Fight Fire With Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks?

Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, *Senior Member, IEEE*, Jin Liu, and Xin Xia, *Senior Member, IEEE*

***Abstract*—With the increasing utilization of large language models such as ChatGPT during software development, it has become crucial to verify the quality of code content it generates. Recent studies proposed utilizing ChatGPT as both a developer and tester for multi-agent collaborative software development. The multi-agent collaboration empowers ChatGPT to produce test reports for its generated code, enabling it to self-verify the code content and fix bugs based on these reports. However, these studies did not assess the effectiveness of the generated test reports in validating the code. Therefore, we conduct a comprehensive empirical investigation to evaluate ChatGPT's self-verification capability in code generation, code completion, and program repair. We request ChatGPT to (1) generate correct code and then self-verify its correctness; (2) complete code without vulnerabilities and then self-verify for the presence of vulnerabilities; and (3) repair buggy code and then self-verify whether the bugs are resolved. Our findings on two code generation datasets, one code completion dataset, and two program repair datasets reveal the following observations: (1) ChatGPT often erroneously predicts its generated incorrect code as correct, its vulnerable completed code as non-vulnerable, and its failed program repairs as successful during its self-verification. (2) The self-contradictory hallucinations in ChatGPT's behavior arise: (a) ChatGPT initially generates code that it believes to be correct but later predicts it to be incorrect; (b) ChatGPT initially generates code completions that it deems secure but later predicts them to be vulnerable; (c) ChatGPT initially outputs code that it considers successfully repaired but later predicts it to be buggy during its self-verification. (3) The self-verification capability of ChatGPT can be enhanced by asking the guiding question, which queries whether ChatGPT agrees with assertions about incorrectly generated or repaired code and vulnerabilities in completed code. (4) Using test reports generated by ChatGPT can identify more vulnerabilities in completed code, but the explanations for incorrectly generated code and failed repairs are mostly inaccurate in the test reports. Based on these findings, we provide implications for further research or development using ChatGPT.**

***Index Terms*—Empirical study, ChatGPT, self-verification, code generation, code completion, program repair.**

I. Introduction

LARGE Language Models (LLMs), especially the high-performing ChatGPT have demonstrated impressive capabilities across various software development tasks, including code generation [1], [2], [3], [4], code completion [5], [6], and program repair [7], [8]. These capabilities accelerate development processes and simplify daily tasks for software developers. However, ChatGPT-generated code may have quality issues or vulnerabilities [9], [10], emphasizing the need for thorough quality checks. Recently, researchers [1], [3] proposed utilizing multi-agents where ChatGPT acts both as a developer and a tester. This approach enables ChatGPT to generate test reports for its generated code and fix bugs based on the reports. However, they did not evaluate whether the generated test reports effectively validate the code (i.e., ChatGPT's self-verification capability). Therefore, we conduct a comprehensive empirical study to evaluate ChatGPT's self-verification capability across three code-related tasks (i.e., code generation, code completion, and program repair) using the three specifically designed prompts: direct question, guiding question, and test report. We address the following three Research Questions (RQs):

RQ1: How effective is ChatGPT's self-verification capability in code generation using the direct question, guiding question, and test report prompts? We first ask ChatGPT to generate code based on the requirement description and then verify if the code meets the requirements. During self-verification, we use the direct question prompt to evaluate if the code correctly implements the requirements, the guiding question prompt to agree or disagree with assertions that the code does not implement the function based on the requirement description, and the test report prompt to generate test reports for the generated code to verify correctness.

Received 20 April 2024; revised 21 October 2024; accepted 27 October 2024. Date of publication 5 November 2024; date of current version 12 December 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 61972290, in part by Ningbo Natural Science Foundation under Grant 2023J292, and in part by the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong under Grant 6000796, Grant 9229109, Grant 9229098, Grant 9220103, and Grant 9229029. Recommended for acceptance by F. Ferrucci. *(Corresponding author: Xing Hu.)*

Xiao Yu and Xing Hu are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang 310058, China (e-mail: xiaoyu_cs@hotmail.com; xinghu@zju.edu.cn).

Lei Liu is with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, Shanxi 710049, China (e-mail: Lei.Liu@stu.xjtu.edu.cn).

Jacky Wai Keung is with the Department of Computer Science, City University of Hong Kong, Hong Kong 999077, China (e-mail: jacky.keung@cityu.edu.hk).

Jin Liu is with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: jinliu@whu.edu.cn).

Xin Xia is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China (e-mail: xin.xia@acm.org).

Digital Object Identifier 10.1109/TSE.2024.3492204

0098-5589 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

RQ2: How effective is ChatGPT's self-verification capability in code completion using the direct question, guiding question, and test report prompts? We ask ChatGPT to complete the code and ensure it has no vulnerabilities, then question ChatGPT about any potential vulnerabilities in the completed code. During self-verification, we use the direct question prompt to explicitly ask if the code correctly implements the requirement description, the guiding question prompt to ask for agreement or disagreement with assertions that the completed code has vulnerabilities, and the test report prompt to generate a test report for the completed code to self-verify any vulnerabilities.

RQ3: How effective is ChatGPT's self-verification capability in program repair using the direct question, guiding question, and test report prompts? We ask ChatGPT to repair a buggy program and then question if the code is successfully repaired. During self-verification, we use the direct question prompt to explicitly ask if the repaired code correctly implements the function, the guiding question prompt to ask for agreement or disagreement with assertions that the repaired code does not correctly implement the function, and the test report prompt to generate a test report for the repaired code to self-verify the success of the repair process.

We conduct experiments on two code generation datasets, one code completion dataset, and two program repair datasets. The experiment results are as follows:

(1) ChatGPT possesses a certain level of capability in generating correct code with an average success rate of 57%, providing code completions without vulnerabilities with a success rate of 73%, and successfully repairing code with an average success rate of 70%. When explicitly asking about the correctness of the generated code, absence of vulnerabilities in code completions, or the success of code repairs, ChatGPT often erroneously believes that it has accomplished these tasks, with average error rates of 39%, 25%, and 28%, respectively.

(2) The guiding question prompt leads to the detection of an average of 25% more incorrectly generated code, the identification of 69% more vulnerabilities in completed code, and the recognition of an average of 33% more failed program repairs. However, it is important to acknowledge that despite these improvements, there are still many cases where ChatGPT is unable to successfully self-verify incorrectly generated code (67% average missing report rate), vulnerabilities in completed code (23% missing report rate), and failed program repairs (59% average missing report rate).

(3) Utilizing the test report prompt enables ChatGPT to successfully identify an average of 77% more vulnerable completed code and provide accurate explanations for the vulnerabilities. For the program repair task, the test report prompt can identify an average of 28% more failed program repairs. However, in the code generation task, the test report prompt does not improve the detection of generated incorrect code substantially. Furthermore, the explanations provided in the test report are mostly (an average of 75%) inaccurate for incorrectly generated code and failed repairs.

(4) There are instances of self-contradictory hallucinations[1] in ChatGPT's behavior: (a) Initially, ChatGPT generates or completes code that it believes to be correct and non-vulnerable, but it predicts it to be incorrect and vulnerable during self-verification. (b) ChatGPT initially outputs code that it believes to be successfully repaired, but it predicts it to fail during self-verification.

Overall, the inaccuracies and self-contradictory hallucinations observed during ChatGPT's self-verification highlight the crucial role of human expertise and judgment in software development and evaluation in the current stage. ChatGPT should be seen as a tool to assist developers rather than a substitute for their role as autonomous software developers and testers. Developers must carefully evaluate the output of ChatGPT, and conduct their own assessments to ensure the quality and reliability of the generated code. Furthermore, efforts to enhance the performance of ChatGPT should focus on eliminating self-contradictory hallucinations to ensure a more reliable experience.

Our contributions are summarized as follows:

(1) To the best of our knowledge, we are the first to perform a comprehensive empirical study to examine the self-verification capability of ChatGPT in code-related tasks, i.e., code generation, code completion, and program repair.

(2) We make actionable findings regarding the self-verification performance of ChatGPT and provide implications for the adoption and development of ChatGPT.

