, and F1-scores consistently at or above 0.99. The confusion matrix demonstrated near-perfect separation, correctly classifying 1,413 out of 1,420 human-written samples and 749 out of 761 LLM-generated samples, indicating reliable distinction based on these combined features.

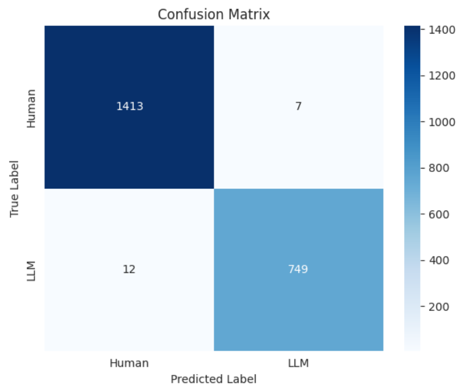

Figure 2 Classifier Confusion Matrix

Table 3 Classifier Metrics

	Precision	Recall	F1-score	support
Human	0.99	1.00	0.99	1420
LLM	0.99	0.98	0.99	761
accuracy			0.99	2181

SHAP analysis was conducted to interpret the model's decision-making transparently. The results (shown in Figure 3) identified the normalized count of unique AST subtrees per line of code (unique_subtrees_per_loc) as the most influential predictor, clearly distinguishing LLM-generated from human-authored code. AST depth was also influential, with various embedding dimensions contributing to the classification decisions. These findings confirm the structural characteristics, particularly the extent of subtree repetition and structural depth, are strong indicators of code authorship.

FUTURE WORK

As LLMs continue to evolve, future research will extend this verification framework to evaluate newer, more capable models. It will be important to assess whether advances in model architecture and training lead to measurable improvements in code quality, structural efficiency, and alignment with human engineering practices. Additionally, real-world software development rarely involves a single LLM interaction. Developers often iteratively prompt, modify, and combine AI-generated code, a process that may amplify structural issues or introduce new risks. Expanding this analysis to include multi-turn, real-world coding workflows will provide a more realistic assessment of LLM-generated code trustworthiness in operational environments.

CONCLUSION

In this study, the trustworthiness and structural quality of LLM-generated code was rigorously analyzed and compared against highly optimized, human-authored Python solutions. Significant structural and complexity differences between human-written and LLM-generated code were identified through quantitative analyses. Specifically, LLM-generated code consistently demonstrated higher cyclomatic complexity, lower Halstead volume, and lower maintainability indices compared to human-authored solutions. Additionally, AST-based analysis revealed substantially higher structural diversity within LLM-generated samples, characterized by increased unique subtree counts, indicating redundancy or less optimized structural patterns relative to human-written code.

A highly accurate and explainable machine learning classifier, trained on a diverse feature set combining software engineering metrics, AST structural features, and semantic embeddings, successfully distinguished human from LLM-generated code with 99% accuracy. SHAP interpretability analysis revealed that normalized unique subtree counts (subtrees per line of code) was the strongest predictor, followed by AST depth and various embedding dimensions, underscoring subtle yet significant structural and stylistic differences indicative of code authorship.

Importantly, the code samples evaluated in this study represent a "best-case" scenario: clearly defined programming problems solved through a single LLM interaction. In real-world software development contexts, developers frequently engage in iterative "vibe coding," involving multiple interactions with LLMs to refine solutions for problems that are often ambiguously specified. Such iterative workflows are likely to exacerbate the identified structural issues, producing code that is increasingly complex, less maintainable, and prone to hidden defects. Consequently, the observed structural and complexity differences in this controlled study likely represent a lower bound of the practical challenges associated with deploying LLM-generated code. These findings strongly advocate for rigorous verification and careful review processes when incorporating LLM-generated solutions into operational software environments to maintain high standards of reliability, maintainability, and trustworthiness.

