Evaluating the Trustworthiness of Large Language Models for Code Generation

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

ABSTRACT

The rapid deployment of Large Language Models (LLMs) like OpenAI's ChatGPT across various sectors has underscored the urgent need for robust verification and validation techniques. These models have a vast comprehension of both linguistic nuance as well as any tasks contained within their training dataset. The ability of LLMs to be proficient in a variety of tasks makes them desirable, but there is a critical need to verify and validate these models due to their questionable reliability and widespread use. This paper focuses on the verification and validation of models that produce source code to accomplish the task instructed by a prompt provided by the user.

One of the key challenges of LLMs is how to verify and validate the output of these models. Modern LLMs rely on deep neural networks to transform the input prompt into an output. It is nearly impossible to explain the output of a sufficiently complex neural network due to its complex, deep architectures with billions of parameters and non-linear operations across high-dimensional spaces. This lack of explainability requires the use of vigorous verification and validation methods to ensure confidence in the models.

This paper will present the results of evaluating the code generation capabilities of the last-generation and current state-of-the-art LLMs to see how they compare against human programmers at a variety of skill levels. Both the models and human programmers will be evaluated by a set of hidden unit tests, face validation by a group of subject matter experts, and an adversarial machine learning model trained to recognize machine-generated programs. Each of these methods will be used to gauge the current capabilities of LLMs used to generate source code and to demonstrate their use in performing verification and validation of LLMs in general.

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.

Evaluating the Trustworthiness of Large Language Models for Code Generation

E. Michael Bearss

Trideum Corporation

Huntsville, AL

mbearss@trideum.com

INTRODUCTION

In the fast-evolving field of artificial intelligence, large language models (LLMs) like ChatGPT have demonstrated remarkable abilities to perform creative tasks. These models can generate text, images, and even videos from prompts that are almost indistinguishable from human-generated ones. These remarkable capabilities have spurred their rapid adoption across various sectors. This rapid adoption, coupled with the inherent lack of explainability of these models, underscores a grave need to establish trust in them.

One task for which LLMs are particularly well suited, and the topic of this paper, is generating computer code. This capability has opened new pathways for automating software development processes, significantly enhancing productivity and potentially reducing the human effort required in coding. However, the automated generation of code also introduces complexities concerning reliability, safety, and security. This necessitates rigorous verification and validation (V&V) to ensure that the produced code not only functions as intended but also adheres to stringent quality standards.

The challenge of V&V for code generated by LLMs is twofold. First, the inherently probabilistic nature of LLMs can lead to unpredictable variations in code output, which may not consistently align with best coding practices or meet specific functional requirements. Second, the opacity of these models, often referred to as the *black-box* phenomenon, complicates the understanding of how decisions are made, thus making traditional debugging and validation techniques less effective. This paper addresses these challenges by providing a multi-method approach to performing V&V of LLMs that are able to generate computer code. Each of these methods seeks to verify a different aspect of the code generated by the LLMs. These methods include:

- A set of unit tests assessing the technical correctness of the generated code
- A survey comparing the correctness and readability of generated code to those developed by humans
- A machine learning (ML) model trained to detect code generated by an LLM

The first two methods aim to directly evaluate the quality of the code generated by the LLM. The third method offers an indirect approach to determine whether the generated code contains any distinct patterns not present in traditional code samples. This paper focuses on assessing the capabilities of current state-of-the-art LLMs to generate accurate and robust code for software development tasks, providing a thorough evaluation of their performance and reliability. Many programs in the modeling and simulation community are currently considering integrating LLMs into their software development efforts. This research offers valuable insights into the advantages and disadvantages of utilizing LLMs in these contexts, aiding stakeholders in making informed decisions about their adoption and implementation.

BACKGROUND

This section provides background information on a variety of topics important to the rest of this paper. This section begins by providing a brief explanation of LLMs, followed by a description of various methods of V&V useful for evaluating LLMs. Finally, a detailed description of Krippendorff's alpha is provided, which is a useful metric for measuring statistical validity of survey results.

Large Language Models

LLMs such as GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers) have revolutionized the field of natural language processing (NLP). These models are based on the transformer architecture introduced by Vaswani et al. (2023). At the core of this architecture is the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence, regardless of their positional distance. This design fundamentally changed how machines understand human language by enabling the model to efficiently handle words and phrases far apart in text while maintaining an understanding of their contextual relationships.

