How Does ChatGPT Make Assumptions When Creating Erroneous Programs?

Sadia Jahan
The University of Texas at San Antonio,
Texas, USA,
sadia.jahan@utsa.edu

Xiaoyin Wang
The University of Texas at San Antonio,
Texas, USA,
xiaoyin.wang@utsa.edu

***Abstract*—Large Language Models (LLMs) like ChatGPT are increasingly integrated into software development environments due to their strong performance in code generation. However, they often struggle with complex logic, security vulnerabilities, and code quality issues. These problems frequently originate from misunderstandings of problem requirements and logical inconsistencies, which can lead to faulty or vulnerable software. In this study, we conduct an initial empirical analysis to investigate the causes of erroneous code generated by the state-of-the-art LLM model GPT-4o. Using the HumanEval dataset, we prompt GPT-4o to generate Python solutions and list its 3 most important assumptions. We validate these outputs against the provided test cases in dataset and identify 17 defective programs out of 164 total solutions. By analyzing the 17 failures and 51 assumptions made on these tasks, we find that about 53% the failures are directly related to wrong or erroneously implemented assumptions raised by the GPT model itself, and totally 71% of code generation failures are related to erroneously made or implemented assumptions.**

I. Introduction

LLM-based code generation has made significant strides, with models like OpenAI's GPT-4 [1], DeepMind's AlphaCode [2], and Meta's Code Llama [3] demonstrating the ability to generate functional code, assist in debugging, and even optimize performance. These models are increasingly integrated into development environments, such as GitHub Copilot [4] and Amazon CodeWhisperer [5]. While they excel at generating boilerplate code and automating small tasks, they still struggle with complex logic, security [6] [7] and privacy [8], quality of code with dependencies [9] [10] [11].

Studying scenarios where LLMs generate erroneous programs is crucial for understanding their limitations and improving their reliability. These errors can stem from misunderstanding problem constraints, logical inconsistencies, or even subtle security vulnerabilities, which could lead to faulty or unsafe software. By analyzing these failure cases, researchers and developers can refine model architectures, enhance training data, and implement better guardrails to mitigate risks. Additionally, identifying patterns in mistakes helps create tools for automated error detection and correction, reducing the burden on human developers. Ultimately, this research is essential for making LLM-based code generation more trustworthy and practical for real-world applications.

Unclear requirements are one of the most common causes of bugs in code [12] because they lead to misunderstandings between stakeholders, developers, and automated systems like LLMs. When requirements are vague, incomplete, or ambiguous, developers may make incorrect assumptions, resulting in functionality that does not align with user expectations. This often leads to logic errors, unexpected edge cases, and difficulties in maintaining or extending the software. Clear, well-defined requirements help ensure that both humans and AI models produce accurate and efficient code. Improving requirement clarity through better documentation, structured specifications, and iterative feedback loops is essential for reducing software defects.

In this paper, to better understand whether and how the missing details in requirement prompts may affect LLM-based code generation, we present an empirical study on the GPT-4o model and its application on the HumanEval [13] dataset. For each prompt in the HumanEval dataset, We first feed it to GPT-4o and ask the model to raise three questions that may help it to better create the program, and then ask the model to give its own assumption on the answer of each of the questions. After this, we ask the model to generate the program. After we finish this process for all prompts, we run the generated python programs against the test suite in the HumanEval dataset to check their correctness. From this checking process, we identified 17 erroneously generated programs. Finally, we manually investigated the assumptions ChatGPT made on the 17 programs, and compared them with the defects in the programs.

In particular, our initial study tries to answer the following research questions.

- Is ChatGPT asking reasonable questions and making correct assumptions when generating code from prompts?
- How frequently are failed code generations caused by missing information and wrong assumptions?

II. The Empirical Study

A. Study Process

To conduct our research, we utilized the HumanEval dataset [13] consisting of problem statements and corresponding test cases. The study involved a two-step approach using the GPT-4o mini model (gpt-4o-mini-2024-07-18) [1]. In one step, we provided the HumanEval prompts to the model and instructed it to generate 3 key information items along with explicit assumptions made by the model regarding that information. Our prompt for this step is structured as follows.