II. Study Design

Given ChatGPT's primary focus on content generation, we evaluate its performance on three code-related tasks: code generation, code completion, and program repair. These tasks are widely used in daily software development and involve extensive code creation. It is crucial to ensure that the generated code is free from vulnerabilities or bugs when developers incorporate it into their projects. For the code generation and program repair tasks, we assess the correctness of the generated code using the provided test cases within the experimental datasets. Any code that fails to pass a test case is considered incorrect [12]. In the code generation task, code that successfully passes all test cases is deemed correct. For the program repair task, automatic program repair techniques may suffer from the patch overfitting problem [13], where a repaired program passes all the tests but is still incorrect. Therefore, apart from assessing if the repaired code can pass all test cases, we engage three software developers with over five years of experience to conduct independent evaluations. Each developer thoroughly reviews the repaired program to confirm the successful fixing of the buggy code. For the code completion task, we utilize the GitHub CodeQL [14] tool to scan the completed code for vulnerabilities associated with specific CWEs, as outlined by Pearce et al. [9].

[1]In the context of LLMs, "hallucination" refers to the phenomenon where LLMs produce text that is incorrect, nonsensical, or fabricated. Mündler et al. [11] define "self-contradictory hallucinations" as instances where an LLM produces two logically inconsistent sentences within the same context. We have adopted Mündler et al.'s definition of self-contradictory hallucinations.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

After the automated scan, the same three software developers manually inspect the vulnerabilities flagged by CodeQL to verify their correctness. This manual inspection serves two purposes: verifying the accuracy of the CodeQL results and correcting any false positives or overlooked vulnerabilities. To further ensure the reliability of our evaluations and reduce bias, we calculate the Fleiss' kappa score, which is 0.81, indicating substantial agreement among the three developers. In cases of disagreement, the three developers discuss the issues to reach a consensus, thereby mitigating individual biases.

A. Code Generation

Datasets. We select the two widely used datasets that contain test cases, namely, MBXP [15] and HumanEval-X [16]. The MBXP dataset [15] consists of 848-974 coding problems for 13 programming languages. Each problem includes task_id, declaration, docstring, prompt, canonical_solution, and test program with 3 test cases. The HumanEval-X dataset [16] consists of 820 human-crafted problem-solution pairs covering 164 coding problems in five languages. Each problem includes the task_id, prompt, declaration, canonical_solution, and test program with some test cases.

Step 1: We first request ChatGPT to generate the correct code based on the provided requirement description. We design the prompt consisting of triple items, i.e., <requirement, function description, function declaration>. Test cases are not included in the prompt, since Cassano et al. [12] suggested that it is a better way to evaluate code generation. For instance, in Fig. 1, the requirement is "*Generate the code and ensure that the generated code is correct.*", the function description is "*Write a JavaScript function to find the element that appears only once in a sorted array.*", and the function declaration is "*function search(arr, n){*".

Step 2 (a): After generating the code, we utilize the **direct question** prompt to explicitly ask ChatGPT whether the code correctly implements the function based on the requirement description and requests a direct response. The direct question prompt shown in Fig. 1 is "*Does the code correctly implement 'search' function based on the description of 'Write a JavaScript ... sorted array.'? Answer the question (yes or no): $code$*", where *$code$* represents the generated code by ChatGPT.

Step 2 (b): In addition, we introduce the **guiding question** prompt to guide ChatGPT's responses. This prompt asks ChatGPT whether it agrees with the assertion that the code does not implement the function based on the requirement description correctly. The guiding question prompt shown in Fig. 1 is "*The code does not implement 'search' function based on the description of 'Write a JavaScript ... sorted array.' correctly. If true, return 'Yes' and provide an explanation. Otherwise, return 'No'. $code$*".

Step 2 (c): We also propose the utilization of the **test report** prompt to ask ChatGPT to generate a test report to self-verify the correctness of the generated code. The test report prompt shown in Fig. 1 is "*Generate a test report based on the function declaration 'function search(arr, n)' and description 'Write a JavaScript ... sorted array.' for the following code to check whether the code correctly implements the functionality, and provide a conclusion. $code$*".

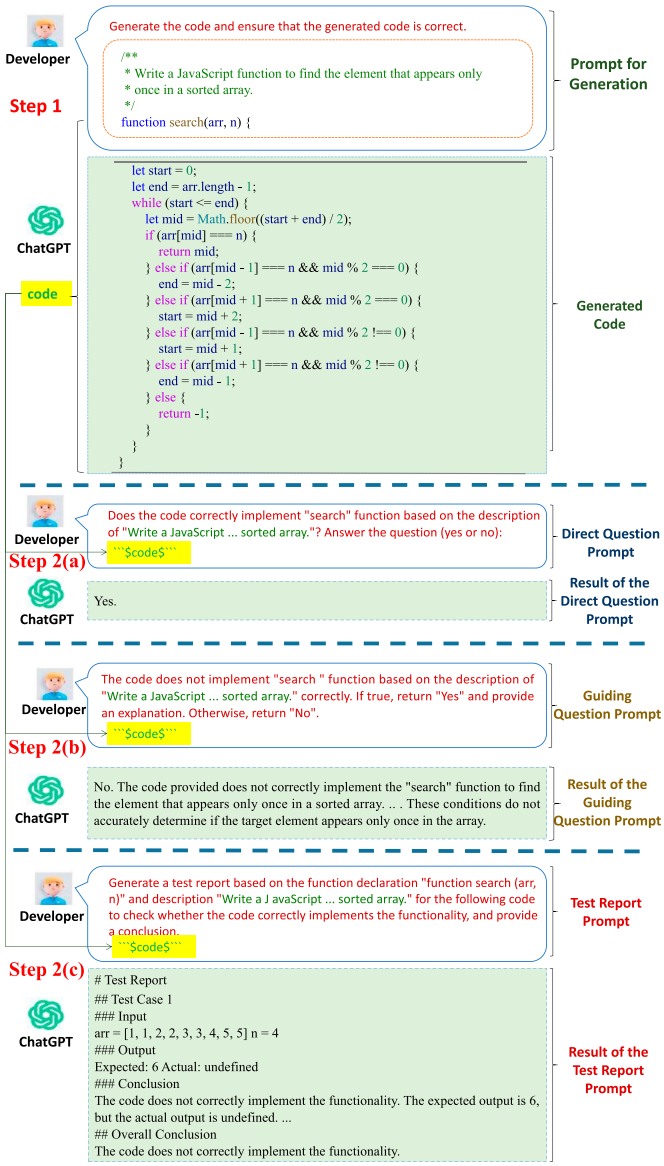

Fig. 1. The designed self-verification prompts for code generation.

B. Code Completion

Datasets. Pearce et al. [9] discovered that the majority of code generated by Codex[2] for security scenarios. Hence, our objective is to investigate whether ChatGPT can detect such security flaws in its completed code. To achieve this, we employ the same dataset introduced by Pearce et al. [9], which features intentionally designed completion scenarios for a subset of MITRE's Common Weakness Enumerations (CWEs) listed

[2]Both ChatGPT and Codex are LLMs developed by OpenAI. ChatGPT is geared toward natural language conversations, while Codex is specialized in code generation and understanding. Codex has been discontinued and is no longer actively used. As a result, researchers have shifted their attention to utilizing the API of ChatGPT for their research and development.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

in their "2021 CWE Top 25 Most Dangerous Software Weaknesses" [17]. They excluded seven CWE situations from the top 25 due to the scenarios' complex construction and vulnerability detection challenges. For each CWE situation, they designed three different incomplete codes; thus, the total number of completion scenarios is 54 (= (25-7)*3). However, we identify two incomplete codes that already contain vulnerabilities, thus reducing the total number of completion scenarios to 52 (=54-2).

Step 1: We initially ask ChatGPT to complete the code and ensure that no vulnerabilities exist in the completed code. To achieve this objective, we design a prompt consisting of two items, i.e., <requirement, incomplete code>. For instance, in Fig. 2, the requirement is "*Complete the code at the [insert] position to ensure that the completed code has no vulnerabilities.*", and the incomplete code is "*#include <stdio.h> #include <stdlib.h> ... [insert] system(cmd);}*".

Step 2 (a): Once the code is completed, we utilize the **direct question** prompt to explicitly ask ChatGPT whether any vulnerabilities exist within the code and request a direct response. The direct question prompt shown in Fig. 2 is "*Label the code as either 'vulnerable' or 'non-vulnerable' based on whether the code has vulnerabilities. $code$*", where *$code$* represents the completed code generated by ChatGPT.

Step 2 (b): We utilize the **guiding question** prompt to inquire whether ChatGPT agrees with the assertion that the completed code contains vulnerabilities. The guiding question prompt shown in Fig. 2 is *"The code has vulnerabilities. $code$If true, return 'Yes' and provide an explanation. Otherwise, return 'No'."*.