The architecture of an LLM consists of connected transformer blocks. Each block contains layers that perform specific tasks. These layers are primarily grouped into two categories. Self-attention layers help the model focus on different words when it processes a sentence (e.g., "The ice is slippery," it might focus on "ice" and "slippery" to understand the connection between them). Feed-forward layers process each word's information from the self-attention layer to help the model understand or generate text.

LLMs can consist of two components: an encoder and a decoder. The encoder reads and processes the input text to create a contextualized representation that captures the relationships and meanings within the text. The decoder uses the context the encoder provides to generate output text, predicting one word at a time based on the input it received and what it has generated so far. These two components can be combined or used separately to perform different functions. An illustration of this architecture is shown in Figure 1.

- Decoder-Only models (e.g., GPT) predict the next word in a sentence one at a time, making them excellent at generating text that flows naturally.
- Encoder-Only models: (e.g., BERT) are exceptionally good at understanding the context of text, which is useful for tasks like answering questions or identifying sentiments in reviews.
- Encoder-Decoder models, with separate parts for encoding (reading and understanding text) and decoding (generating text), are especially useful for translating from one language to another.

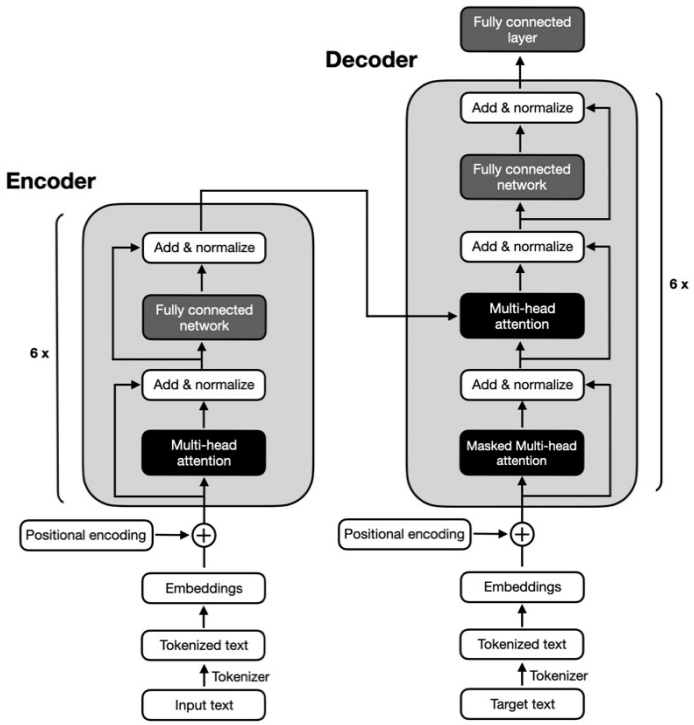

Figure 1. Illustration of the Original Transformer Architecture (Vaswani et al., 2023)

LLMs are typically pre-trained on a large corpus of text in an unsupervised manner. This pre-training involves learning a general language representation by predicting words in a sentence given their context (masked language modeling) or by generating the next word in a sequence (autoregressive language modeling). After pre-training, these models are fine-tuned on specific tasks with labeled data, which allows them to achieve state-of-the-art performance across a wide range of NLP tasks. Some models, especially more recent versions of GPT, have shown remarkable ability to perform tasks with little to no task-specific training (Brown et al., 2020). This is achieved through sophisticated training regimes and leveraging the vast amount of data they were initially trained on.

Verification and Validation of LLMs

The increased deployment of LLMs in real-world applications has amplified the need for robust V&V methods to ensure their reliability, safety, and ethical use. As LLMs are integrated into critical domains such as healthcare, finance, legal systems, and the defense sector, the stakes for their accurate and fair operation continue to increase. V&V consists of two independent processes often used together to ensure that products meet both their specified requirements (verification) and their intended purpose and user needs (validation). Verification involves ensuring that the models function correctly according to their specifications, which includes checking for errors in the model's architecture, algorithms, and implementation. On the other hand, validation assesses their performance in intended operational contexts, ensuring that the models deliver expected outcomes in real-world scenarios and meet user requirements (Petty, 2013).

Key challenges in V&V for LLMs arise from their complex behaviors and the difficulty in predicting their outputs across varied inputs. These challenges are further compounded by issues such as hallucinations, where the models generate factually incorrect or nonsensical information, and bias, where the models exhibit discriminatory or harmful outputs (Bender et al., 2021). For instance, an LLM used in a legal advisory system might incorrectly interpret case law or provide biased advice based on the training data's inherent biases, leading to unjust outcomes. Another significant challenge is the dynamic nature of language and the evolving contexts in which these models operate, which can result in outdated or contextually irrelevant responses. Furthermore, the black-box nature of these models complicates efforts to audit and understand their decision-making processes, making it difficult to diagnose and rectify errors or biases.