> "If you are asked to generate python code given the following requirement, please list up to 3 additional information items that would be helpful to produce a more precise solution. Also, what is your current assumption on those information items."

In the second step, we ask the same GPT-4o mini model to generate the solution code based on the HumanEval prompts. The generated solutions were then validated using the test cases provided with the HumanEval dataset. After validation, we performed an analysis to identify and investigate the failed test cases. This analysis included determining the root causes of each identified failure. Our research workflow is presented in Figure 1.

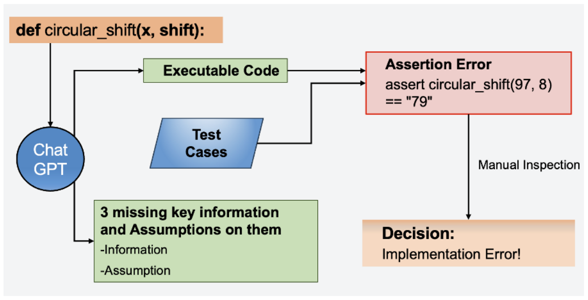

Fig. 1. Research Workflow Diagram

B. Study Results

We analyzed 17 failed code generations by ChatGPT. The investigation of the failures is presented in Tables I to IV. In particular, based on our findings from the study, we define and categorize four types of code generation errors as follows.

- **Explicit Wrong Assumptions** happen when the code generation error is caused by a wrong assumption listed as an answer of the top three questions raised by the GPT model. For this type of errors, the GPT model successfully identified an unclear requirement in the prompt and asked a correct question, but failed to make the correct assumption to answer it.
- **Implicit Wrong Assumptions** happen when the code generation error is caused by a wrong assumption, but the assumption is not list as an answer of the questions raised by the model. For this type of errors, the GPT model did not realize the unclear requirement and made a wrong assumption without doubt.
- **Erroneously Implemented Assumptions** happen when the GPT model correctly made all assumptions, but the generated code is violating one of the assumptions. This type of errors show that the GPT model's comprehension capability is not aligned with its code generation ability.
- **Other Implementation Errors** happen when the code generation error is caused by wrong implementation of a clear requirement in the prompt.

From Table I, we can see that there are four code generation errors falling into the category of explicit wrong assumptions. For all these errors, the wrong assumptions are either the first or the second one, showing that the GPT model is able to identify the unclear requirement very quickly. Most of the unclear requirements are related to special cases (i.e., punctuations in prompt 86, trailing white spaces in prompt 134, and negative numbers in prompt 145).

From Table II, we can see that there are three code generation errors falling into the category of implicit wrong assumptions. Among these errors, for prompt 163, the GPT model misunderstood the meaning of "digit", which is used in the prompt in a less common way. Typically a "digit" is referred to a part of a number, but it is referred to a whole number between 0 and 9 in this prompt, which confused the model. For the other two prompts, the GPT model makes wrong assumptions on the corner cases but did not realize that there were other possibilities to implement those corner cases.

From Table III, we can see that there are five code generation errors falling into the category of assumption implementation errors. Four out of the five errors are caused by the erroneously implementation of Assumption 1 (the remaining fails to implement Assumption 2 correctly), showing that the GPT model is able to identify these assumptions as the key of implementing the prompt correctly. However, the implementation was not consistent with the assumptions, indicating that the GPT model is not able to produce the implementation even if its understanding is correct. However, this also shows that if asking the model or another model to double check the code semantics against the assumptions, an error may be detected automatically.

From Table IV, we can see that there are five code generation errors falling into the category of other implementation errors. Except for prompt 154 where the generated code does not make any sense, the GPT model is able to generate an almost-correct program for all remaining four prompts and all the errors are one-line errors. For prompts 65, 118, and 135, the errors are just an operator or expression error. For prompt 83, the error is also just a missing `if` condition on one block, and the condition has already been existing on other blocks. This observation shows that the generated code errors due to other implementation errors may be detected through mutation oriented testing and relatively easily fixed through automatic repair techniques.

