Can LLMs Generate Higher Quality Code Than Humans? An Empirical Study

Mohammad Talal Jamil
Department of Computer Science
Lahore University of Management Sciences
DHA, Lahore, Pakistan
23030005@lums.edu.pk

Shamsa Abid
Department of Software Engineering
National University of Computer and Emerging Sciences
Chiniot-Faisalabad Campus, Pakistan
shamsa.abid@nu.edu.pk

Shafay Shamail
Department of Computer Sciences
Lahore University of Management Sciences
DHA, Lahore, Pakistan
sshamail@lums.edu.pk

***Abstract*—Large Language Models are being extensively used for AI-assisted programming and code generation. The challenge is to ensure that the generated code is not only functionally correct but also safe, reliable and trustworthy. In this direction, we conduct a comprehensive empirical analysis of AI-generated code to assess whether large language models (LLMs) can produce correct and higher-quality code than humans. We evaluate the code quality of 984 code samples generated by GPT-3.5-Turbo and GPT-4 using various prompt types (simple, instructional, and enhanced) against input queries from the HumanEval dataset. We also enhance the HumanEval benchmark by calculating code quality metrics for the human-written code it contains. Code quality metrics are calculated using established tools like Radon, Bandit, Pylint, and Complexipy, with human-written code serving as a baseline for comparison. To quantify performance, we employ the TOPSIS method to rank the models and human code by their proximity to ideal and anti-ideal code quality metrics. Our results demonstrate that GPT-4, when used with advanced prompts, produces code closest to the ideal solution, outperforming human-written code in several key metrics. Our work provides evidence that LLMs, when properly guided, can surpass human developers in generating high-quality code. Our code and datasets are available online.**

***Index Terms*—Large Language Models (LLMs), AI-assisted programming, code generation, code quality assessment, code quality metrics, trustworthy AI, GPT, HumanEval**

I. Introduction

Large Language Models, which are pre-trained on vast amounts of source code and natural language data, have demonstrated exceptional capabilities in understanding code structures and generating code or texts [1–3]. LLM-based code generation has become a highly active and promising area in AI-assisted programming; delivering remarkable results and becoming widely adopted by developers for software development and code reuse. Nonetheless, ensuring the generated code's correctness, reliability, and trustworthiness remains a significant challenge [4, 5].

Auto-generated code not only needs to be functionally correct but also reliable, safe and of high quality to be useful for developers [6]. While many researchers argue that discussions of code quality should follow once we achieve correctness, we counter that quality assessment is crucial from the outset, as LLMs are already being widely used in production environments [7]. As developers increasingly integrate model-generated code, understanding the quality of this code is essential. A proactive evaluation of quality not only enhances trust in these models but also serves as a safety precaution, providing developers with a quick analysis that can help them assess risks before reuse.

Code quality metrics provide valuable insights for assessing the trustworthiness of code-generating models, encompassing multiple dimensions: correctness, security, maintainability, readability. Existing research studies demonstrate the impact of code metrics on code quality [8, 9], providing evidence that certain metrics consistently influence it and serve as reliable indicators of code quality [10, 11]. Building on this evidence, this paper leverages code metrics to assess the correctness and overall quality of code. In terms of safety, which in this context refers specifically to code security, static analysis metrics are invaluable for detecting common risks like SQL injection and buffer overflows, identifying vulnerabilities that could compromise the generated code's reliability and trustworthiness. Maintainability is yet another vital factor, as code readability and adherence to coding standards significantly impact how easily developers can understand and work with the generated code. Modular, readable code contributes to long-term reliability, making it easier to test and refactor as necessary.

This paper investigates the quality of code generated by LLMs, particularly GPT-3.5-Turbo [12] and GPT-4 [13], in comparison with human-authored code. We leverage the HumanEval dataset [3], a widely-used collection of 164 human-written Python code samples, to calculate a range of code quality metrics, establishing a baseline for human-written code quality and reliability. By comparing these baseline metrics with those of code generated by various prompt and model configurations of GPT, we aim to determine whether LLMs can indeed generate higher-quality code than humans. Our methodology involves evaluating code quality metrics across different model-generated codebases resulting from different prompt designs—simple, instructional, and enhanced—to understand how prompting impacts the overall code quality produced by these models. We have the following main RQ and sub RQ:

- RQ 1: Can GPT models generate better quality code than humans?

- RQ 1.1: Which combination of prompt and GPT model gives the closest to ideal code quality metrics?

To answer the research questions, we calculate the code quality metrics of human-written and model-generated codebases using established tools like Radon [14], Bandit [15], Pylint [16], and Complexipy [17]. To quantify performance, we employ the TOPSIS [18] method to rank the model-generated and human-written codebases. Our code and datasets are available online [19].

In this paper, we make the following main contributions:

- Enriched HumanEval dataset with comprehensive code quality metrics.
- Devised progressively complex prompts to generate code from GPT-3.5-Turbo and GPT-4 for HumanEval queries.
- Generated six unique codebases by executing a set of prompts on GPT-3.5-Turbo and GPT-4 for HumanEval queries.
- Compared model-generated code for correctness and broader quality factors against human-written code.
- Ranked human-written and model-generated codebases separately on critical (correctness, security, etc.) and human-centered (readability, maintainability etc.) metrics using TOPSIS analysis.
- Developed scripts for prompts execution, automated code quality metrics calculation and test case execution for reuse and replicability.

In summary, our comprehensive analysis of code quality metrics helps to evaluate LLM-generated code across several dimensions, building an understanding of its reliability, security, and maintainability among other quality factors. This paper provides a detailed assessment of these metrics - in comparison to human-written code metrics - to evaluate GPT models' effectiveness in producing high-quality code, thereby contributing to the broader discourse on trustworthiness in AI-driven software development.

II. Motivating Example

The three solutions presented here for the *rolling_max* function from the HumanEval dataset, and our custom prompts to GPT-3.5-Turbo, and GPT-4—demonstrate how AI models can generate code that is not only functionally correct but also sometimes optimized beyond human-written solutions. Here, optimization refers to improvements in code quality metrics such as correctness, readability, maintainability, security, and adherence to best coding practices. Analyzing key quality metrics such as Cyclomatic Complexity (CC), Cognitive Complexity (CogC), Maintainability Index (MI), and Lines of Code (LOC) reveals insights into each code's efficiency, readability, and maintainability.