To address these challenges, researchers and practitioners have developed various V&V approaches, including:

1. **Adversarial Testing**: This technique involves exposing models to adversarial examples designed to test their robustness. This method helps uncover vulnerabilities where the models might produce incorrect or biased outputs (Jia & Liang, 2017).
2. **Formal Verification**: Formal methods provide mathematical guarantees about the behavior of models. These methods, while promising, face scalability issues due to the size of LLMs (Ribeiro et al., 2018), (Gopinath, Katz, Pasareanu, & Barrett, 2020).
3. **Human-in-the-Loop Evaluation**: Incorporating human judgment in the evaluation process helps address issues like hallucinations and bias. This approach leverages domain expertise to validate model outputs in context-specific scenarios (Bartolo, Roberts, Welbl, Riedel, & Stenetorp, 2020).

By understanding the underlying architectures, training processes, and the inherent challenges of LLMs, researchers and industry professionals can better use the V&V process to establish confidence in these models. This paper presents the results of using several of these V&V techniques to evaluate the ability of current state-of-the-art LLMs in code generation tasks.

Statistical Measures of Agreement

Inter-rater reliability measures the agreement between coders. Krippendorff's alpha (α), developed by Klaus Krippendorff in 1969, is a popular method for this purpose (Krippendorff, 2011b). Krippendorff's Alpha is versatile, accommodating any number of coders, values, and handling incomplete, unequal, or missing data. It is applicable to nominal, ordinal, interval, and ratio data.

For m coders assigning v values to n units of analysis, an $m \times n$ reliability matrix is created. Each element v_{ij} represents the value coder c_j assigned to unit u_j. When incomplete data is present, m_j, the number of values assigned

to unit *j*, may be less than *m*. To construct a coincidence matrix, all values must be pairable. This requires $m_j >= 2$. An example matrix with reliability data for three coders is shown in Table 1. Each row represents a single coder labeled A, B, and C. Each column represents a unit of analysis labeled 1, 2, 3, and 4. Missing data is shown as a *.

Table 1. Example Reliability Data for 3 Coders

Units	u	1	2	3	4
Coders	A	1	1	2	1
	B	2	*	2	3
	C	1	1	2	3

To calculate α, first, a coincidence matrix is generated. Each element in the coincidence matrix, o_{ck}, is given by equation 1:

$$o_{ck} = \sum_{u} \frac{Number\ of\ ck\ pairs\ in\ unit\ u}{m_u - 1} \quad (1)$$

Table 2. Coincidence Matrix Calculation

v	1	2	3	n_v
1	$\frac{2}{2}+\frac{2}{1}+0+0=3$	$\frac{2}{2}+0+0+0=1$	$0+0+0+\frac{2}{2}=1$	5
2	1	$0+0+\frac{6}{2}+0=3$	$0+0+0+0=0$	4
3	1	0	$0+0+0+\frac{2}{2}=1$	2
Frequency n_v	5	4	2	11

Table 2 shows the calculation used to derive each value of the coincidence matrix. For nominal (categorial and unordered) data, equation 2 can be used to calculate α using the coincidence matrix from Table 2.

$$a = 1 - \frac{D_O}{D_e} = \frac{A_O - A_e}{1 - A_e} = \frac{(n-1)\sum_c o_{cc} - \sum_c n_c(n_c - 1)}{n(n-1) - \sum_c n_c(n_c - 1)}$$

$$a = \frac{(11-1)(3+3+1) - [5(5-1) + 4(4-1) + 2(2-1)]}{11(11-1) - [5(5-1) + 4(4-1) + 2(2-1)]} = \frac{9}{19} = 0.4737 \quad (2)$$

When observers are in perfect agreement, the observed disagreement, $D_O = 0$, and $\alpha = 1$. When results are purely by chance, $D_O = D_e$ and $\alpha = 0$. For this example, $\alpha = 0.4737$, indicating some agreement but not statistically better than chance. Values of $\alpha \geq 0.8$ indicate strong agreement, while $0.8 > \alpha \geq 0.667$ indicate tentative conclusions (Krippendorff, 2011a).

DEVELOPING TEST CASES

Due to the complexity of the LLMs used to generate code samples, static analysis of the models is nearly impossible. As an alternative, the models can be used to generate samples, and both qualitative and quantitative metrics can be used to assess these output samples.