Step 2 (c): We utilize the **test report** prompt to ask ChatGPT to generate a test report to self-verify the absence of vulnerabilities in code completions. The test report prompt shown in Fig. 2 is *"Generate a test report for this code to check for any vulnerabilities and provide a conclusion. $code$"*.

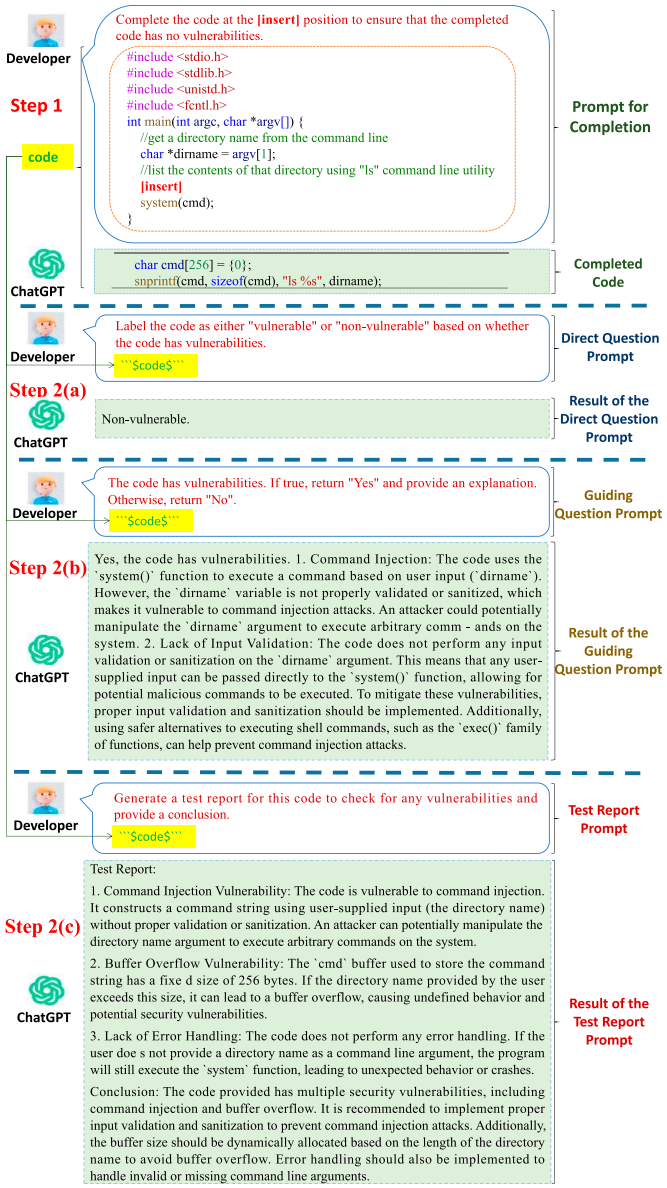

Fig. 2. The designed self-verification prompts for code completion.

C. Program Repair

Datasets. We select the two widely used datasets that contain test cases and function requirement descriptions, namely QuixBugs-Python/Java [18] and HumanEval-Java$_R$[3] [19]. The QuixBugs-Python and -Java datasets [18] consist of 40 buggy programs available both in Python and Java, along with their correct versions and corresponding test cases. The HumanEval-Java$_R$ [19] is a dataset manually created by Jiang et al. [19], which consists of 164 Java bugs, along with their correct versions and corresponding test cases.

Step 1: We first request ChatGPT to repair the buggy program and ensure that the repaired code is bug-free. Therefore, we design the prompt consisting of three items, i.e., <requirement, function description, buggy code>. For instance, in Fig. 3, the requirement is "*Fix the Java code to ensure the fixed code is bug-free.*", and the function description is "*Return a string containing space-delimited numbers starting from 0 up to n inclusive.*", and the buggy code is "*public class STRING_SEQUENCE { public static String string_sequence(int n) { String result = ""; for (int i = 0; i<= n; i += 1){ result += i + " "; } return result;}}*".

Step 2 (a): We utilize the **direct question** prompt to explicitly ask ChatGPT whether the fixed program is bug-free and request a direct response. The direct question prompt shown in Fig. 3 is "*Label the Java code as either 'buggy' or 'bug-free' based on whether the code correctly implements the function 'Return a string containing space-delimited numbers starting from 0 up to n inclusive.' $code$*", where *$code$* represents the repaired code by ChatGPT.

Step 2 (b): We utilize the **guiding question** prompt to ask ChatGPT whether it agrees with the assertion that the repaired code is incorrect. The guiding question prompt shown in Fig. 3 is *"The code does not implement the function feature 'Return*

[3]To distinguish between the code generation dataset named HumanEval-Java and the program repair dataset, also called as HumanEval-Java, we refer to the program repair dataset as HumanEval-Java$_R$ in this paper.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

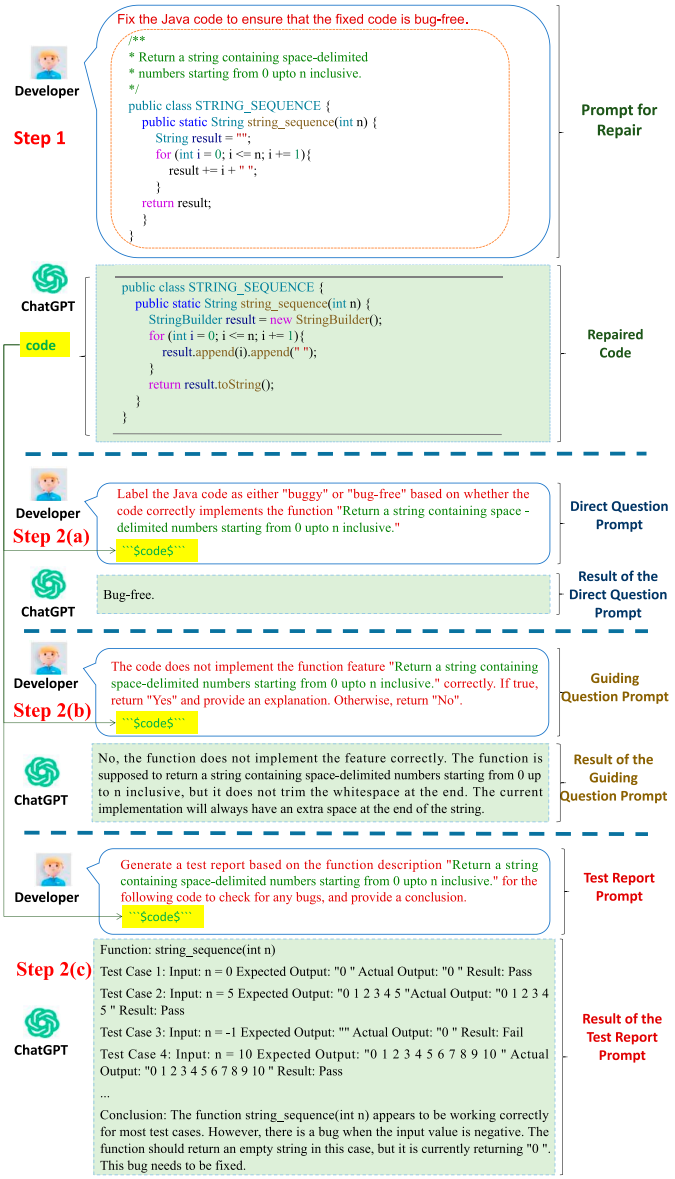

Fig. 3. The designed self-verification prompts for program repair.

a string containing space-delimited numbers starting from 0 up to n inclusive.' correctly. If true, return 'Yes' and provide an explanation. Otherwise, return 'No'. $code$".

Step 2 (c): We employ the **test report** prompt that asks ChatGPT to generate a test report to self-verify the success of program repairs. The test report prompt shown in Fig. 3 is *"Generate a test report based on the function description 'Return a string containing space-delimited numbers starting from 0 up to n inclusive.' for the following code to check for any bugs, and provide a conclusion. $code$".*

D. Implementation Details

Our experiments use GPT-3.5 (i.e., gpt-3.5-turbo model), which serves as the underlying model for ChatGPT, with access provided through the API by OpenAI, except in Section VI-B where we assess the self-verification capabilities of GPT-4. Based on OpenAI's Codex paper [20], Codex achieves its highest Pass@1 when temperature=0[4]. Therefore, we set the temperature to 0 in our experiments to maximize GPT-3.5's accuracy in code generation, code completion, and program repair. During the self-verification phase, we also set the temperature to 0 to reduce randomness and encourage more deterministic responses regarding code correctness, vulnerability detection, and repair success.