C. Result Summary and Lessons Learned

Summarizing the analysis and findings above, we have the following summary and lessons learned.

- Even for the prompts with erroneously generated code, ChatGPT is able to raise the correct questions about the missing information more than half of them time. In particular, 53% (Table I and III, 9 of 17) of the code errors are directly related to wrong assumptions of the raised questions, or erroneous implementation of correct assumptions. Furthermore, all of the error-related

TABLE I
THE ASSUMPTIONS AND DEFECT SUMMARIES OF EXPLICIT WRONG ASSUMPTION ERRORS

ID	Prompt Summary	Assumptions	Defect Summary
86	Write a function that takes a string and returns an ordered version of it.	1. Punctuation should be preserved at their original position in the string. 2. The sorting is case-sensitive. 3. The exact spacing in the original string is maintained.	Assumption 1 is wrong. Special characters should not be preserved with their original position but also be involved in the shuffle.
130	Return the first $n+1$ numbers of the Tribonacci sequence: $tri(1)=3$, $tri(n) = 1+n/2$, if n is even, $tri(n) = tri(n-1)+tri(n-2)+tri(n+1)$, if n is odd.	1. The sequence's recurrence for odd $'n'$ is not standard. 2. Unclear about the value of $'tri(0)'$ 3. The sequence starts at $'tri(1) = 3'$.	Assumptions 1 and 2 are wrong. The model was confused by the first and second questions and cannot provide valid assumptions.
134	Create a function that returns True if the last character of a given string is an alphabetical character and is not a part of a word, and False otherwise.	1. A "word" is a group of alphabetical characters only. 2. The function will ignore trailing whitespace. 3. The function should return 'False' for an empty string	Assumption 2 is wrong. The model assumes that black spaces should be ignored which is not the case.
145	Write a function which sorts the given list of integers in ascending order according to the sum of their digits.	1. The digit sum will ignore negative signs. 2. The function will return an empty list for an empty input. and will work only with lists containing integers. 3. The function will return a new list that is the sorted version of the input	Assumption 1 is wrong. The model assumes negative signs should be ignored, which is not the case.

TABLE II
THE ASSUMPTIONS AND DEFECT SUMMARIES OF IMPLICIT WRONG ASSUMPTION ERRORS

ID	Prompt Summary	Assumptions	Defect Summary
93	Write a function that takes a message, and encodes in such a way that it swaps case of letters, replaces each vowel with the letter that appears 2 places ahead in the english alphabet.	1. Only vowels need to be replaced and the function handles both lowercase and uppercase vowels. 2. Non-alphabetic characters remain unchanged in the output. 3. When a vowel is replaced by the letter two places ahead in the alphabet, the replacement should wrap around if it goes past 'z' or 'Z'.	The program makes a wrong assumption on whether to swap case for vowel letters. The assumption is not raised as a question.
101	Given a string of words separated by commas or spaces, split the string into words and return an array of the words.	1. The function should split the string based on commas and spaces, and handle consecutive spaces as a single delimiter. 2. Preserve the original form of the words, including any punctuation attached to them. 3. Return all words as they appear in the input string	The program wrongly assumes an empty word may exist after a delimiter at the end of the string. No related questions were asked.
163	Given two positive integers a and b, return the even digits between a and b, in ascending order.	1. The range of a and b is inclusive 2. Both a and b are positive integers 3. The output must be a list	Missed the requirement that digits needing to be between 0 and 9. No related questions were asked.