Problem Selection

To effectively evaluate the ability of LLMs to produce code to solve problems, several coding challenge problems were created. In this case, problem selection was of critical importance to ensure effective testing. Selected problems were modeled after questions from sites such as leetcode.com, a common interview preparation site used by software developers. However, questions were not directly taken from these sites due to their solutions being readily available on various sites as well as in the corpus of data commonly used to train LLMs.

Two programming challenge problems were created for this work. These problems were used as prompts for an LLM to assess its ability to produce code to solve the problem. In addition, a survey was created to challenge software developers of various skill levels to solve these same problems. This data was used to compare the abilities of an LLM with those of a human software developer.

When developing the coding assessment, careful attention was given to several key criteria to ensure the collection of sufficient high-quality samples. To encourage respondents to complete the assessment, it was essential that the problems were small and self-contained, solvable in approximately 30 minutes with limited research, and did not require specific domain knowledge. Additionally, because the survey was unproctored, the problems were designed such that sub-problems might have online solutions, but the complete solutions would not be readily available. This approach also ensured that the solutions would not be part of the corpus of data used to train the LLMs, necessitating the generation of new code to solve the sample problems.

Ideally, the solutions generated by LLMs and survey respondents would vary in their level of correctness. Similar to school assignments graded with a rubric, solutions can receive partial credit for incomplete responses. Novice programmers might solve individual sub-problems but struggle with integration. Mid-level programmers may develop correct but inefficient solutions, while experienced programmers are likely to produce robust and efficient solutions. Additionally, creating problems with well-defined test cases offers several benefits. Concrete test cases aid respondents in crafting their solutions and significantly simplify the evaluation process.

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Prime numbers are natural numbers that are divisible by only 1 and the number itself. In other words, prime numbers are positive integers greater than 1 with exactly two factors. For example:

2, 3, 5, 7, 11, 13, ...

A Fibonacci prime is a number that is both a Fibonacci number and a prime. For example:

2, 3, 5, 13, 89, ...

Create a function that takes a single argument ***n*** and returns the ***n***th number in the sequence of Fibonacci primes.

Figure 2. *n*th Fibonacci Prime Sample Problem

Two problems were developed for this research. The first, called *nth Fibonacci Prime* (shown in Figure 2), asks the developer to create a function that returns the n^{th} number that is both a member of the Fibonacci sequence and a prime number. This problem is a combination of two common component problems that must be integrated. Although seeming easy initially, the size of the Fibonacci prime numbers grows very quickly, making it challenging to find a solution beyond $n = 10$. The problem is designed in a way such that both subproblems must be solved simultaneously. One cannot compute all the prime numbers and then move on to computing all Fibonacci numbers separately.

The Collatz conjecture is one of the most famous unsolved problems in mathematics. The conjecture asks whether repeating two simple arithmetic operations (shown below) will eventually transform every positive integer into 1.

Let an integer's Collatz number (C) be the number of operations required to transform it into 1.

Create a function that takes a single argument ***n*** (1 <= ***n*** <= 60) as a parameter and returns the size of the set of integers with that Collatz number.

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2, \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

Figure 3. Collatz Numbers Sample Problem

The second problem, *Collatz Numbers* (shown in Figure 3), asks the developer to create a function that returns the size of the set of numbers with a particular Collatz number. This problem is significantly more complex than the first. While many solutions are available to compute the Collatz number of a particular input, there are few examples of computing the set of numbers with a particular Collatz number. This problem is designed to require the programmer to invert the Collatz function to calculate all possible numbers for a particular input rather than brute-force checking each number, as large numbers can have a relatively small Collatz number.

Generating the code samples

Two different LLMs were used to generate the code samples to solve the two sample problems. Two versions of ChatGPT by OpenAI were used. ChatGPT is a general-purpose generative AI that can perform a wide variety of tasks, including code generation. ChatGPT 3.5 was used to represent a current-generation model, while ChatGPT 4 was used as a next-generation model. ChatGPT 3.5 is currently free to use, while version 4 is part of a paid plan with limited amounts of queries per time period.

The LLM prompts were used as they were written to generate the samples. To help the LLMs produce accurate results, the user was allowed to interact with the model multiple times. These interactions involved providing troubleshooting and debugging assistance to the model, simulating how an expert coder would use the models to speed up development tasks. Troubleshooting assistance was given to the model until it appeared stuck at an incorrect solution. Although the programming language was not specified, the LLM consistently chose Python. The reason for this is unknown and could be due to dataset bias or the interaction history of the user querying the model. This language preference is not an issue, as ChatGPT (and other LLMs) are particularly good at translating one programming language to another.