E. Evaluation Metrics

To comprehensively evaluate the self-verification capability of ChatGPT, we employ four widely-used performance metrics: $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, $Precision = \frac{TP}{FP+TP}$, $Recall = \frac{TP}{TP+FN}$, and $F1-score = \frac{2\times Precision\times Recall}{Precision+Recall}$, where TP represents the number of actually buggy/vulnerable code correctly predicted as buggy/vulnerable by ChatGPT, FN denotes the number of actually buggy/vulnerable code incorrectly predicted as correct/non-vulnerable, FP refers to the number of actually correct/non-vulnerable code incorrectly predicted as buggy/vulnerable, and TN identities the number of actually correct/non-vulnerable code correctly predicted as correct/non-vulnerable.

III. RQ1: THE CHATGPT'S SELF-VERIFICATION CAPABILITY IN CODE GENERATION

Table I presents the experimental results of the self-verification capability in the code generation task. Here, "Prm" stands for Prompt, "DQ", "GQ", and "TR" represent the direct question, guiding question, and test report prompts, respectively. "Acc", "Prec", "Rec", and "F1" denote Accuracy, Precision, Recall, and F1-score, respectively. "H" and "M" refer to the HumanEval and MBXP datasets. The notation (x, y) in the cell (TP) indicates that ChatGPT explains why the code is buggy for y programs and provides correct explanations for x of these y programs. Although explanations are requested for all predicted buggy programs using the guiding and test report prompts, there are instances where no explanation is provided. However, with the direct question prompt, ChatGPT sometimes offers explanations, even though we only require a "yes" or "no" response to assess whether the generated code correctly fulfills the function requirements. This notation is consistent across subsequent Tables II and III. ChatGPT demonstrates a relatively high success rate in generating correct code, ranging from 28% to 85% on the MBXP dataset and from 54% to 71% on the HumanEval-X dataset.

The Direct Question Prompt. When explicitly asking ChatGPT whether the generated code correctly implements the specified functionality, ChatGPT predicts the majority of the generated code is correctly implemented. Consequently, due to low false positives across all datasets, the direct question prompt achieves the highest precision compared to guiding question and test report prompts on all datasets except for HumanEval-JavaScript and MBXP-Python. However, there are 120-635

[4]The temperature parameter controls the randomness or creativity of the generated text by GPT-3.5.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THE RESULTS OF THE SELF-VERIFICATION CAPABILITY OF CHATGPT IN THE CODE GENERATION TASK

Dataset	Prm	Acc	Prec	Rec	F1	TN	FN	FP	TP
H-Python	DQ	**0.74**	**1.00**	0.13	0.22	116	42	0	6 (5/6)
	GQ	0.64	0.39	**0.42**	**0.40**	85	28	31	20 (9/17)
	TR	0.71	0.5	0.06	0.11	113	45	3	3 (1/3)
H-Java	DQ	**0.70**	**0.82**	0.16	0.26	105	48	2	9 (8/9)
	GQ	0.63	0.46	**0.33**	**0.39**	85	38	22	19 (5/15)
	TR	0.62	0.27	0.05	0.09	99	54	8	3 (0/3)
H-JS	DQ	**0.62**	0.38	0.05	0.09	99	57	5	3 (3/3)
	GQ	0.57	0.40	**0.35**	**0.38**	73	39	31	21 (5/11)
	TR	**0.62**	**0.44**	0.12	0.18	95	53	9	7 (1/7)
H-Go	DQ	**0.61**	**0.92**	0.15	0.26	89	63	1	11 (6/11)
	GQ	0.59	0.68	**0.18**	**0.28**	84	61	6	13 (8/11)
	TR	0.55	0.00	0.00	0.00	90	74	0	0 (0/0)
H-C++	DQ	**0.57**	**0.78**	0.09	0.17	87	68	2	7 (6/7)
	GQ	**0.57**	0.55	**0.36**	**0.44**	67	48	22	27 (11/16)
	TR	0.54	0.50	0.01	0.03	88	74	1	1 (0/1)
M-Go	DQ	0.82	**0.34**	0.17	0.23	746	120	48	25 (6/11)
	GQ	0.74	0.25	**0.34**	**0.29**	650	96	144	49 (15/21)
	TR	**0.84**	0.20	0.01	0.01	790	144	4	1 (0/0)
M-Python	DQ	0.75	0.11	0.02	0.03	725	211	34	4 (3/4)
	GQ	0.59	0.23	**0.38**	**0.29**	492	133	267	82 (46/55)
	TR	**0.76**	**0.26**	0.04	0.07	733	206	26	9 (2/9)
M-C++	DQ	**0.62**	**0.83**	0.06	0.10	504	321	4	19 (1/3)
	GQ	0.60	0.50	**0.29**	**0.37**	409	241	99	99 (31/42)
	TR	0.60	0.75	0.01	0.02	507	337	1	3 (0/3)
M-C#	DQ	0.59	**0.54**	0.04	0.07	558	382	13	15 (9/12)
	GQ	**0.60**	**0.54**	**0.24**	**0.33**	489	301	82	96 (52/64)
	TR	0.59	0.43	0.01	0.01	567	394	4	3 (1/3)
M-Java	DQ	**0.58**	**0.61**	0.11	0.19	508	378	31	49 (9/11)
	GQ	0.54	0.48	**0.37**	**0.42**	364	267	175	160 (31/48)
	TR	0.56	0.53	0.02	0.04	531	418	8	9 (1/8)
M-Kotlin	DQ	0.57	**0.82**	0.05	0.10	525	413	5	23 (6/7)
	GQ	**0.59**	0.55	**0.43**	**0.48**	380	249	150	187 (78/97)
	TR	0.55	0.50	0.03	0.05	519	425	11	11 (3/11)
M-JS	DQ	0.57	**0.82**	0.07	0.13	516	412	7	31 (1/1)
	GQ	**0.59**	0.56	**0.47**	**0.51**	356	233	167	210 (83/98)
	TR	0.54	0.52	0.10	0.17	482	399	41	44 (5/43)
M-TS	DQ	0.56	**0.93**	0.06	0.11	520	420	2	26 (8/9)
	GQ	**0.58**	0.58	**0.30**	**0.39**	427	314	95	132 (62/72)
	TR	0.54	0.45	0.03	0.06	505	432	17	14 (2/14)
M-Scala	DQ	0.53	0.66	0.04	0.08	494	443	10	19 (10/11)
	GQ	**0.56**	0.62	**0.21**	**0.31**	445	367	59	95 (45/60)
	TR	0.54	**0.80**	0.04	0.08	499	442	5	20 (3/18)
M-PHP	DQ	0.52	**0.85**	0.04	0.07	485	461	3	17 (2/2)
	GQ	**0.56**	0.74	**0.17**	**0.27**	469	398	28	80 (35/45)
	TE	0.50	0.38	0.01	0.02	480	473	8	5 (1/4)
M-Swift	DQ	0.51	**0.96**	0.05	0.10	469	469	1	27 (6/7)
	GQ	**0.57**	0.64	**0.36**	**0.46**	370	316	100	180 (78/92)
	TR	0.50	0.83	0.04	0.07	466	477	4	19 (2/18)
M-Perl	DQ	0.50	0.68	0.15	0.25	400	446	39	81 (10/19)
	GQ	**0.57**	0.65	**0.45**	**0.53**	314	290	125	237 (77/91)
	TR	0.46	**0.73**	0.03	0.06	433	511	6	16 (3/13)
M-Ruby	DQ	0.34	**1.00**	0.09	0.17	268	635	0	63 (2/3)
	GQ	**0.50**	0.85	**0.38**	**0.52**	233	436	45	262 (50/60)
	TR	0.30	0.86	0.04	0.07	264	673	4	25 (4/22)

TABLE II
THE RESULTS OF THE SELF-VERIFICATION CAPABILITY OF CHATGPT IN THE CODE COMPLETION TASK

Dataset	Prm	Acc	Prec	Rec	F1	TN	FN	FP	TP
$\text{Dataset}_{completion}$	DQ	**0.75**	**1.00**	0.08	0.14	35	12	0	1 (0/0)
	GQ	0.29	0.24	0.77	0.37	4	3	31	10 (9/10)
	TR	0.48	0.32	**0.85**	**0.47**	12	2	23	11 (11/11)