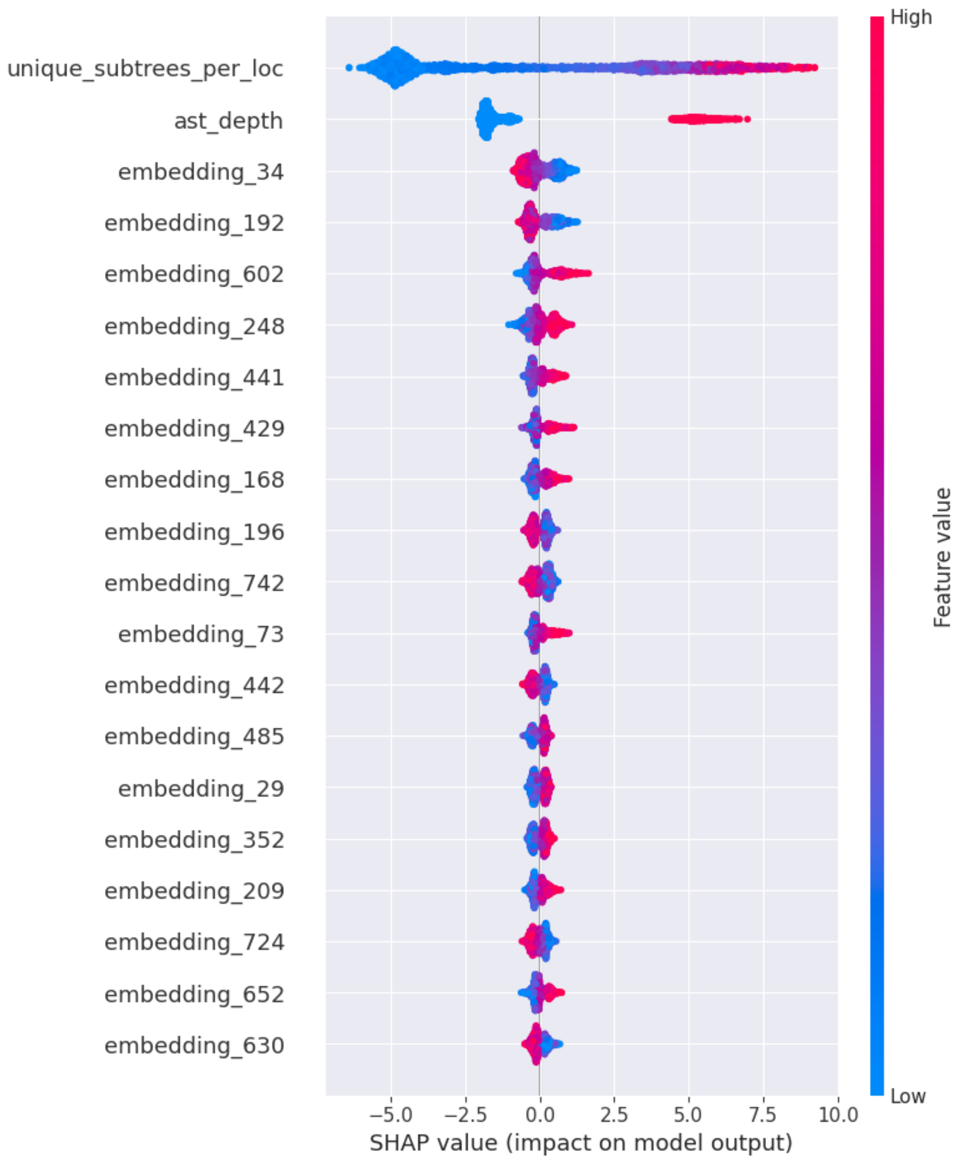

Figure 3 SHAP Analysis of Classifier

REFERENCES

Abbassi, A. A., Da Silva, L., Nikanjam, A., & Khomh, F. (2025). Unveiling inefficiencies in LLM-generated code: Toward a comprehensive taxonomy. arXiv. https://arxiv.org/abs/2503.06327

Ashrafi, N., Bouktif, S., & Mediani, M. (2025). *Enhancing LLM code generation: A systematic evaluation of multi-agent collaboration and runtime debugging for improved accuracy, reliability, and latency*. arXiv. https://arxiv.org/abs/2505.02133

Bearss, E. (2024, December). *Evaluating the trustworthiness of large language models for code generation*. I/ITSEC 2024 Conference.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv. https://arxiv.org/abs/2107.03374

Cramer, M., & McIntyre, L. (2025). *Verifying LLM-generated code in the context of software verification with Ada/SPARK*. arXiv. https://arxiv.org/abs/2502.07728

Halstead, M. H. (1977). *Elements of software science*. Elsevier North-Holland.

Licorish, S. A., Bajpai, A., Arora, C., Wang, F., & Tantithamthavorn, K. (2025). *Comparing human and LLM generated code: The jury is still out!* arXiv. https://arxiv.org/abs/2501.16857

Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L., & Liu, Y. (2024). *LMs: Understanding code syntax and semantics for code analysis*. arXiv. https://arxiv.org/abs/2305.12138

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Orel, D., Azizov, D., & Nakov, P. (2025). *CoDet-M4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings*. arXiv. https://arxiv.org/abs/2503.13733

Radon. (n.d.). *Radon documentation: Introduction*. Read the Docs. https://radon.readthedocs.io/en/latest/intro.html

Rose, A. J., Schubert Kabban, C. M., Graham, S. R., Henry, W. C., & Rondeau, C. M. (2025). Malware classification through abstract syntax trees and L-moments. *Computers & Security, 148*, 104082. https://doi.org/10.1016/j.cose.2024.104082

Squire, M. D., Maynard-Nelson, L. A., Brown, T. A., Crumbley, R. T., Holzmann, G. J., Jennings, M., Luu, K., Moleski, W. F., & Marchetti, J. D. (2020). *Cyclomatic complexity and basis path testing study* (NASA Technical Memorandum NASA/TM−20205011566; NESC-RP-20-01515). NASA. https://ntrs.nasa.gov/api/citations/20205011566/downloads/20205011566.pdf

Takerngsaksiri, W., Fu, M., Tantithamthavorn, C., Pasuksmit, J., Chen, K., & Wu, M. (2025). *Code readability in the age of large language models: An industrial case study from Atlassian*. arXiv. https://arxiv.org/abs/2501.11264

Xu, Z., & Sheng, V. S. (2025). *CodeVision: Detecting LLM-generated code using 2D token probability maps and vision models*. arXiv. https://arxiv.org/abs/2501.03288