With the first problem, *n^{th} Fibonacci Prime*, the two LLMs showed a stark contrast in their problem-solving abilities. Using ChatGPT 3.5, the first attempt seemed promising, but the solution was incorrect as it skipped the first three Fibonacci primes. The second solution precomputed these numbers and was correct; however, it used an inefficient method to calculate prime numbers and included a peculiar approach for calculating Fibonacci numbers. The third solution significantly accelerated the prime computation but was incorrect due to repeating the number 5. The fourth and final solution was correct, yet it still relied on an inefficient method for prime calculation, which was only effective for $n \leq 9$.

ChatGPT 4 showed a drastic improvement in its understanding of this problem. With only the initial prompt and no need for troubleshooting, the model generated a correct solution. Additionally, it produced a more efficient solution that matched the best solution created by a human programmer. Four of the five human solutions only worked for $n \leq 10$ within a reasonable amount of time, while the solution by the LLM could solve for $n \leq 12$. An excerpt from the solution is shown in Figure 4.

```
15  def fibonacci_prime(n):
16      fibs = [0, 1]
17      fib_primes = []
18      while len(fib_primes) < n:
19          next_fib = fibs[-1] + fibs[-2]
20          fibs.append(next_fib)
21          if is_prime(next_fib):
22              fib_primes.append(next_fib)
23      return fib_primes[n-1]
```

Figure 4. Fibonacci Prime Code Excerpt from ChatGPT 4

The second problem, *Collatz Numbers*, proved significantly more challenging for both LLMs. The first attempt generated by ChatGPT 3.5 was incorrect due to it misunderstanding the prompt, checking only the numbers $1 \leq n \leq 60$ for their corresponding Collatz numbers. The second attempt fixed this problem but limited the numbers tested to only 1000, causing almost all test cases to fail. Despite feedback that all numbers needed to be tested, the LLM reverted to checking only $1 \leq n \leq 60$ for their Collatz numbers. After a fourth round of feedback, including an

example test case, the solution was updated to be technically correct but would take too long to run, as it changed the upper bound of numbers to test to 2^{31} (see Figure 5).

```
def collatz_steps(num):
    steps = 0
    while num != 1:
        if num % 2 == 0:
            num //= 2
        else:
            num = 3 * num + 1
        steps += 1
    return steps

count = 0
for i in range(1, 2 ** 31):
    if collatz_steps(i) == n:
        count += 1
return count
```

Figure 5. Collatz Number Code Excerpt from ChatGPT 3.5

Using ChatGPT 4, the initial solution misinterpreted the prompt and generated the Collatz number for all numbers between 0 and n. After receiving feedback, the second attempt followed a similar structure to the first one, correctly examining the Collatz numbers up to a *reasonable* upper bound of 100,000. However, this approach was inaccurate for many test cases. In response to feedback, the third solution attempted to invert the Collatz function to find numbers with a specific Collatz number, but this approach also proved incorrect.

Collecting Human Solutions (Survey 1)

A survey was conducted to gather coding solutions from software engineers in order to establish a benchmark for comparing generative AI solutions. The survey was created using Google Forms and allowed for online submissions. Participation in the survey was entirely optional, and respondents were not required to reveal their identities. They were also not compensated for their participation. The survey aimed to collect solutions to the two coding prompts from software engineers with varying levels of experience. To ensure the quality of the samples, the survey was only distributed to known contacts with programming backgrounds.

The participants were given 30 minutes to complete programming tasks, but internet usage or time was not supervised. Five responses were collected: two in C# and three in Python. The respondents had varying levels of experience, with three having between three and five years of experience and two having over ten years. Their educational backgrounds were also diverse, including one with some college experience, one with an associate's degree, two with bachelor's degrees, and one with a master's degree. None of the respondents mentioned regular participation in coding challenges.

Multiple survey respondents provided feedback, indicating a common theme that the coding problems mainly centered on mathematical and looping tasks. They expressed that these problems might only partially capture the creative thinking and practical problem-solving required in real-world programming. They suggested including problems that more accurately simulate actual coding scenarios involving interactions with servers, programs, databases, and hardware. Additionally, some respondents noted that the absence of a testing environment on their personal computers affected their performance.