The HumanEval solution is fairly robust and clear, utilizing a loop to maintain a *running_max* variable and iteratively appending the maximum value encountered so far. However, this approach includes multiple lines for handling the initialization and comparison, resulting in moderate Cyclomatic

```python
from typing import List, Tuple


def rolling_max(numbers: List[int]) -> List[int]:
    running_max = None
    result = []

    for n in numbers:
        if running_max is None:
            running_max = n
        else:
            running_max = max(running_max, n)

        result.append(running_max)

    return result
```

Listing 1. Human-written code from HumanEval 9.py

```python
from typing import List


def rolling_max(numbers: List[int]) -> List[int]:
    if not numbers:
        return []

    result = [numbers[0]]
    for i in range(1, len(numbers)):
        result.append(max(result[-1], numbers[i]))

    return result
```

Listing 2. GPT-3.5-Turbo generated code

```python
from typing import List
from itertools import accumulate


def rolling_max(numbers: List[int]) -> List[int]:
    return list(accumulate(numbers, max))
```

Listing 3. GPT-4.0 generated code

Fig. 1. Code for rolling_max functionality

TABLE I
Code metrics for example 9.py from HumanEval and from code generated by GPT-3.5-Turbo and GPT-4 with special prompts

Codebase	LOC	CC	MI	CogC	Pass
HumanEval	16	3	72.1	3	True
GPT-3.5-Turbo	12	3	73.57	2	True
GPT-4	6	1	100	0	True

and Cognitive Complexity, which can impact readability and maintainability. The MI is decent at 72.1, indicating good maintainability, but the code could be simplified.

The GPT-3.5-Turbo solution simplifies initialization by using a list with the first element of numbers directly, avoiding the None check used in HumanEval. This leads to a slight reduction in Cognitive Complexity, making it easier to follow the control flow with fewer decision points. The reduced LOC (12) and slightly higher MI (73.57) show improved maintainability and readability. This solution exemplifies how a model can simplify human-written code by focusing on essential elements.

GPT-4's approach leverages Python's *itertools.accumulate* to compute the rolling maximum in a single, highly optimized line. This dramatically reduces the complexity of the code, with the lowest possible CC (1) and CogC (0), making it extremely easy to maintain and understand. The MI reaches 100, indicating higher maintainability, and the LOC is minimized to just 6. This code is both concise and elegant, demonstrating that GPT-4 can leverage advanced built-in functions to create highly readable and performant solutions, potentially surpassing human-generated solutions in simplicity and efficiency.

This example illustrates how AI models can produce code that is not only functionally correct but also optimized for readability and maintainability. The models progressively simplify the code, with GPT-4 providing a solution that is both shorter and more maintainable than the HumanEval example. Such comparisons highlight the potential for AI models to contribute innovative, high-quality code solutions that emphasize modern programming practices.

Our study's focus on code quality metrics provides a holistic view of model performance, beyond accuracy. By benchmarking model-generated solutions with code quality metrics, we can more rigorously assess models' capability to generate not only correct but also high-quality code. This approach aligns with the goals of prior works that call for enhanced benchmarks and evaluation standards for model reliability, as it sets a new foundation for understanding and improving code generation models' effectiveness in producing production-ready code.

III. Background

A. HumanEval Dataset

The HumanEval dataset is a collection of Python programming problems which are written by humans [3]. Each problem includes a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem. It consists of 164 original programming problems, assessing language comprehension, reasoning, algorithms, and simple mathematics. The intended usage of the HumanEval dataset is to evaluate the functional correctness and problem-solving capabilities of various code generation models against the HumanEval dataset. In our work, we not only evaluate the functional correctness, we also evaluate the code quality metrics for a comparison again human-written code quality. For the code quality comparison, we first generate the code quality metrics of the human-written programs in the HumanEval dataset and for the code produced by the code generation models, and then perform a comparative analysis to determine the best quality codebase.

B. Code Metrics Tools

To evaluate code quality across human-written and AI-generated codebases, we leverage four widely used static analysis tools: Radon [14], Complexipy [17], Bandit [15], and Pylint [16]. The selection of these tools for our study was driven by their alignment with our study's focus on evaluating a combination of human-centric and critical code quality metrics. These tools provide complementary metrics that collectively address diverse aspects of code quality, enabling a balanced analysis of code. Additionally, they are widely used, in both academic and practical contexts [20–22], and readily integrable into automated evaluation pipelines. However, despite their practical applicability, we recognize certain limitations of these tools, which we address in our discussion of threats to validity (Section VII-B). Each of these tools provides specific insights into different aspects of code quality:

- Radon [14]: Radon is a Python tool that computes various metrics from the source code. Radon can compute McCabe's complexity, i.e. cyclomatic complexity, raw metrics (LOC, LLOC, SLOC, comment lines etc.), Halstead metrics, and Maintainability Index. Metrics like cyclomatic complexity and maintainability index help identify areas of high logical complexity that may affect readability and maintainability. Radon outputs numerical scores that represent the complexity of individual functions or modules, with higher values often indicating greater complexity.
  Cyclomatic Complexity (CC) corresponds to the number of decisions a block of code contains plus 1. This number (also called McCabe number) is equal to the number of linearly independent paths through the code.
  Maintainability Index (MI) is a software metric which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of SLOC (Source Lines Of Code), Cyclomatic Complexity and Halstead volume.
  Halstead's metrics form a set of quantitative measures designed to assess the complexity of software based on operator and operand usage within the code. Using the base values of operators and operand, Halstead's metrics calculate additional measures to estimate the cognitive and structural effort involved in understanding and maintaining code:
  - Program Vocabulary (η): Defined as the sum of distinct operators and distinct operands. This metric reflects the unique linguistic elements in the code, with a larger vocabulary suggesting a more complex codebase.
  - Program Length (N): The total number of operators and operands, representing the size of the program in terms of tokens used. Program length can indicate code verbosity, where excessive length may imply redundancy or inefficiency.
  - Volume (V): Volume measures the size of the program in "mental space" and indicates the amount of information the code contains. Higher volume implies increased difficulty in understanding the program, as it occupies more cognitive resources.
  - Difficulty (D): This metric quantifies the difficulty of understanding the program. Difficulty highlights how challenging the program might be for developers, with higher difficulty suggesting a more intricate

flow of logic.

  - Effort (E): Effort provides an estimation of the mental workload needed to implement or maintain the code. Effort serves as an indicator of development time, with higher values suggesting increased complexity and, consequently, a greater likelihood of defects.
  - Time required to program (T): This metric provides an estimate of the time required for a programmer to implement or comprehend the code.
  - Number of delivered bugs (B): This metric estimates the potential number of defects in the code, based on its complexity.

- Complexipy [17, 23]: Cognitive complexity (CogC) captures the mental effort required to understand code, emphasizing readability and simplicity. The output consists of cognitive complexity scores, where lower values are preferable as they indicate easier-to-understand code. The complexipy tool for Python code checks the cognitive complexity of a file or function and if it is greater than the default cognitive (15), then the return code will be 1, otherwise it will be 0.
- Bandit [15]: Bandit is a tool which performs static code analysis for potential security vulnerabilities in Python code. Bandit processes each file, builds an AST from it, and runs appropriate plugins against the AST nodes. Once Bandit has finished scanning all the files it generates a report. It scans for common security risks, such as insecure imports or improper use of cryptography. Bandit's output includes a list of issues flagged with severity levels, enabling targeted remediation of critical vulnerabilities.
- Pylint [16]: Pylint is a static code analysis tool that checks for errors, enforces a coding standard, looks for code smells, and can make suggestions about how the code could be refactored. Pylint locates and outputs messages for errors (Err.), potential refactorings (R), warnings (W), and code convention (C) violations in code.