TABLE III
THE ASSUMPTIONS AND DEFECT SUMMARIES OF ERRONEOUSLY IMPLEMENTED ASSUMPTIONS

ID	Prompt Summary	Assumptions	Defect Summary
32	Evaluates polynomial with coefficients xs at point x.	1. xs starts with the constant term. 2. xs has no length limit 3. The return value can be a floating-point value.	Assumption 2 is correct, but the implementation is wrong and requires the length of xs to be an even number.
55	Return n-th Fibonacci number	1. $fib(0) = 0$ and $fib(1) = 1$ (Base Cases) 2. Input n is a non-negative integer 3. May not perform well for large values of n	Assumption 1 is correct, but the implementation is wrong and set $fib(0)$ to error and $fib(1)$ and $fib(2)$ both to 1.
108	Write a function that takes an integer array and returns the number of elements with a sum of digits $>$ 0. If a number is negative, then its first digit will be negative.	1. For negative numbers, the negative sign is considered part of the first digit. 2. Input array will contain integers only. 3. Length of input array can vary.	Assumption 1 is correct but the implementation is wrong. It separately considers the negative sign as -1, instead of considering the negative sign and the digit after it as a whole
140	Given a string text, replace all spaces in it with underscores. If a string has more than 2 consecutive spaces, replace all consecutive spaces with -	1. Leading, trailing spaces will be replaced with underscores. 2. All consecutive spaces (more than two) should be replaced with a hyphen. 3. Existing underscores should remain unchanged	Assumption 1 is correct, but the implementation is wrong and ignored leading and trailing spaces due to wrong API usage
160	Given two lists, one with algebraic operators and the other one with integers, use them to build the algebraic expression and return the evaluation of this expression.	1. It will respect operator precedence. 2. The input will be valid. 3. 'eval()' can be used to build and evaluate the expression.	Assumption 1 is correct, but its implementation is wrong and fails to follow the mathematic operation preference (multiplication higher than addition), maybe due to lack of reference

TABLE IV
THE ASSUMPTIONS AND DEFECT SUMMARIES OF OTHER IMPLEMENTATION ERRORS

ID	Prompt Summary	Assumptions	Defect Summary
65	Circular shift the digits of the integer x, shift the digits right by $shift$ and return the result as a string. If $shift >$ number of digits, return digits reversed.	1. Input x is always a non-negative integer 2. Input $shift$ is a non-negative integer 3. If it's greater than the number of digits, reverse the digits	Off by one error when shift is exactly the same with the number of the digits, $>$ instead of $>=$ should be used
83	Given a positive integer n, return the count of the numbers of n-digit positive integers that start or end with 1.	1. n-digit positive integers cannot have leading zeros 2. Unclear of whether to count numbers that start with 1, end with 1, or both. Additionally, if a number both starts and ends with 1 should be counted once or twice. 3. n should be at least 1.	Partially generated code, in which special processing of input `n = 1` is applied to numbers starting with 1 but not numbers ending with 1
118	Given a word the closest vowel that stands between two consonants from the right side of the word (case sensitive).	1. Vowels are defined as 'a', 'e', 'i', 'o', 'u' (including their uppercase versions) 2. The function should return an empty string for single-letter words, words made entirely of vowels, or when two consecutive consonants do not enclose a vowel. 3. The input is a word with English letters only.	Logic error, the model misunderstood the requirement of between two letters, but checked the two letters before the vowel letter.
135	Create a function which returns the largest index of an element not greater than or equal to the element immediately preceding it. If no such element then return -1.	1. The function will return -1 for empty arrays. 2. The function will work with an array of integers. 3. The function should return -1 if no valid index exists.	Logic Error, the generated code returns the qualified element with smallest instead of the largest index.
154	Given 2 words, return True if the second word or any of its rotations is a substring in the first word.	1. A rotation of a word involves moving characters from the beginning of the string to the end. 2. The check is case-sensitive. 3. The input strings can contain alphabetic characters only.	All the assumptions are correct but the implementation does not make any sense, indicating a lack of reference.

assumptions are raised as the first (mostly) or the second assumptions. This shows that through double checking of the first several questions and assumptions raised by the GPT model, we may easily detect or fix the errors in GPT-generated code.