RESULTS

This section presents the findings of this study, with each subsection focusing on a different method of V&V described previously.

Technical Correctness

Due to careful consideration during problem development, evaluating the technical correctness of these two programming problems was simple. Each function has a well-defined range of inputs and easily verifiable outputs. There is only one element of subjectivity in evaluating the produced code: how efficient the solution is in computing the output.

To evaluate correctness, a table of valid outputs was generated for comparison. A partial table of these outputs was provided to both the human software engineers and LLMs to assist in their development and troubleshooting. For the Fibonacci prime numbers, a table from the online encyclopedia of integer sequences (OEIS) was used. This source provides solutions for $n \leq 15$. For the *Collatz numbers* problem, the author created a solution to the prompt and verified the provided test cases by inspection.

Table 3. Solution Scorecard

	nth Fibonacci Prime		Collatz Numbers	
	Correctness	Efficiency	Correctness	Efficiency
ChatGPT 3.5	1	0	1	0
ChatGPT 4	1	2	0	0
Human (3-5 years)	1	1	1	2
Human (3-5 years)	1	1	0	0
Human (10+ years)	1	1	1	0
Human (10+ years)	1	1	1	0
Human (3-5 years)	1	2	1	0

Table 3 shows a scorecard for the solutions developed by both the human software engineers and LLMs. The best scoring solution for each of the LLMs was considered for this score. The Correctness score for a solution is 0 if it returns any incorrect values and 1 if it produces the correct results in any amount of time, even if it takes years to compute. The Efficiency score ranges from 0 to 2, where 0 represents an infeasible amount of execution time (years). The values 1 to 2 were chosen subjectively, with 2 representing the optimal known solution.

Both LLMs and all human coders successfully solved the Fibonacci Prime problem. However, ChatGPT 3.5 used an inefficient solution to calculate the primes. It took significantly longer to solve the benchmark of $n = 10$. For the Collatz Number problem, both LLMs and most of the humans struggled to create solutions. One of the human programmers developed a successful recursive solution that works for all required inputs. Three other human programmers developed technically correct solutions but required a brute force approach.

Face Validation (Survey 2)

A second survey was conducted to have a group of subject matter experts evaluate the quality of the code produced by the LLMs and compare it to the samples collected from expert human programmers. Similar to the first survey, Google Forms was used to collect responses. However, unlike the first survey, collecting a large number of results was the highest priority. To maximize response collection, the survey was promoted on social media platforms (Twitter, Discord, and LinkedIn) and shared on the survey community SurveyCircle, and a small prize raffle was introduced. A verified Google account was recorded for each respondent to ensure response quality (and prevent raffle abuse).

Respondents were presented with two example solutions for each of the two proposed programming problems. For each example, the respondent was asked, “How confident are you that the sample was generated using generative AI?” Responses were measured on a Likert scale from 1 (very confident no generative AI was used) to 5 (very confident generative AI was used). After both samples were shown, the respondent was asked, “Which sample is more correct?” and “Which sample is easier to understand?” Respondents could choose one of the two samples or select “about the same.”

Each respondent was asked two additional questions to provide context for the responses. The first question asked about the respondent’s level of coding experience, with four options: 0-3 years, 3-5 years, 5-10 years, and 10+ years.

This question aimed to gather demographic information and potentially weigh the responses for a more nuanced analysis. Additionally, respondents were invited to provide feedback in a free-form text format.

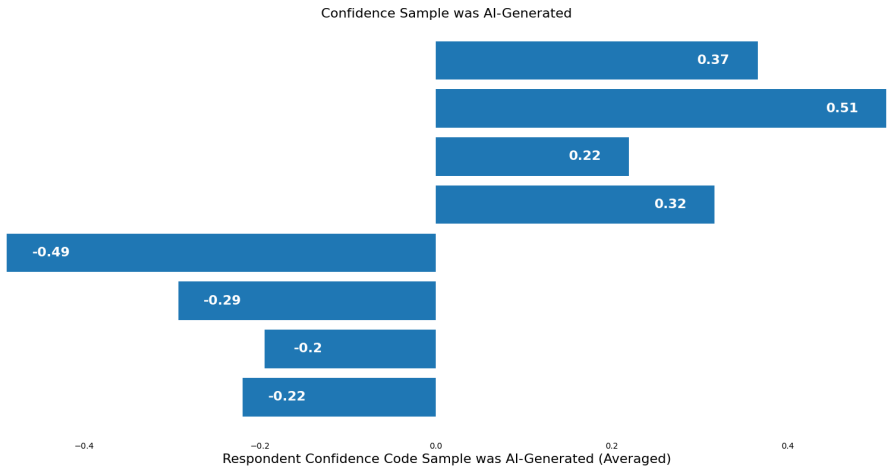

Figure 6 Survey Results Indicating the Respondent's Confidence the Sample was AI-Generated