These tools collectively provide a multi-faceted evaluation of code quality, covering complexity, readability, security, and stylistic conformity, which is essential for an objective comparison between human-written and AI-generated code.

C. TOPSIS

The Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) [18] is a multi-criteria decision analysis method. TOPSIS is based on the concept that the chosen alternative should have the shortest geometric distance from the positive ideal solution (PIS) or ideal worst and the longest geometric distance from the negative ideal solution (NIS) or ideal worst. It compares a set of alternatives, normalising scores for each criterion and calculating the geometric distance between each alternative and the ideal alternative, which is the best score in each criterion.

Our goal is to rank various GPT model alternatives with respect to code generation quality and to find the best based on a set of code metrics. The idea is to choose the solution closest to the ideal solution (best values) and farthest from the worst (worst values).

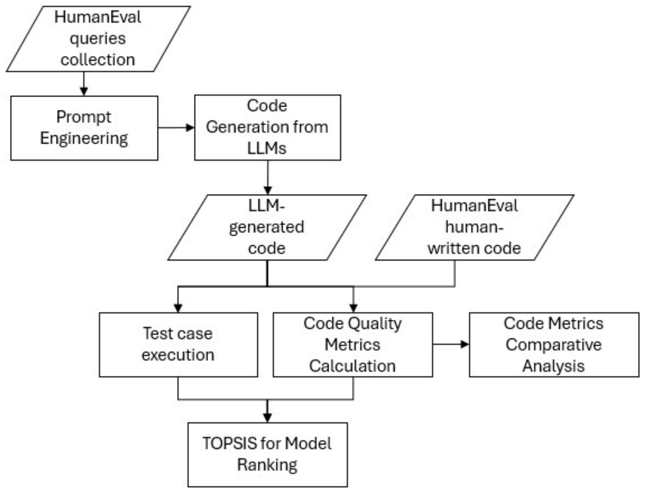
HumanEval queries collection
Prompt Engineering
Code Generation from LLMs
LLM-generated code
HumanEval human-written code
Test case execution
Code Quality Metrics Calculation
Code Metrics Comparative Analysis
TOPSIS for Model Ranking

Fig. 2. A Workflow of Code Quality Evaluation and Ranking of Code Generation Models

TABLE II
MODEL CONFIGURATIONS FOR GPT-3.5-TURBO AND GPT-4

Model	GPT-3.5-Turbo	GPT-4
Context Window	16,385 tokens	8,192 tokens
Max Output Token	4,096 tokens	8,192 tokens
Training Data	Up to Sep 2021	Up to Sep 2021
Hyper-parameter Setting	Default	Default
Temperature	1.0	1.0

IV. METHODOLOGY

Our workflow, illustrated in Figure 2, begins by generating prompts from the 164 HumanEval queries. We create three types of prompts—basic, instructional, and enhanced—which are input to two GPT models; GPT-3.5-Turbo and GPT-4, resulting in six unique model-prompt combinations. By executing each type of prompt on each model, we generate six unique codebases each with 164 code samples. We then evaluate the correctness of the codebases by executing corresponding test cases to determine if the generated code passes the test cases. Table II presents the model configurations for GPT-3.5-Turbo and GPT-4 used in our experiments.

Next, we compute the code quality metrics for each code sample across all codebases, including HumanEval and model-generated outputs. We then conduct a comparative analysis of these code quality metrics and, finally, perform TOPSIS analysis to rank each model based on both critical quality aspects (e.g., correctness and security) and human-centered quality factors (e.g., readability, maintainability, and complexity). To facilitate replication, we automate metrics calculations and dataset generation through scripts.

For given function definition and docstring $\{input\}$ provide the code solution. Output Format: $\langle code \rangle$

Fig. 3. Simple Prompt

For given function definition and docstring $\{input\}$, your task is to provide the code solution and ensure that your completed code passes the given test cases $\{testcases\}$.
• Provide only the function implementation (no test cases, comments, or any extra code). • Do not change the function name in the function definition • Do not change the argument types and names in the function definition • Identify any external module dependencies and import them before the function definition. • There should be no nested functions in your code. All helper functions should be top-level, not defined within other functions. • Do not use undefined functions in your code. • Make sure there are no syntax or semantic errors in your code. Output Format: $\langle code \rangle$

Fig. 4. Instructional prompt

Here is a query $\{input\}$ and a human-written solution $\{code\}$. Your task is to provide a better and innovative code solution than the human-written solution by making your code more optimized, readable, and safe, while reducing complexity. The code must pass the provided test cases $\{testcases\}$. Follow the guidelines below:
• Provide only the function implementation (no test cases, comments, or any extra code). • Do not change the function name in the function definition or argument types and names • Do not change the argument types and names in the function definition • Identify any external module dependencies and import them before the function definition. • There should be no nested functions in your code. All helper functions should be top-level, not defined within other functions. • Do not use undefined functions in your code. • Make sure there are no syntax or semantic errors in your code • Ensure error handling is incorporated if applicable to avoid runtime failures. • Focus on improving performance and code clarity. • Maintain or improve safety and readability in the solution.
Output Format: $\langle code \rangle$

Fig. 5. Enhanced Prompt

A. Prompt Engineering

The goal of prompting a model is to request a model to generate a code solution against an input query, along with any optional special instructions or contextual information. We prompt a model for all 164 code queries from the HumanEval dataset, where each query consists of a function definition and a docstring. We create three distinct prompt types — Simple (S), Instructional (I), and Enhanced (E) — to examine the impact of prompt specificity on model performance. Each prompt provides varying levels of instructions and contextual information to the code models.

In the Simple prompt shown in Figure 3, the prompt simply requests for a code solution against a given function definition and docstring of a code sample from the HumanEval dataset. Additionally, the prompt explicitly defines an output format using a placeholder, guiding the model to produce code as the expected response.

For the Instructional prompt shown in Figure 4, we specify additional instructions on top of the Simple Prompt. First, we specify the relevant test cases from the HumanEval dataset corresponding to code query and instruct the model to ensure the generated code passes the test cases. Additionally, we provide explicit instructions to limit the output by excluding the test cases, comments, or extra code from the response. We also instruct the model to retain the original function name and arguments names and types in the function definition, to not produce nested functions, to use accessible helper functions, to not use undefined functions, to import any external module dependencies, and to ensure that the generated code is free from syntax and semantic errors.

In the Enhanced prompt shown in Figure 5, we further refine these instructions and provide additional context, including the original human-written solution. We instruct the model to provide a better and innovative code solution as compared to the given human-written solution and explicitly request for code optimization through enhanced readability, safety, and lower complexity. While these objectives are desirable, they may sometimes conflict (e.g., conciseness vs. readability); however, we aim for a solution that strikes a near-optimal balance among these competing factors. Additionally, we instruct the model to ensure error handling to avoid runtime failures, and to focus on performance and clarity.