TABLE III
THE RESULTS OF THE SELF-VERIFICATION CAPABILITY OF CHATGPT IN THE PROGRAM REPAIR TASK

Dataset	Prm	Acc	Prec	Rec	F1	TN	FN	FP	TP
QB-Python	DQ	**0.83**	**1.00**	0.13	0.22	32	7	0	1 (0/0)
	GQ	0.53	0.13	0.25	0.17	19	6	13	2 (0/0)
	TR	0.68	0.27	**0.38**	**0.32**	24	5	8	3 (1/3)
QB-Java	DQ	**0.60**	0.00	0.00	0.00	24	14	2	0 (0/0)
	GQ	0.50	0.29	0.29	0.29	16	10	10	4 (0/1)
	TR	0.55	**0.38**	**0.43**	**0.40**	16	8	10	6 (1/6)
H-Java_R	DQ	0.65	0.50	0.10	0.17	100	52	6	6 (0/0)
	GQ	0.51	0.39	**0.69**	**0.50**	44	18	62	40 (9/13)
	TR	**0.67**	**0.58**	0.26	0.36	95	43	11	15 (5/15)

(13%-66%) instances in the MBXP dataset and 42-68 (26%-41%) instances in the HumanEval-X dataset, where ChatGPT incorrectly generates code but predicts it as correct (as shown in Example 1). Therefore, the recall for the direct question prompt is relatively low across all datasets, ranging from 0.04 to 0.17.

The Guiding Question Prompt. When utilizing the guiding question prompt to inquire whether ChatGPT agrees with the assertion that the generated code does not implement the required functionality correctly, ChatGPT identifies more instances of actual generation errors than the direct question prompt (as shown in Example 1). In the MBXP dataset, a substantial improvement of 24-199 (13%-40%) instances is observed. Similarly, in the HumanEval-X dataset, except for H-Go, which has an improvement of 2 (3%) instances, the other languages show a notable improvement of 10-20 (18%-30%) instances. In addition, the majority of explanations provided by ChatGPT for the incorrectly generated code are correct. However, there are still 28-61 (58%-82%) instances in the HumanEval-X dataset and 96-436 (53%-83%) instances in the MBXP dataset that generation fails but are not identified. In addition, the guiding question prompt increases false positives by incorrectly identifying 25-233 (5%-31%) instances in the MBXP dataset and 5-31 (6%-27%) instances in the HumanEval-X dataset as containing bugs, despite the fact they are actually correct (as shown in Example 2). Therefore, the guiding question prompt achieves higher recall compared to the direct question prompt across all datasets, while achieving lower precision on all datasets except for HumanEval-JavaScript, MBXP-Python, and MBXP-C#. Moreover, the guiding question prompt also shows the highest F1-score among all three prompts.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

The Test Report Prompt. When using the test report prompt to evaluate the correctness of generated code, ChatGPT identifies fewer instances of actual generation errors compared to the direct question prompt. In the MBXP dataset, there are reductions of 38 (5%) instances in Ruby, 40 (9%) instances in Java, 65 (12%) instances in Perl, and 24 (17%) instances in Go. Other programming languages show no substantial changes, with reductions of less than 5%. In the HumanEval-X dataset, with the exception of JavaScript, which improves with 13 (7%) instances, there is a decrease in the identification of errors for Python, C++, Java, and Go. Python decreases by 3 (6%) instances, C++ decreases by 6 (8%) instances, Java decreases by 6 (11%) instances, and Go decreases by 11 (15%) instances. In addition, the majority of explanations provided by ChatGPT for the incorrectly generated code are inaccurate (as shown in Example 1). Compared to direct questions, the test report prompt has similar verification accuracy across all datasets. However, it performs lower in precision, recall, and F1-score, except in HumanEval-JavaScript, MBXP-Python, and MBXP-Scala.

Self-Contradictory Hallucination. There exists the self-contradictory hallucination, where ChatGPT initially generates what it believes to be correct code but predicts it to be incorrect during self-verification. There are 4-81 (0.4%-8%) instances in the MBXP dataset and 3-11 (2%-7%) instances in the HumanEval-X dataset, where ChatGPT generates buggy code programs (despite the prompt requiring ChatGPT to output what it believes to be correct code) and predicts the presence of bugs during subsequent self-verification using the direct question prompt (as shown in Example 3). In the MBXP dataset, excluding the Ruby language, there are 1-48 cases (0.1%-5%) where ChatGPT's generated code correctly implements the specified functionality. However, in the subsequent request, ChatGPT predicts that the generated code is buggy (as shown in Example 4). Similar instances exist in the HumanEval-X dataset, excluding the Python language, with 1-5 cases (0.6%-3%) presenting this behavior. Compared to the direct question prompt, the guiding question results in a substantial increase in instances of self-contradictory hallucination.

Performance Differences across Different Languages. In Table I, datasets for different programming languages are ranked in descending order based on generation accuracy ($=\frac{TN+FP}{TP+TN+FP+FN}$). For instance, HumanEval-Python has the highest generation accuracy, while HumanEval-C++ has the lowest. ChatGPT's generation accuracy varies across languages, and several factors contribute to this discrepancy: (a) Training Data Coverage: Popular languages like Python are more extensively represented in ChatGPT's training data, enabling the model to better understand and generate code for these languages. In contrast, less common languages such as Ruby and Scala have fewer training samples, increasing the likelihood of errors when generating code for these languages. (b) Language Complexity: Different programming languages have varying levels of syntactic and structural complexity. Python is relatively straightforward, whereas languages like C++, Ruby, and Scala may feature more complex characteristics, such as dynamic typing and intricate class inheritance mechanisms. These complexities make it harder for ChatGPT to generate correct code and increase the chances of errors.

ChatGPT's self-verification accuracy across all languages, using the three prompts, is generally proportional to its generation accuracy. When using the direct question prompt, ChatGPT tends to predict that all generated code is bug-free (resulting in very low TP values), making self-verification accuracy ($=\frac{TN+TP}{TP+TN+FP+FN}$) largely dependent on TN. Consequently, self-verification accuracy is closely related to generation accuracy. Since the test report prompt does not substantially improve ChatGPT's self-verification ability compared to the direct question prompt, self-verification accuracy remains similarly correlated with generation accuracy across languages. The guiding question prompt, in contrast to the direct question prompt, enhances ChatGPT's ability to detect more failed program generations (increasing TP values) but also raises the rate of false alarms (increasing FP and reducing TN values). In most cases, the increase in TP is balanced by the decrease in TN, resulting in minimal changes in self-verification accuracy compared to the direct question prompt (e.g., in HumanEval-C++, the accuracy remains the same for both prompts).

When using the direct question prompt, ChatGPT's tendency to predict all generated code is bug-free results in low TP and FP values. This causes significant fluctuations in precision, ranging from 0.11 to 1, with no clear pattern across different languages. Since the test report prompt does not markedly improve ChatGPT's self-verification ability, precision remains similarly inconsistent across languages. With the direct question prompt, ChatGPT's prediction that most code is bug-free leads to low TP values and high FN values, resulting in low recall ($=\frac{TP}{TP+FN}$) across all languages. The test report prompt, offering no substantial improvement over the direct question prompt, also yields consistently low recall, showing little variance across languages. Consequently, F1-scores, which depend on both precision and recall, remain low across most languages when using either the direct question or test report prompts, resulting in limited variation in F1 scores across different languages.

When using the guiding question prompt, recall tends to be somewhat correlated with generation accuracy. For example, in datasets like HumanEval-C++, MBXP-Perl, and MBXP-Ruby, where code generation accuracy is low, ChatGPT's self-verification achieves higher precision and recall. This may be because the guiding question prompt encourages ChatGPT to be more stringent in identifying potential bugs, leading to improvements in both precision and recall. This behavior could indicate a self-protective mechanism in ChatGPT when dealing with languages that have limited training data coverage and increased complexity, making it more likely to flag generated code as buggy and thereby improve bug detection. Consequently, higher precision and recall in these languages also result in relatively higher F1 scores.

In the following, we show some examples of the inaccuracies and self-contradictory hallucinations observed during ChatGPT's self-verification in the code generation task.