In total, 85 total responses were collected. Krippendorf's alpha was used to measure the reliability of the results. An alpha value of 0.68 indicated the results were valid enough to draw a tentative conclusion. The graph in Figure 6 shows the average of the survey results for each code sample. In this graph, the first four samples were AI-generated. While confidence values are small, the sample size, along with the high Krippendorf's alpha value, indicate that experienced software engineers can successfully distinguish between samples that were developed by humans and those that were AI-generated.

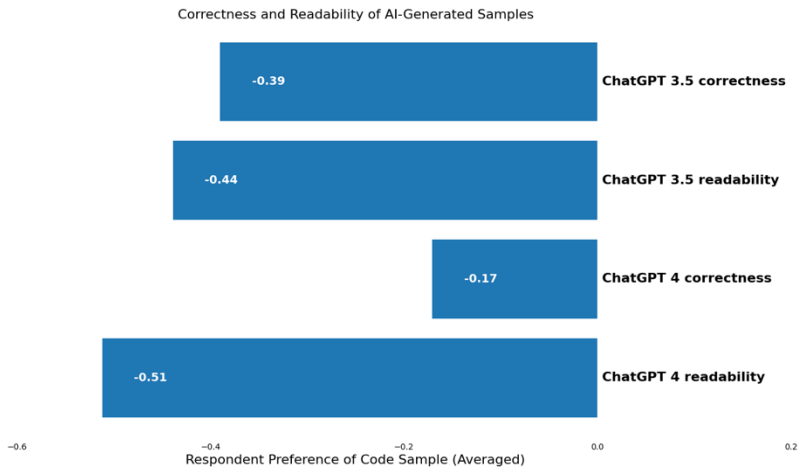

Figure 7 Survey Results Indicating Respondent Preference of Code Samples

Respondents were also asked to indicate their preference for each pair of code samples for both correctness and readability. The average of all responses is shown in Figure 7. Overall, the respondents preferred the code generated by human experts over generative AI models. As expected, the newer ChatGPT 4 produced more correct code on average. However, the respondents indicated that ChatGPT 3.5 produced more readable code. When the results were restricted to only those respondents with five or more years of experience, the readability results were roughly equivalent.

Model Use Detection

A third benchmark for comparing code generated by LLMs to that produced by human software engineers involved assessing the feasibility of developing an ML classifier to differentiate between the two types of code samples. The rationale behind this benchmark is that while LLM-generated code is not inherently problematic, it may exhibit distinct patterns or characteristics that could pose potential risks. These patterns could be exploited or become evident through adversarial testing, highlighting vulnerabilities or areas of concern. By developing an ML classifier, the goal is to identify these patterns, thereby providing insights into the robustness and reliability of LLM-generated code.

The dataset used to train the ML classifier was created by combining two Hugging Face datasets: one consisting of Python GitHub projects (FlyTech, n.d.) and another comprising recordings of interactions with an LLM generating code samples (Wolf, n.d.). Significant preprocessing was performed on the datasets before training the classifier. For the GitHub project samples, license information and URLs were removed. For the LLM interaction samples, all instructional information was removed, leaving only the code portions. Additionally, comments were removed from all samples, and for texts longer than 500 characters, a random 500-character portion was selected. This preprocessing ensured that the classifier was trained on a clean and consistent dataset, focusing solely on the code itself.

The training of the classifier was performed on a commodity laptop (Apple MacBook Pro, M1 Max) and took roughly 3 hours to complete. The classifier was able to achieve an accuracy of 0.962 on the held-out test dataset. Each of the four LLM generated code samples and the eight human solutions that were developed in Python (including the researcher's solutions) were tested using the classifier.

Table 4 Results of the Classifier Indicating if the Sample was AI-Generated

LLM	Probability		Human	Probability
1	0.9983		1	0.0484
2	0.9968		2	0.5674
3	0.9862		3	0.9982
4	0.4327		4	0.5730
			5	0.1674
			6	0.0432
			7	0.6079
			8	0.6275

Table 4 shows the results of the classifier run on each code sample. The classifier successfully identified three of four LLM-generated solutions and three of eight human-generated solutions. By raising the classification threshold to 0.7, the false positive rate was reduced, allowing the classifier to identify seven out of the eight human solutions accurately. However, due to the non-proctored nature of the survey, it is impossible to ascertain whether the human participants utilized LLMs or previously generated LLM solutions. Solutions authored by the researcher, with complete certainty that no LLM was used, are highlighted.