B. Code Generation from LLMs

To generate code using both models GPT-3.5-Turbo and GPT-4, we populate the prompt placeholders with the relevant data for all 164 code queries. This results in 164 prompts for each of the three prompt types. Executing these prompts on both models yields a total of 984 prompt executions. The entire process is fully automated via a Python script.

C. Test Case Execution

The primary purpose of test case execution is to validate whether the generated code passes all the test cases provided in the HumanEval dataset. Test cases are provided for each problem in the HumanEval dataset. To evaluate a generated code's correctness, we developed a Python script that systematically concatenates the generated code with the provided test case code. This script then invokes a 'check' function to automatically verify whether the generated code successfully passes each test case. We record the number of test cases successfully passed by the model-generated codes, along with any assertion errors and runtime errors encountered during execution. The results of test case execution are stored in a csv file.

D. Code Quality Metrics Calculation

We develop a Python script that automatically generates code metrics for the code samples in HumanEval as well for the code samples generated against our six prompt-model combinations. Our script uses the Python libraries for each metric calculation tool: Radon, Complexipy, Pylint, and Bandit. We store the results of each tool's output in csv files.

V. Evaluation Results

A. RQ1: GPT models vs human-written code quality

In this section, we discuss the differences in code quality between human-written code and GPT-generated code, analyzing the metrics provided by each tool.

1) Radon: We obtain code metrics from Radon such as Lines of Code (LOC), Cyclomatic Complexity (CC), and Maintainability Index (MI) to evaluate and compare the quality of code samples from HumanEval against GPT-generated models. Table III presents the results of the Wilcoxon signed-rank test for comparing LOC, CC, and MI across these code samples. The Wilcoxon test, a non-parametric method that does not assume normality, is well-suited for paired comparisons. In this context, it enables us to determine whether the differences in code metric values between HumanEval samples and those generated by GPT models are statistically significant.

Table III shows the Wilcoxon test results for LOC under the column heading T_{LOC} and the respective p-values under the column heading p_{LOC}. We note significant differences in LOC between HumanEval and some of the GPT-generated code. For example, GPT-3.5-S, GPT-4-S, GPT-4-I, and GPT-4-E produce significantly different LOC than HumanEval as indicated by p-values < 0.05. In contrast, GPT-3.5-I and GPT-3.5-E show no significant difference (p-values > 0.05), suggesting that their LOC values are similar to those of HumanEval code. The Wilcoxon test results for CC appear under the column heading T_{CC} and the respective p-values under the column heading p_{CC}. GPT-3.5-E, GPT-4-I, and GPT-4-E all show significant differences (p-values < 0.05), indicating these models produce code with notably different Cyclomatic Complexity. The smallest p-value (0.0047) is seen for GPT-4-E, suggesting it is the most distinct from HumanEval in terms of Cyclomatic Complexity, followed by GPT-4-I. The Wilcoxon test results for MI appear under the column headings T_{MI} and p_{MI}. GPT-3.5-S shows a strong difference with a very low p-value (1.88e-22), indicating that its maintainability is significantly different from HumanEval. The GPT-3.5-I model is almost indistinguishable from HumanEval with a p-value of 0.0529, which is close to the threshold for significance. However, all other models (GPT-3.5-E, GPT-4-S, GPT-4-I, and GPT-4-E) show highly significant differences with p-values well below 0.05, indicating that these models generate code with significantly different maintainability than HumanEval.

2) Complexipy: Complexipy provides the CogC metric for assessing cognitive complexity for code samples in HumanEval and those generated by GPT models. Table III shows the Wilcoxon test results for CogC under the column heading T_{CogC} and the respective p-values under the column heading p_{CogC}. Both GPT-3.5-S and GPT-3.5-I show no significant differences in complexity from HumanEval, with p-values of 0.7576 and 0.4755, respectively, indicating that their complexity is similar to HumanEval. However, the other models show highly significant differences. GPT-3.5-E, with a p-value of 3.09e-07, indicates a significant difference, while GPT-4-S (p-value 0.0166), GPT-4-I (p-value 2.78e-06), and GPT-4-E (p-value 1.35e-09) all exhibit notable differences in complexity compared to HumanEval.

3) Pylint: Table IV compares the Pylint metrics across the HumanEval dataset's canonical solutions and outputs from various GPT model and prompt combinations, focusing on Errors, Refactor suggestions, Warnings, and Convention issues. The canonical solutions set a benchmark with no Errors, moderate Refactor needs (33), minimal Warnings (9), and a higher number of Convention issues (537). GPT-3.5, when prompted with the Simple prompt (GPT-3.5-S), generated code with 8 Errors, 39 Refactor suggestions, 17 Warnings, and 513 Convention issues, reflecting slightly more structural and stylistic issues than the canonical solutions. With the Instructional prompt (GPT-3.5-I), error counts decreased to 7, and Refactor needs dropped to 34, suggesting a minor quality improvement. The Enhanced prompt (GPT-3.5-E) notably reduced Refactor needs to 11, though Errors, Warnings, and Convention issues remained similar, indicating that added context positively impacted code structure. In contrast, GPT-4's Simple prompt (GPT-4-S) generated more Errors (25) and Refactor suggestions (47) but fewer Convention issues (486), pointing to an increased need for structural refinement. The Instructional prompt (GPT-4-I) for GPT-4 maintained 25 Errors but reduced Refactor needs to 42, with a slight decrease in Warnings, suggesting improved alignment with the guidance. Overall, the Enhanced prompt (GPT-4-E) achieved optimal quality with zero Errors, 21 Refactor suggestions, and 527 Convention issues, aligning closely with canonical solution standards. Our results show that the enhanced prompts can improve code quality, especially with GPT-4.

4) Bandit: Table V compares security vulnerabilities flagged by Bandit across the HumanEval dataset's canonical solutions and code generated by various GPT model and prompt combinations, with some differences in issue frequency. The HumanEval solutions serve as a baseline, exhibiting three Bandit-flagged issues, whereas GPT-3.5 models generally showed fewer security concerns. Our results indicate that GPT-generated code can be less prone to security risks than human-authored solutions, especially with simpler prompts, while more detailed prompts may increase security concerns slightly.