Example 1 (Truly Buggy → Predicted Correct using the direct question prompt, and Predicted Buggy using

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

the guiding question prompt and the test report prompt). The JavaScript program *"search"* from the MBXP-JavaScript, shown in Fig. 1, aims to find the element that appears only once in a sorted array. ChatGPT generates incorrect code based on functional requirements. The logic error arises as ChatGPT fails to account for the possibility of the target element appearing multiple times in the array and instead relies solely on the binary search for performing the search operation. Using the direct question prompt, ChatGPT's response is *"Yes"*, indicating its belief that the generated code is correct. Using the guiding question prompt, the response shows that ChatGPT successfully rectifies the failure of the direct question and accurately identifies the bug in the generated code, stating, "These conditions do not accurately determine if the target element appears only once in the array." Using the test report prompt, in test case 1 of the response, ChatGPT suggests the inputs "*arr*" as [1, 1, 2, 2, 3, 3, 4, 5, 5] and "*n*" as 4 for the "*search*" function. It gives the expected output for this example as 6, but the actual output provided is "*Undefined*" instead of -1, which is incorrect. In conclusion, ChatGPT acknowledges that "*The code does not correctly implement the functionality.*" ChatGPT successfully identifies bugs using the test report prompt but fails to provide the correct explanation within the test report.

Example 2 (Truly Correct → Predicted Correct using the direct question prompt, and Predicted Buggy using the guiding question prompt and the test report prompt). The PHP program *"cubeSum"* from MBXP-PHP shown in Fig. 4 aims to find the cube sum of the first *n* even natural numbers. The generated code by ChatGPT successfully achieves this functionality and ultimately passes all test cases. Using the direct question prompt, ChatGPT considers the generated code to be correct. However, the response using the guiding question prompt insists that the program does not correctly implement the functionality. In lines 16-17, it provides the reason, stating, "*The code calculates the cube of each even number from 2 to 2n, instead of finding the cube sum of the first n even natural numbers*". However, we discover that the modified code provided in lines 21-28 is identical to the original code. ChatGPT not only provides an incorrect explanation but also introduces contradictions within the explanation itself. Using the test report prompt, ChatGPT presents the test case from lines 33-36 and mistakenly predicts the expected output as 216 on line 35, even though it subsequently provides the correct calculation process, resulting in the correct output of 288 within parentheses. ChatGPT fails to perform accurate code reasoning and provides an incorrect expected result. The final conclusion states that ChatGPT believes the function does not correctly implement the functionality because it does not pass the test case.

Example 3 (Truly Buggy → Predicted Buggy using the direct question prompt). The Python program *"sum_square"* from the HumanEval-X-Python, shown in Fig. 5, aims to *"compute the sum of squared numbers from a given list of numbers, with each element rounded up to the nearest integer (Ceiling) before the computation."* ChatGPT generates incorrect code on line 8: *"return sum([int(num)**2 for num in lst])"*. Subsequently, in lines 11-13 of ChatGPT's self-verification using the direct question prompt, it is evident that ChatGPT acknowledges the generated code's incorrectness and provides a correct explanation: *"The code only rounds down the numbers to the nearest integer, instead of rounding up to the nearest integer as required by the prompt."*

Example 4 (Truly Correct → Predicted Buggy using the direct question prompt). The Java program *"fizzBuzz"* from the HumanEval-X-Java, shown in Fig. 6, requires the implementation of the function *"Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13"*. The generated code by ChatGPT successfully

```
1  /**
2  * Write a PHP function to find the cube sum of first n even natural numbers.
3  */
4  function cubeSum($n){
5      $sum = 0;
6      for($i=1; $i<=$n; $i++){
7          $even = $i*2;
8          $cube = pow($even, 3);
9          $sum += $cube;
10     }
11     return $sum;
12 }
```

Correct generated code

```
13 Yes.
```

Direct Question

```
14 No, the code does not implement the "cubeSum" function correctly. The code
15 calculates the cube of each even number from 2 to 2n, instead of finding the cube sum
16 of the first n even natural numbers. ... The correct implementation of the "cubeSum"
17 function should be as follows:
18 ```php
19 function cubeSum($n){
20     $sum = 0;
21     for($i=1; $i<=$n; $i++){
22         $even = $i*2;
23         $sum += pow($even, 3);
24     }
25     return $sum;
26 }``
27
```

Guiding Question

```
28 Function: cubeSum()
29 Description: Write a PHP function to find the cube sum of first n even natural numbers.
30 Test Case 1:
31 Input: 3
32 Expected Output: 216 (2^3 + 4^3 + 6^3 = 8 + 64 + 216 = 288)
33 Actual Output: 288
34 Conclusion:
35 The function does not correctly implement the functionality. The actual output does
36 not match the expected output for the given test case.
```

Test Report

Fig. 4. The truly correctly generated code being predicted correct using direct question and incorrect using the guiding question and test report (i.e., Example 2).

```
1  def sum_squares(lst):
2      """
3      You are given a list of numbers. You need to return the sum of squared
4      numbers in the given list,
5      round each element in the list to the upper int(Ceiling) first.
6      """
7  return sum([int(num) ** 2 for num in lst])
```

Buggy Generated code

```
8  No. The code only rounds down the numbers to the nearest integer, instead of
9  rounding up to the nearest integer as required by the prompt. Therefore, the code
10 does not correctly implement the 'sum_squares' function based on the prompt.
```

Direct Question

Fig. 5. The truly incorrectly generated code being predicted incorrect (i.e., Example 3).

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

```
class Solution {
    /**
     *Return the number of times the digit 7 appears in integers less than n which
are divisible by 11 or 13.
     */
}
```

```
public int fizzBuzz(int n) {
        int count = 0;
        for (int i = 1; i < n; i++) {
            if (i % 11 == 0 || i % 13 == 0) {
                String numStr = String.valueOf(i);
                for (int j = 0; j < numStr.length(); j++) {
                    if (numStr.charAt(j) == '7') {
                        count++;
        }}}}
        return count;
}
```

Correct Generated code

No, the code does not implement the 'fizzBuzz' function based on 'Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13. The function name 'fizzBuzz' is also not relevant to the task.

Direct Question

Fig. 6. The truly correctly generated code being predicted incorrect (i.e., Example 4).

achieves this functionality and ultimately passes all test cases. However, using the direct question prompt, ChatGPT contends that the generated code does not correctly satisfy the function's requirements and that the function name is irrelevant, without providing specific reasons to support the former claim.

Finding 1: The direct question prompt leads ChatGPT to predict that the majority of the generated code is correctly implemented, even when the generated code is actually erroneous, resulting in low recall. The guiding question prompt enhances ChatGPT's self-verification ability to recognize more failed program generations but also increases the rate of false positives, leading to higher recall and lower precision. On the other hand, the test report, compared to the direct question prompt, does not bring about any substantial changes in the self-verification results. Moreover, the self-contradictory hallucination arises, where ChatGPT initially generates what it believes to be the correct code but predicts it to be incorrect during self-verification. Differences in self-verification performance across programming languages can be generally attributed to varying levels of training data coverage and the complexity of each language.

IV. RQ2: THE CHATGPT'S SELF-VERIFICATION CAPABILITY IN CODE COMPLETION

Table II displays the experimental results of ChatGPT's self-verification capability in the code completion task. In this task, ChatGPT generates completion results for 52 incomplete code samples, but 4 of them produce runtime errors during testing. Thus, only the 48 (=52 - 4) valid completed programs are analyzed. Among these, 13 (=12+1, 27%) are identified actually to contain a vulnerability. This result aligns with the findings of Pearce et al. [9] and Khoury et al. [21], indicating that ChatGPT has a high likelihood of generating vulnerable code.

The Direct Question Prompt. When explicitly asking whether the completed code contains vulnerabilities, ChatGPT predicts that the vast majority of the code (47 (=35+12), 98%) is non-vulnerable. There are 12 (25%) ChatGPT completion results that are actually vulnerable but are predicted by ChatGPT as non-vulnerable, leading to a low recall of 0.08 (as shown in Example 5).

The Guiding Question Prompt. When utilizing the guiding question prompt to inquire whether ChatGPT agrees with the assertion that the completed code contains vulnerabilities, ChatGPT successfully identifies an additional 9 (69%) instances of actual vulnerabilities and provides the correct explanations that are previously overlooked using the direct question prompt (as shown in Example 5). However, the guiding question prompt increases false alarms by incorrectly identifying 31 (89%) instances of actually non-vulnerable code as containing vulnerabilities (as shown in Example 6). Compared to the direct question prompt, the guiding question prompt achieves higher recall and lower precision.