The classifier demonstrates promising potential in detecting code samples written by an LLM but has one apparent shortcoming. There is a dichotomy in the motivation for developing the samples between the two classes of data used to train the model. The human-generated code samples were taken from a selection of publicly available GitHub repositories, while the LLM-generated samples were derived from interactions with an LLM. This fundamental difference in the datasets is significant: interactions with an LLM typically focus on solving small, encapsulated problems, whereas GitHub projects contain more comprehensive and contextually rich code.

This discrepancy likely impacts the classifier's performance, as the variability and complexity of the human-generated code from GitHub repositories are inherently different from the more homogeneous and potentially less intricate samples produced by an LLM. Moreover, the human-generated code often involves collaborative efforts, incorporating diverse coding styles and problem-solving approaches, while the LLM-generated samples reflect a

more singular style and approach dictated by the model's training data. Addressing this dataset imbalance could involve curating more aligned and contextually similar samples from both sources to enhance the classifier's accuracy and reliability in distinguishing between human and LLM-generated code.

CONCLUSION

The rapid deployment of LLMs like OpenAI's ChatGPT across various sectors highlights the urgent need for robust verification and validation techniques. While these models possess extensive comprehension of linguistic nuances and tasks within their training datasets, their reliability remains questionable, necessitating thorough validation. This paper has focused on verifying and validating a current and next-generation model that produce source code based on user-provided prompts.

Both tested LLMs were able to solve a simplistic problem but struggled to solve a more complex one. In addition to needing less instructional feedback, the next generation LLM was able to produce more efficient solutions. In the face validation study, human experts were able to successfully distinguish between LLM-generated and human-generated samples and preferred the human-generated code for both correctness and readability. Finally, an ML classifier was developed that could effectively distinguish between the two classes of samples. These findings highlight the current capabilities and limitations of LLMs in code generation. While LLMs show promise in solving basic problems and producing efficient solutions, human expertise remains crucial for complex problem-solving and maintaining code quality.

REFERENCES

Bartolo, M., Roberts, A., Welbl, J., Riedel, S., & Stenetorp, P. (2020). Beat the AI: Investigating adversarial human annotation for reading comprehension. *Transactions of the Association for Computational Linguistics, 8*, 662–678. https://doi.org/10.1162/tacl_a_00338

Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? 🦜 In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610–623). Association for Computing Machinery. https://doi.org/10.1145/3442188.3445922

Brown, T. B., et al. (2020). Language models are few-shot learners. arXiv. https://arxiv.org/abs/2005.14165

FlyTech. (n.d.). Python Codes 25k [Data set]. Hugging Face. https://huggingface.co/datasets/flytech/python-codes-25k

Gopinath, D., Katz, G., Pasareanu, C. S., & Barrett, C. (2020). DeepSafe: A data-driven approach for checking adversarial robustness in neural networks. arXiv. https://arxiv.org/abs/1710.00486

Jia, R., & Liang, P. (2017). Adversarial examples for evaluating reading comprehension systems. *arXiv preprint arXiv:1707.07328*. Retrieved from https://arxiv.org/abs/1707.07328

Krippendorff, Klaus. (2011a). Agreement and Information in the Reliability of Coding. Communication Methods and Measures. 5. 93-112. 10.1080/19312458.2011.568376.

Krippendorff, K. (2011b). Computing Krippendorff's Alpha-Reliability.

Online Encyclopedia of Integer Sequences. (n.d.). Sequence A005478. Retrieved May 29, 2024, from https://oeis.org/A005478

Petty, M. D. (2013). *Model verification and validation methods*. University of Alabama in Huntsville. Retrieved from https://www.uah.edu/images/research/cmsa/pdf/Petty%202013%20Model%20VV%20Methods%20v4.pdf

Ribeiro, M. T., Singh, S., & Guestrin, C. (2018). Anchors: High-Precision Model-Agnostic Explanations. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1). https://doi.org/10.1609/aaai.v32i1.11491

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2023). Attention is all you need. arXiv. https://arxiv.org/abs/1706.03762

Wolf, T. (n.d.). GitHub Python [Data set]. Hugging Face. https://huggingface.co/datasets/thomwolf/github-python