5) Test Case Execution Results: The results presented in Table VI summarize the performance of various code generation models against the Human Eval dataset, highlighting key metrics such as the number of test cases passed, assertion errors, runtime errors, and overall pass rates. The canonical solutions from the Human Eval dataset achieved a perfect

TABLE III
WILCOXON TEST STATISTIC (T) AND P-VALUES (p) FOR LOC, CC, MI AND COGC COMPARISONS BETWEEN HUMANEVAL AND GPT MODELS' CODE

Code Comparison	T_{LOC}	p_{LOC}	T_{CC}	p_{CC}	T_{MI}	p_{MI}	T_{CogC}	p_{CogC}
HumanEval vs GPT-3.5-S	502.0	1.42×10^{-22}	1852.0	0.2553	467.0	1.88×10^{-22}	2387.0	0.7576
HumanEval vs GPT-3.5-I	4588.5	0.319	2121.0	0.8008	4566.0	0.0529	2225.5	0.4755
HumanEval vs GPT-3.5-E	3651.0	0.575	1148.0	0.0138	2932.0	3.01×10^{-7}	761.0	3.09×10^{-7}
HumanEval vs GPT-4-S	3352.5	8.22×10^{-5}	1343.0	0.1287	2272.0	1.85×10^{-8}	1602.0	0.0166
HumanEval vs GPT-4-I	3055.0	0.0035	1305.0	0.0197	2507.5	4.79×10^{-6}	944.0	2.78×10^{-6}
HumanEval vs GPT-4-E	2047.5	7.14×10^{-7}	1875.5	0.0047	2308.0	8.50×10^{-10}	975.0	1.35×10^{-9}

TABLE IV
PYLINT RESULTS

Codebase	Error	Refactor	Warning	Convention
Human Eval	0	33	9	537
GPT-3.5-S	8	39	17	513
GPT-3.5-I	7	34	9	522
GPT-3.5-E	10	11	8	521
GPT-4-S	25	47	6	486
GPT-4-I	25	42	5	522
GPT-4-E	0	21	8	527

TABLE V
BANDIT RESULTS

Codebase	Bandit Issues Frequency
HumanEval	3
GPT-3.5-S	1
GPT-3.5-I	1
GPT-3.5-E	2
GPT-4-S	1
GPT-4-I	2
GPT-4-E	1

score, with all 164 test cases passing without any assertion or runtime errors, resulting in a pass rate of 100%, serving as a benchmark for evaluating the effectiveness of the language models. Among the model outputs, GPT-3.5 with the Simple prompt (GPT-3.5-S) successfully passed 117 test cases but encountered a significant number of assertion errors (36) and runtime errors (11), yielding a pass rate of 71.3%. Use of the Instructional prompt (GPT-3.5-I) demonstrated a decrease in performance, with only 107 test cases passed, and an increase in both assertion (44) and runtime errors (13), resulting in a lower pass rate of 65.2%, suggesting that the additional contextual guidance may have inadvertently complicated the model's ability to generate accurate solutions. Conversely, the Enhanced prompt version of GPT-3.5 (GPT-3.5-E) improved upon these results, with 124 test cases passed, 28 assertion errors, and 12 runtime errors, leading to a pass rate of 75.6%, indicating that providing both the test cases and canonical solutions positively influenced the model's performance. For the GPT-4 models, the Enhanced prompt (GPT-4-E) exhibited the strongest model performance, passing 143 test cases with only 17 assertion errors and 14 runtime errors, resulting in a pass rate of 82.2%. Overall, the GPT-4 model results indicate a clear trend where the models' performance improves with

TABLE VI
TEST CASE EXECUTION RESULTS

Codebase	Test Case Passed	Assertion Errors	Runtime Errors	Pass Rate
HumanEval	164	0	0	100.0
GPT-3.5-S	117	36	11	71.3
GPT-3.5-I	107	44	13	65.2
GPT-3.5-E	124	28	12	75.6
GPT-4-S	114	29	21	69.5
GPT-4-I	118	25	21	72.0
GPT-4-E	143	17	4	82.2

more detailed prompts.

6) Metrics Improvement over HumanEval: In Table VII, we present a comparative analysis of the percentage of code samples generated by GPT-3.5 and GPT-4 that not only pass the test case but also surpass the HumanEval benchmark on specific software quality metrics, namely Lines of Code (LOC), Cyclomatic Complexity, Vocabulary, Maintainability Index, and Cognitive Complexity.

Among the GPT-3.5 variants, GPT-3.5 Enhanced (E) shows the most substantial performance improvement, with 27.44% of its code samples achieving better LOC and 31.71% surpassing HumanEval in Vocabulary. Its Maintainability Index (41.46%) and Cognitive Complexity (29.27%) also indicate relative strength compared to other GPT-3.5 models, highlighting GPT-3.5 E's general effectiveness in generating syntactically efficient and conceptually concise code.

For GPT-4, our results indicate an overall marked improvement. GPT-4 Enhanced (E) achieves the highest performance, with 46.95% of its code samples having fewer LOC and 34.15% with lower Cyclomatic Complexity. Additionally, it outperforms in terms of Vocabulary (39.02%), Maintainability Index (53.66%), and Cognitive Complexity (45.73%), which suggests that GPT-4-E generates code that is not only functionally correct but also maintains simplicity and readability. The improvements in Maintainability Index and Cognitive Complexity are especially pronounced, suggesting that GPT-4-E emphasizes ease of code maintenance and reduced mental load for code comprehension. GPT-4 Instructional (I) and Simple (S) models also outperform their GPT-3.5 counterparts, albeit to a lesser extent than GPT-4-E.

In summary, GPT-4 E demonstrates superior performance across all evaluated metrics, indicating its ability to generate concise, complex, and maintainable code.

B. RQ1.1: Best prompt and GPT model combination

We apply the TOPSIS method to identify the codebase with the optimal quality metrics. To accurately assess model performance under distinct quality priorities, we conduct separate TOPSIS analyses for critical quality factors and for human-centered quality metrics. This approach allows us to identify which codebases rank highest when quality concerns center around critical factors such as correctness and security, and which codebases excel when evaluated on human-centric factors such as cognitive complexity, maintainability index, difficulty, lines of code (LOC), and refactoring opportunities, among others.

This two-fold evaluation provides a detailed ranking of codebases, covering HumanEval as well as those generated by different combinations of prompts and GPT models. The highest-ranking codebase is determined based on the maximum relative closeness value calculated for each codebase, representing the closest approximation to an ideal quality solution.

Tables VIII, IX, X list the empirical results of the TOPSIS analysis of various codebases with respect to human-centered quality factors. In Table VIII, we present the human-centric code quality metrics for each codebase averaged over the 164 Python code samples contained within the codebase. The LOC, CC, MI, N, η, Vol, D, E and T metrics are from Radon, the CogC metric is from Complexipy, and the R, W and C metrics are from Pylint. We note that the GPT-4 model when used with the enhanced prompt, to generate the GPT-4-E codebase, provides on average smaller codes with 6.4 LOC being the overall minimum among all codebases. This codebase also delivers code with the least cyclomatic complexity score, the smallest vocabulary size (η), and the lowest cognitive complexity (CogC) on average. We note that the GPT-3.5-Turbo model when used with the enhanced prompt, to generate the GPT-3.5-E codebase, provides, on average, code with the smallest density, and lowest values of effort, time to program, and refactorings. In Table IX, we normalize the quality metrics in Table VIII and also identify the best and worst quality criteria among all codebase alternatives depending on whether the objective is to minimize or maximize a quality criteria. For example, we take the minimum LOC as the ideal best value and the maximum LOC as the ideal worst value. Next, in Table X, we have calculated the Calculate the distances between the codebases' metric values and the ideal best and worst values. Finally, we calculate the relative closeness (RC) to the ideal solution. The codebase with the highest RC value is ranked at the top as the codebase with the best quality metrics. From the results in Table X, we see that using the GPT-3.5 model with our enhanced prompt ranks at the top, giving us the closest to ideal quality metrics for human-centric quality factors. We note that all GPT-4 generated codebases surpass the HumanEval code. GPT-3.5-I produces code of a similar quality to human-written code as its RC value of 0.52 is the same as HumanEval. Only GPT-3.5 with the simple prompt gives a lower quality code than human-written code as seen from the RC value of 0.13 which is lower than that of HumanEval.