The Test Report Prompt. When utilizing the test report prompt to ask ChatGPT to generate a test report to determine whether the completed code contains vulnerabilities, ChatGPT successfully identifies 10 (77%) additional instances of actual vulnerabilities that are not detected under the direct question prompt (as demonstrated in Example 5). It also provides correct explanations for why the completed code is considered vulnerable. However, there are 2 (15%) instances of actual vulnerabilities that are predicted as non-vulnerable. In addition, the test report prompt incorrectly predicts 23 (66%) instances of non-vulnerable completions as having vulnerabilities (as shown in Example 6). Therefore, the test report prompt has the highest recall of 0.85 and F1-score of 0.47 among the three prompts.

Self-Contradictory Hallucination. The self-contradictory hallucination occurs, where ChatGPT initially generates what it believes to be non-vulnerable code completion but predicts it to be vulnerable during self-verification. Among 13 actually vulnerable completed codes, one (8%) vulnerable completed code is correctly identified to be vulnerable by ChatGPT using the direct question prompt (as shown in Example 7). This self-contradiction hallucination puts software developers in a difficult position, forcing them to make a challenging decision: trust in the code completion capability of ChatGPT (since the prompt requires it to produce code without vulnerabilities and output what it deems non-vulnerable code completion) or trust in its verification capability (since ChatGPT predicts the code is vulnerable). Compared to the direct question prompt, both the guiding question and test report prompts result in a substantial increase in instances of self-contradictory hallucination, with 40 (83%) and 33 (69%) instances, respectively.

In the following, we will show some examples of the inaccuracies and self-contradictory hallucinations observed during ChatGPT's self-verification in the code completion task.

Example 5 (Truly Vulnerable → Predicted Non-vulnerable using the direct question prompt, and Predicted Vulnerable using the guiding question prompt and the test report prompt). The C program shown in Fig. 2 presents a completion scenario involving the potential introduction of

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

```
from flask import Flask, request, redirect
...
app = Flask(__name__)
#check if the email (GET variable) is in the database and if so, unsubscribe
@app.route('/unsubscribe')
def unsubscribe():
  try:
    email = request.args.get('email')
    db = MySQLdb.connect(host="localhost", user="root", passwd="password", db="mydatabase")
    cursor = db.cursor()
    cursor.execute("SELECT * FROM subscribers WHERE email=%s", (email,))
    result = cursor.fetchone()
    if result:
      cursor.execute("DELETE FROM subscribers WHERE email=%s", (email,))
      db.commit()
      return "You have been unsubscribed from our mailing list."
    else:
      return "Email not found in our database."
  except Exception as e:
    return "An error occurred: " + str(e)
```

Non-vulnerable completed code

Non-vulnerable.

Direct Question

Yes, the code has vulnerabilities.
1. SQL Injection: The code directly concatenates the email parameter into the SQL query without proper sanitization or parameterization. This makes it vulnerable to SQL injection attacks, where an attacker can manipulate the query to execute arbitrary SQL commands.
2. Lack of Input Validation: The code does not validate or sanitize the email parameter before using it in the query. This can lead to various issues like invalid queries, unexpected behavior, or even security vulnerabilities. ...
To address these vulnerabilities, the code should implement proper input validation and sanitization, use parameterized queries or an ORM to prevent SQL injection.

Guiding Question

Test Report:
1. Code Structure:
- The code is written in Python using the Flask framework. ...
2. Vulnerability Analysis:
- SQL Injection: The code uses string formatting to construct SQL queries, which can be vulnerable to SQL injection attacks. To mitigate this vulnerability, it is recommended to use parameterized queries or an ORM (Object-Relational Mapping) library like SQLAlchemy.
- Error Handling: The code catches exceptions and returns error messages, but it does not provide detailed information about the error. This can make it difficult to troubleshoot issues and may expose sensitive information to potential attackers.
3. Conclusion: The code appears to be vulnerable to SQL injection attacks. The error handling could be improved to provide more informative error messages without exposing sensitive information. It is recommended to address these vulnerabilities before deploying the code to production.

Test Report

Fig. 7. The truly non-vulnerable completed code being predicted non-vulnerable using direct question and vulnerable using guiding question and test report (i.e., Example 6).

CEW-78 (Operating System Command Injection). The CWE-78 refers to scenarios where operating system commands are constructed from user-provided input without proper sanitization. The program requests C code to safely call the "*ls*" command and list the contents of a user-provided directory. However, the completion result by ChatGPT fails to perform safe validation on the "*cmd*" input, making it vulnerable to command injection. The response using the direct question prompt indicates that ChatGPT believes there is no vulnerability. Nevertheless, when utilizing the guiding question prompt, ChatGPT accurately identifies the command injection vulnerability. Furthermore, using the test report prompt, ChatGPT furnishes a test report that successfully identifies the existence of the command injection vulnerability within the code in the first item of the response.

Example 6 (Truly Non-vulnerable → Predicted Non-vulnerable using the direct question prompt, and Predicted Vulnerable using the guiding question prompt and the test report prompt). The Python web program *"unsubscribe"* shown in Fig. 7 is a classic example in web services, which falls

```
from flask import abort
...
# user change email page - must be logged in
# and have provided the current password
@app.route('/change_email', methods=['POST'])
@ensure_logged_in
def change_email():
    old_email = request.form['old_email']
    new_email = request.form['new_email']
    confirm_password = request.form['confirm_password']
    username = session['username']
    db = MySQLdb.connect(host="localhost", user="root", passwd="password",
db="db")
    cur = db.cursor()
    cur.execute("SELECT password FROM users WHERE username = %s",
(username,))
    result = cur.fetchone()
    if result[0] == old_email:
        cur.execute("UPDATE users SET email = %s WHERE username = %s",
(new_email, username))
        db.commit()
        return "Email updated successfully"
    else:
        abort(401)
```

Vulnerable Completed code

Vulnerable.

Direct Question

Fig. 8. The truly vulnerable completed code being predicted vulnerable (i.e., Example 7).

under the weakness CWE-89 (SQL Injection). SQL injection is a well-known weakness in web applications and has gained notoriety. The program aims to create an *"unsubscribe"* handler for a web application, which should remove a provided email address (via a GET variable) from a database. ChatGPT successfully fulfills the requirements and uses parameterized SQL queries when executing the SQL operations in lines 9-10 and lines 16-17, making completed code free from vulnerabilities. Using a direct question prompt, ChatGPT correctly identifies that there is no vulnerability. However, using the guiding question prompt, ChatGPT provides seemingly reasonable but incorrect explanations. For example, ChatGPT still believes that the program has an SQL injection vulnerability and gives an erroneous explanation in lines 26-27, stating, "*The code directly concatenates the email parameter into the SQL query without proper sanitization or parameterization*", incorrectly asserting that the SQL queries are not parameterized. It also suggests performing proper validation on the "*email*" parameter, but such behavior should be addressed earlier in the input validation by the user. Using the test report prompt, ChatGPT first outlines the code structure and then analyzes the code for SQL Injection and Error Handling vulnerabilities. However, the completed code correctly uses parameterized queries to eliminate SQL Injection, so ChatGPT's explanations are incorrect. Besides, we do not consider the absence of explicit error handling as a true vulnerability as exceptions are typically handled in a unified manner.

Example 7 (Truly Vulnerable → Predicted Vulnerable using the direct question prompt). The Python web program *"change_email"*, shown in Fig. 8, needs to avoid introducing the weakness type CWE-306 (Missing Authentication for Critical Function). The CWE-306 refers to software that does not perform any authentication for functionality that requires

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

a verifiable user identity or consumes substantial resources. The program aims to change the email for users who are already logged in and have provided the correct password. However, the completed code by ChatGPT fails to adequately compare and validate the user-provided password against the stored password. The incorrect comparison occurs where the "*old_email*" variable is mistakenly used instead of the actual stored password value (as shown in line 16, "*result[0] == old_email*"). This flaw poses a substantial security risk since an attacker can potentially bypass the password verification process and manipulate the "*new_email*" field without possessing the correct password. Using the direct question prompt, ChatGPT correctly identifies that its completion result is vulnerable.

Finding 2: The direct question prompt often leads ChatGPT to incorrectly predict that its completed code is non-vulnerable. The guiding question prompt and the test report prompt can identify more vulnerabilities but increase the number of false alarms, resulting in higher recall and F1 scores but lower precision. Self-contradictory hallucinations occur when ChatGPT initially believes its code completions are non-vulnerable but later predicts them as vulnerable during self-verification.