- Among the remaining code errors, additional 3 errors (Table II, 18%) are caused by erroneous assumptions which are not raised as one of the top three questions by GPT-4o. So, in total, about 71% of all the errors in generated code are caused by either wrong assumptions or erroneously implemented assumptions. This observation shows that checking assumptions made by the LLM models may have further potential in enhancing the code generation quality.
- The remaining code errors (29%) are generic code logic errors. Furthermore, 80% (4 out of 5) of them are one-line errors, which may be detected or fixed through mutation-oriented testing or repair techniques.

III. RELATED WORKS

AI-driven code generation began with initial efforts in API method recommendation [14] [15], code auto-completion [16], and script repair [17] [18] using traditional machine learning techniques [19]. These approaches later evolved to incorporate deep neural networks [16] [20]. Inspired by the idea that code shares similarities with natural language [21], researchers started training large-scale code models [22] or integrating vast amounts of code into general-purpose large language models [23]. This progression has streamlined the coding process, reduced human error [24], and contributed to more efficient and dependable software development.

Research have been performed to study the errors of LLM-based code generation. Liu et al. [25], Song et al. [26], and Chen et al. [27] all performed categorization of the LLM-based code generation errors. However, their taxonomy all focuses on the code structures, and does not take into consideration the questions / assumptions raised by the model (which are enabled in our study). There are additional works on the interaction-based LLM code generation [28], which we believe may benefit from the findings of our study.

IV. CONCLUSION AND FUTURE WORKS

In this empirical study, we analyzed the generated code and assumptions made by ChatGPT using problem statements from the HumanEval dataset, identifying 17 code defects out of 164 evaluated cases. We find that more than half of the code generation errors are directly related to the first of second questions raised by the GPT model, and almost all the remaining errors are related to either wrongly made assumptions or one-line logic errors. These initial findings show a promising direction to validate and repair LLM-generated code based on its assumptions. In the future, we plan to (1) expand our study to more datasets and LLM models to check whether our findings are generalizable, and (2) leverage our findings on LLM assumptions to detect or repair code generation errors, such as interactive checking of assumption and checking the consistency between assumptions and code semantics.

References

[1] 2024, openAI GPT-4O.

[2] 2024, deepMind Alpha Code.

[3] 2024, code llama.

[4] 2024, github CoPilot.

[5] 2024, amazon Code Whisper.

[6] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 536–547.

[7] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1559–1575.

[8] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violations in android application code," in *Proceedings of the 38th International conference on software engineering*, 2016, pp. 25–36.

[9] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 481–503, 2025.

[10] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 457–466.

[11] J. Molina, X. Qin, and X. Wang, "Automatic extraction of code dependency in virtual reality software," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 381–385.

[12] G. S. Walia, J. Carver, and T. Philip, "Requirement error abstraction and classification: an empirical study," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 336–345.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[14] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 353–363.

[15] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 293–304.

[16] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, "Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches," *AI*, vol. 2, no. 1, pp. 1–16, 2021.

[17] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1078–1089.

[18] ——, "Mining readme files to support automatic building of java projects in software repositories," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 277–279.

[19] X. Liu, L. Huang, and V. Ng, "Effective api recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 282–292.

[20] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proceedings of The Web Conference 2020*, 2020, pp. 2309–2319.

[21] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[22] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 323–334.

[23] G. Adesso, "Gpt4: The ultimate brain," *Authorea Preprints*, 2022.

[24] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

[25] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[26] D. Song, Z. Zhou, Z. Wang, Y. Huang, S. Chen, B. Kou, L. Ma, and T. Zhang, "An empirical study of code generation errors made by large language models," in *7th Annual Symposium on Machine Programming*, 2023.

[27] Q. Chen, J. Yu, J. Li, J. Deng, J. T. J. Chen, and I. Ahmed, "A deep dive into large language model code generation mistakes: What and why?" *arXiv preprint arXiv:2411.01414*, 2024.

[28] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, 2024.