Tables XI, XII, XIII list the empirical results of the TOPSIS analysis of various codebases with respect to critical code quality factors. In Table XI, we present the critical code quality metrics for each codebase averaged over the 164 Python code samples contained within the codebase. The Bugs metric is reported from Radon, Security metrics are reported from Bandit, the Errors and Warnings come from Pylint. The Tests Passed indicate the average number of test case suites passed by each code sample. We observe that, on average, each code sample in HumanEval successfully passes a full suite of test cases. However, the model-generated codebases have a lower average pass rate. Futhemore, while there are no compilation errors for human-written code and for the GPT-4-E codebase, the remaining codebases suffer from some compilation errors. Varying levels of bugs and warnings are also indicated across all codebases. In Table XII, we normalize the quality metrics in Table XI and also identify the best and worst quality criteria among all codebase alternatives. Next, in Table XIII, we have calculated the distances between the codebases' metric values and the ideal best and worst values, and the relative closeness (RC) to the ideal solution. We see that GPT-4-E and GPT-3.5-I surpass the HumanEval codebase in terms of critical code quality factors. GPT-3.5-E produces code of a similar quality to human-written code as its RC value of 0.61 is the same as HumanEval. Only GPT-4 with the simple prompt and instructional prompt give lower quality code than human-written code as seen from the RC values of 0.48 and 0.43 which is lower than that of HumanEval.

VI. Discussion and Implications

Our results indicate that LLMs specifically GPT-3.5-Turbo and GPT-4 are capable of producing better quality code than humans. Furthermore, we note that GPT-3.5-Turbo with the enhanced prompt produces the best overall code in terms of human-centric code quality metrics, whereas GPT-4 with the enahnced prompt produces the best overall code in terms of critical code quality metrics.

Our work establishes a foundation for using code quality metrics to enhance the effectiveness of code generation models in delivering secure, robust, and reliable code. By providing detailed assessments of code properties like security, robustness, and maintainability, our selected metrics can be leveraged to improve model training processes through fine-tuning on high-quality, trustworthy code samples. Furthermore, these metrics could enable the development of intelligent code classifiers capable of detecting safe, reliable, and trustworthy code, allowing for real-time validation and feedback within code generation pipelines. In future work, integrating these metrics into reinforcement learning frameworks for models could guide the generation process toward preferred quality attributes, while also serving as the basis for a feedback loop in which models self-correct based on quality-based reward signals.

TABLE VII
PERCENTAGE OF CODE SAMPLES PASSING TEST CASES AND PRODUCING BETTER METRICS THAN HUMANEVAL

Codebase	LOC	Cyclomatic Complexity	Vocabulary	Maintainability Index	Cognitive Complexity
GPT-3.5-S	3.66	20.73	20.12	54.27	17.68
GPT-3.5-I	22.56	20.73	22.56	28.05	21.34
GPT-3.5-E	27.44	19.51	31.71	41.46	29.27
GPT-4-S	19.51	16.46	31.10	39.63	24.39
GPT-4-I	30.49	22.56	28.66	37.20	29.27
GPT-4-E	46.95	34.15	39.02	53.66	45.73

TABLE VIII
CODE QUALITY METRICS OF HUMAN-WRITTEN AND GPT-GENERATED CODEBASES AVERAGED OVER 164 PYTHON CODE SAMPLES PER CODEBASE

Codebase	LOC	CC	MI	N	η	Vol	D	E	T	CogC	R	W	C
HumanEval	8.41	3.67	73.90	13.09	9.15	48.37	1.94	161.76	8.99	3.14	0.20	0.05	3.27
GPT-3.5-S	15.59	3.82	88.48	14.10	9.84	54.14	2.15	202.06	11.23	3.12	0.24	0.10	3.13
GPT-3.5-I	8.95	3.57	75.25	12.62	9.00	46.33	2.04	162.84	9.05	2.97	0.21	0.05	3.18
GPT-3.5-E	8.18	3.34	77.70	10.63	7.99	37.78	1.77	127.98	7.11	1.91	0.07	0.05	3.18
GPT-4-S	10.62	3.41	81.06	11.48	8.32	42.16	1.85	149.57	8.31	2.50	0.29	0.04	2.96
GPT-4-I	7.30	3.29	78.46	11.13	8.07	39.90	1.80	133.50	7.42	2.17	0.26	0.03	3.18
GPT-4-E	6.41	3.20	79.50	11.15	7.87	40.46	1.86	164.96	9.16	1.5	0.13	0.05	3.21

TABLE IX
IDEAL BEST AND IDEAL WORST CODE QUALITY METRICS OF CODEBASES AVERAGED AND NORMALIZED OVER 164 PYTHON CODE SAMPLES PER CODEBASE

Codebase	LOC	CC	MI	N	η	Vol	D	E	T	CogC	R	W	C
HumanEval	0.33	0.40	0.35	0.41	0.40	0.41	0.38	0.38	0.38	0.47	0.36	0.36	0.39
GPT-3.5-S	0.60	0.41	0.42	0.44	0.43	0.46	0.42	0.48	0.48	0.46	0.43	0.67	0.37
GPT-3.5-I	0.35	0.39	0.36	0.39	0.39	0.39	0.40	0.39	0.39	0.44	0.37	0.36	0.38
GPT-3.5-E	0.32	0.36	0.37	0.33	0.35	0.32	0.35	0.30	0.30	0.28	0.12	0.32	0.38
GPT-4-S	0.41	0.37	0.39	0.36	0.36	0.36	0.36	0.36	0.36	0.37	0.52	0.24	0.35
GPT-4-I	0.28	0.36	0.37	0.35	0.35	0.34	0.35	0.32	0.32	0.32	0.46	0.20	0.38
GPT-4-E	0.25	0.35	0.38	0.35	0.34	0.34	0.37	0.39	0.39	0.22	0.23	0.32	0.38
Ideal Best	0.25	0.35	0.42	0.33	0.34	0.32	0.35	0.30	0.30	0.22	0.12	0.20	0.35
Ideal Worst	0.60	0.41	0.35	0.44	0.43	0.46	0.42	0.48	0.48	0.47	0.52	0.67	0.39

TABLE X
CODEBASE RANKINGS FROM TOPSIS ANALYSIS OF HUMAN-CENTRIC CODE QUALITY METRICS (RC: RELATIVE CLOSENESS, D: DISTANCE)

Rank	Codebase	D_{ideal}	$D_{anti-ideal}$	RC
1	GPT-3.5-E	0.16	0.71	**0.82**
2	GPT-4-E	0.21	0.67	0.76
3	GPT-4-I	0.36	0.67	0.65
4	GPT-4-S	0.46	0.54	0.54
5	HumanEval	0.43	0.47	0.52
6	GPT-3.5-I	0.42	0.46	0.52
7	GPT-3.5-S	0.78	0.11	0.13

TABLE XI
CRITICAL CODE QUALITY METRICS OF HUMAN-WRITTEN AND GPT-GENERATED CODEBASES AVERAGED OVER 164 PYTHON CODE SAMPLES PER CODEBASE

Codebase	Bugs	Tests Passed	Secur.	Err.	Warn.
HumanEval	0.02	1.00	0.02	0.00	0.05
GPT-3.5-S	0.02	0.71	0.01	0.05	0.10
GPT-3.5-I	0.02	0.65	0.01	0.04	0.05
GPT-3.5-E	0.01	0.76	0.01	0.06	0.05
GPT-4-S	0.01	0.70	0.01	0.15	0.04
GPT-4-I	0.01	0.72	0.01	0.15	0.03
GPT-4-E	0.01	0.87	0.01	0.00	0.05

VII. THREATS TO VALIDITY

In this study, certain validity threats may impact the generalizability of our findings.

A. Internal Validity

An internal threat to validity arises from potential data leakage, where the inclusion of human-written code in the training dataset may influence the performance of the GPT models, resulting in biased outcomes.

B. Construct Validity

Construct validity is the degree to which our study accurately captures the intended constructs. The definition of "ideal best" and "ideal worst" values for each metric poses a con-

TABLE XII
BEST AND WORST CRITICAL CODE QUALITY METRIC AVERAGES OF HUMAN-WRITTEN AND GPT-GENERATED CODEBASES AVERAGED AND NORMALIZED OVER 164 PYTHON CODE SAMPLES PER CODEBASE

Codebase	Bugs	Tests Passed	Secur.	Err.	Warn.
HumanEval	0.41	0.48	0.65	0.00	0.36
GPT-3.5-S	0.46	0.35	0.22	0.21	0.67
GPT-3.5-I	0.39	0.32	0.22	0.18	0.36
GPT-3.5-E	0.32	0.37	0.44	0.26	0.32
GPT-4-S	0.36	0.34	0.22	0.65	0.24
GPT-4-I	0.34	0.35	0.44	0.65	0.20
GPT-4-E	0.34	0.42	0.22	0.00	0.32
Ideal Best	0.32	0.48	0.22	0.00	0.20
Ideal Worst	0.46	0.32	0.65	0.65	0.67

TABLE XIII
CODEBASE RANKINGS FROM TOPSIS ANALYSIS OF CRITICAL CODE QUALITY METRICS (RC: RELATIVE CLOSENESS, D: DISTANCE)

Rank	Codebase	D_{ideal}	$D_{anti-ideal}$	RC
1	GPT-4-E	0.14	0.88	**0.87**
2	GPT-3.5-I	0.30	0.72	0.70
3	HumanEval	0.47	0.75	0.61
4	GPT-3.5-E	0.38	0.59	0.61
5	GPT-3.5-S	0.55	0.62	0.53
6	GPT-4-S	0.67	0.62	0.48
7	GPT-4-I	0.70	0.54	0.43

struct validity risk, potentially leading to misrepresentations of quality standards or user expectations. In our existing setup, selecting the ideal best and worst values directly from the max/min values of each criterion helps reduce certain biases, particularly those associated with arbitrary or subjective assignment. This approach leverages the actual range of values within our dataset, making the ideal best and worst more representative of the dataset's distribution. However, it doesn't inherently guarantee that these extreme values represent the true quality ideals relevant to practitioners or theoretical quality standards. Our study acknowledges the risk that the selected metrics may not capture all dimensions of code quality comprehensively. However, we mitigate this threat by including a broad range of widely recognized, standard quality metrics that represent key facets of software quality.

A potential threat to construct validity arises from the limitations of our selected tools and code quality metrics. Code metrics, while informative, may not always be reliable or fully aligned with developers' perceptions of code quality [24]. Moreover, even when these metrics are considered reliable, the tools that implement them may introduce additional sources of error or inconsistency. A notable example is Pylint, where a study highlights its failure to correctly check the usage of imported dependencies [25]. Directly measurable metrics such as Lines of Code (LOC) and Cyclomatic Complexity (CC) reported by Radon, security issues reported by Bandit, and Pylint errors, warnings, refactoring suggestions, and convention violations are widely regarded as reliable indicators of code quality. These metrics assess tangible properties of the code, providing consistent and objective measurements that align with established software engineering practices. As such, they ensure construct validity by capturing fundamental aspects of code structure, security, and adherence to coding standards. Cognitive Complexity (CogC) has been validated in prior research [26] as an effective code-based measure of understandability, thereby satisfying construct validity by accurately capturing aspects of code readability and comprehension. However, the Maintainability Index (MI), a composite metric incorporating cyclomatic complexity, Halstead volume, and LOC, has been subject to scrutiny regarding its reliability. As discussed in an online article [27], MI may not accurately reflect the maintainability of modern codebases, as it was originally derived from studies of older languages like C and Pascal. Furthermore, MI's formula and coefficients have not been updated to account for contemporary programming practices. In contrast, Halstead's complexity metrics, as well as MI, have shown reliability in older languages [28]. Given the reviews on the reliability of the Maintainability Index (MI), we must acknowledge its threat to the construct validity of our evaluation.

C. External Validity

External validity is a concern since our study is limited to the Python programming language, our results may not generalize to other programming languages.

VIII. RELATED WORK

In this section, we review key works related to the evaluation of code quality and trustworthiness, particularly in the context of code generated by large language models (LLMs).

The most recent and relevant work [6] investigates whether ChatGPT generates Python and Java code with quality issues. This work leverages DevGPT [29], a curated dataset of developer-ChatGPT conversations consisting of prompts with ChatGPT's responses, including code snippets. In contrast, our work makes a comparison between the quality of human-written code against model-generated code in terms of both correctness and broader quality factors. While their work highlights the quality issues of ChatGPT, our work shows that with instructional and enhanced prompting, GPT generated code can surpass human-written code in quality. Another recent work investigates whether GitHub Copilot is as bad as humans in generating insecure code [30]. This investigation is performed using a dataset of C/C++ vulnerabilities, and the findings indicate that Copilot is not as bad as human developers at introducing vulnerabilities in code. In comparison to this work, our study uses GPT models to compare the quality of Python code and focuses on broader quality factors. Our findings that code models are capable of writing better quality code than humans, align with the findings of this work. Another work [31] compares the effectiveness of ChatGPT and Co-Pilot for generating quality Python code solutions for LeetCode [32] problems, however, their work does not make a comparison against human-written code.

A recent study evaluates eight popular language models for program generation across five key datasets, using explainable

AI techniques to identify significant code tokens influencing transformations [33]. Our work serves as a valuable addition to their goals of enhancing model reliability within a framework that assesses not only correctness but also quality. The code metrics we calculated for HumanEval can establish a benchmark for evaluating the quality of future models.

Existing code generation models are typically evaluated with metrics like accuracy, BLEU, METEOR, ROUGE-L, chrF, CodeBLEU, and RUBY [34–39], which primarily assess correctness by comparing generated code to a reference sample. However, these metrics are limited to measuring similarity or token-level accuracy. In contrast, our approach focuses on evaluating code quality in code generation models through code quality metrics, providing a more comprehensive view that extends beyond mere correctness.

A recent work involves metrics calculation to improve the security of large language model generated code [40]. This work introduces the LLMSecGuard framework designed to enhance code security by combining the capabilities of static code analyzers with LLMs, providing developers with more secure alternatives to standard LLM-generated code. Another work improves the bug prediction using metrics [41]. This study introduces a method to predict diverse performance bug types using historical data like source code and change history, applying performance-specific code metrics. While this work targets performance issues specifically, our metrics extend to various quality dimensions such as security, maintainability, and reliability, making them adaptable to broader quality assessment goals. Future applications of our metrics could involve training classifiers to detect not only bugs but also overall code trustworthiness and adherence to quality standards, supporting safer and more robust code outputs from automated code generation systems.

Similar to how these works aim to improve model performance using metrics, our work lays the foundation of using the code quality metrics for improving the code generation performance of models through fine-tuning on high-quality, trustworthy code samples. Furthermore, these metrics could enable the development of intelligent code classifiers capable of detecting safe, reliable, and trustworthy code.

IX. Conclusions and Future Work

In conclusion, this study presents a comprehensive analysis of code quality across human-written and LLM-generated code, evaluating key quality factors including correctness, security, maintainability, and readability among others. We explored the code quality of large language models (LLMs) with a focus on GPT models, evaluating their performance against human-written code using various code quality metrics. Our results indicate that while GPT models can generate code with higher internal quality in some cases, they do not consistently produce correct or trustworthy code. In future work, it would be beneficial to conduct human validation to assess whether the metrics used in this study genuinely serve as reliable indicators of code quality, particularly in the context of code generated by LLMs. We also plan to extend this work by enhancing LLMs for the critical quality aspects through targeted fine-tuning and the incorporation of quality metrics. Other future plans include exploring dynamic prompt engineering strategies that adapt to specific quality requirements and providing further insights into optimizing LLM-generated code for production-level use.

References

[1] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[5] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[6] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 152–156, 2024.

[7] Nicole Davila, Igor Wiese, Igor Steinmacher, Lucas Lucio da Silva, André Kawamoto, Gilson José Peres Favaro, and Ingrid Nunes. An Industry Case Study on Adoption of AI-based Programming Assistants. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 92–102, 2024.

[8] Peter Demeter and Pavle Dakic. Visualization of code metrics for code quality and assessment of breach of standards. In *2024 14th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 654–659, 2024.

[9] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, 28(2):30, 2023.

[10] Siyuan Jin, Ziyuan Li, Bichao Chen, Bing Zhu, and Yong Xia. Software code quality measurement: Implications from metric distributions. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 488–496, 2023.

[11] Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler, and Muhammad Usman. A tertiary study on links between source code metrics and external quality attributes. *Information and Software Technology*, 165:107348, 2024.

[12] GPT-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo, Sep. 2024.

[13] GPT-4. https://openai.com/index/gpt-4/, Sep. 2024.

[14] Radon. https://pypi.org/project/radon/, March 2023.

[15] Bandit. https://github.com/PyCQA/bandit, September 2023.

[16] Pylint. https://pypi.org/project/pylint/, July 2024.

[17] Complexipy. https://github.com/rohaquinlop/complexipy, June 2021.

[18] Gwo-Hshiung Tzeng and Jih-Jeng Huang. *Multiple attribute decision making: methods and applications*. CRC press, 2011.

[19] Can LLMs Generate Higher Quality Code Than Humans? An Empirical Study: Code and Dataset. https://github.com/shamsa-abid/PythonCodeMetricsCalculator, Nov. 2024.

[20] Cruz Izu and Claudio Mirolo. Towards Comprehensive Assessment of Code Quality at CS1-Level: Tools, Rubrics and Refactoring Rules. In *2024 IEEE Global Engineering Education Conference (EDUCON)*, pages 1–10, 2024.

[21] Suryadiputra Liawatimena, Harco Leslie Hendric Spits Warnars, Agung Trisetyarso, Edi Abdurahman, Benfano Soewito, Antoni Wibowo, Ford Lumban Gaol, and Bahtiar Saleh Abbas. Django Web Framework Software Metrics Measurement Using Radon and Pylint. In *2018 In-*

donesian Association for Pattern Recognition International Conference (INAPR), pages 218–222. IEEE, 2018.

[22] Divya K Konoor, Rajyalakshmi Marathu, and Prashanth Reddy. Secure OpenStack Cloud with Bandit. In *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 178–181. IEEE, 2016.

[23] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

[24] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91, 2018.

[25] Bart van Oort, Luís Cruz, Maurício Aniche, and Arie van Deursen. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pages 1–8, 2021.

[26] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 1–12, 2020.

[27] Arie van Deursen. Think Twice Before Using the "Maintainability Index". https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/, 2014.

[28] Zoltán Tóth. Applying and Evaluating Halstead's Complexity Metrics and Maintainability Index for RPG. In *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V 17*, pages 575–590. Springer, 2017.

[29] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. DevGPT: Studying Developer-ChatGPT Conversations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 227–230. IEEE, 2024.

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

[31] Nikolaos Nikolaidis, Karolos Flamos, Khanak Gulati, Daniel Feitosa, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. A Comparison of the Effectiveness of ChatGPT and Co-Pilot for Generating Quality Python Code Solutions. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 93–101, 2024.

[32] Leetcode. https:/https://leetcode.com/, Oct. 2023.

[33] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. On the Reliability and Explainability of Language Models for Program Generation. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26, 2024.

[34] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[35] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[36] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[37] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does BLEU score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE, 2019.

[38] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[39] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395, 2015.

[40] Arya Kavian, Mohammad Mehdi Pourhashem Kallehbasti, Sajjad Kazemi, Ehsan Firouzi, and Mohammad Ghafari. LLM security guard for code. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 600–603, 2024.

[41] Guoliang Zhao, Stefanos Georgiou, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. Enhancing performance bug prediction using performance code metrics. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 50–62, 2024.