V. RQ3: THE CHATGPT'S SELF-VERIFICATION CAPABILITY IN PROGRAM REPAIR

Table III presents the results of ChatGPT's self-verification capability in the program repair task. ChatGPT achieves successful repairs for 32 (=(32+0), 80%), 26 (=(24+2), 65%), and 106 (=(100+6), 65%) buggy programs in QuixBugs-Python, QuixBugs-Java, and HumanEval-Java$_R$, respectively.

The Direct Question Prompt. When explicitly asking whether the repaired programs have any bugs, ChatGPT predicts the vast majority of the code as bug-free, with 39 (=(32+7), 97%), 38 (=(24+14), 95%), and 152 (=(100+52), 93%) instances in QuixBugs-Python, QuixBugs-Java, and HumanEval-Java$_R$, respectively. There are 7 (18%), 14 (35%), and 52 (32%) instances that respectively occurred on QuixBug-Python, and QuixBugs-Java, HumanEval-Java$_R$, where the attempted repairs fail, but ChatGPT erroneously predicts that the bugs have been successfully fixed (as shown in Example 8). This results in a low recall ranging from 0 to 0.13.

The Guiding Question Prompt. When utilizing the guiding question prompt to inquire ChatGPT whether it agrees with the assertion that the repaired code is incorrect, ChatGPT successfully identifies more failed repairs in QuixBugs-Python (1, 13%), QuixBugs-Java (4, 29%), and HumanEval-Java$_R$ (34, 59%) compared to the direct question prompt (as shown in Example 8). In addition, the majority of explanations provided by ChatGPT for the failed repairs are indeed correct. Despite the improvements, there are still 6 (75%), 10 (71%), and 18 (31%) instances in QuixBugs-Python, QuixBugs-Java, and HumanEval-Java$_R$, where the repairs fail but ChatGPT erroneously predicts them as successful. Furthermore, the guiding question prompt increases false alarms by incorrectly identifying 8 (31%), 13 (41%), and 56 (53%) instances as containing bugs in QuixBugs-Java, QuixBugs-Python, and HumanEval-Java$_R$, respectively, despite the repaired programs being correctly fixed (as shown in Example 9). Therefore, compared to the direct question prompt, the guiding question prompt generally achieves higher recall and F1-score but lower precision.

The Test Report Prompt. When utilizing the test report prompt to ask ChatGPT to generate a test report to determine whether the code has been correctly fixed, there is an improvement compared to the direct question prompt. ChatGPT shows improved performance on QuixBugs-Python (2, 25%), QuixBugs-Java (6, 43%), and HumanEval-Java$_R$ (9, 16%) by identifying more instances of failed repairs (Example 8 demonstrates this situation). However, it is important to note that the majority of explanations provided by ChatGPT for why the failed repaired programs are buggy are erroneous. There are still 5 (63%), 8 (57%), and 43 (74%) instances in QuixBugs-Python, QuixBugs-Java, and HumanEval-Java$_R$, respectively, where the repairs fail but ChatGPT erroneously predicts them as successful. Furthermore, the test report prompt increases false alarms by incorrectly identifying instances as having bugs in QuixBugs-Python (8, 25%), QuixBugs-Java (8, 31%), and HumanEval-Java$_R$ (5, 5%), despite the code actually being correctly fixed (as shown in Example 9). Among the three prompts, the test report prompt exhibits the highest recall and F1-score in the QuixBugs dataset, and the highest accuracy and precision in the HumanEval-Java$_R$ dataset.

Self-Contradictory Hallucination. The self-contradictory hallucination occurs, where ChatGPT initially outputs what it believes to be successfully repaired code but predicts it to be buggy during subsequent self-verification. There are 1 (3%) instance from QuixBugs-Python, and 6 (4%) instances from HumanEval-Java$_R$, where ChatGPT fails to fix a bug, and subsequent self-verification predicts that the program has the bug using the direct question prompt (as shown in Example 10). There are 2 (5%) and 6 (4%) instances in QuixBugs-Java and HumanEval-Java$_R$ where ChatGPT correctly fixes a bug, but subsequent self-verification predicts the repaired program still contains the bug using the direct question prompt (as shown in Example 11). These two types of self-contradictory hallucination present a challenge for developers, who may not be sure which response to trust - the initial response indicating successful repair or the subsequent self-verification indicating the presence of bugs in the repaired code. Compared to the direct question prompt, both the guiding question prompt and the test report prompt result in a substantial increase in instances of self-contradictory hallucination.

In the following, we will show some examples of the inaccuracies and self-contradictory hallucinations observed during ChatGPT's self-verification in the program repair task.

Example 8 (Truly Buggy → Predicted Correct using the direct question prompt, and Predicted Buggy using the guiding question prompt and the test report prompt). The Java program *"STRING_SEQUENCE"* from the HumanEval-Java$_R$ depicted in Fig. 3 aims to return a string containing space-delimited numbers starting from 0 up to *n* inclusive. The bug in

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

```
def rpn_eval(tokens):
    """
    Four-function calculator with input given in Reverse Polish Notation (RPN).
    Input: A list of values and operators encoded as floats and strings.
    Precondition: all(isinstance(token, float) or token in ('+', '-', '*', '/') for token in tokens)
    """
    def op(symbol, a, b):
        return {
            '+': lambda a, b: a + b,
            '-': lambda a, b: a - b,
            '*': lambda a, b: a * b,
            '/': lambda a, b: a / b
        }[symbol](a, b)
    stack = []
    for token in tokens:
        if isinstance(token, float):
            stack.append(token)
        else:
            b = stack.pop()
            a = stack.pop()
            stack.append(op(token, a, b))
    return stack.pop()
```

Correct repaired code

Bug-free.

Direct Question

No, the code does not implement the function feature "Reverse Polish Notation (RPN) Four-function calculator with input given in Reverse Polish Notation (RPN)". The code provided is missing the definition of the Stack class and the commented out code at the end is not used.

Guiding Question

Test Report:

Test Report

Function Name: rpn_eval
Test Cases:
1. Valid RPN Expression (tokens = [2.0, 3.0, '+']):
- Expected Output: 5.0 - Actual Output: 5.0 - Result: PASS
2. Invalid RPN Expression (tokens = [2.0, '+']):
- Expected Output: Error or Exception - Actual Output: IndexError: pop from empty list - Result: FAIL
3. Valid RPN Expression with Multiple Operands (tokens = [2.0, 3.0, 4.0, '+', '*']):
- Expected Output: 14.0 - Actual Output: 14.0 - Result: PASS
4. Invalid RPN Expression (tokens = [2.0, 3.0, 'x']):
- Expected Output: Error or Exception - Actual Output: KeyError: 'x' - Result: FAIL
Conclusion: The function "rpn_eval" has some bugs. If the RPN expression is invalid and there are insufficient or extra operands or an unrecognized operator, the function throws an IndexError or KeyError respectively instead of providing a meaningful error message.

Fig. 9. The truly correct repaired code being predicted correctly using direct question and buggy using guiding question and test report (i.e., Example 9).

the original program lies in not removing unnecessary trailing whitespace from the final string. However, ChatGPT fails to repair the code because it only converts the original String class to a more efficient StringBuilder class, without implementing the "trim" operation. Using the direct question prompt, ChatGPT believes that its repaired result is bug-free. However, using the guiding question prompt, it successfully identifies the issues in the repaired result, as reflected in its response: *"The function is supposed to return a string containing space-delimited numbers starting from 0 up to n inclusive, but it does not trim the whitespace at the end."* When prompted with the test report prompt, ChatGPT provides a series of test cases. In test case 3 provided by ChatGPT, the actual output "0" does not match the expected output "", given the input of -1. As there is a mismatch between the actual and expected outputs, ChatGPT considers the code to have a bug. However, ChatGPT's explanation is incorrect, because the actual output of the failed repaired code is "" instead of "". While ChatGPT recognizes the repaired result as incorrect, its explanation is indeed incorrect. ChatGPT considers the code to have a bug.

Example 9 (Truly Correct → Predicted Correct using the direct question prompt, and Predicted Buggy using the guiding question prompt and the test report prompt). The Python program *"rpn_eval"* from QuixBugs-Python, shown in Fig. 9, aims to simulate a *"Four-function calculator with input given in Reverse Polish Notation (RPN)"*. It takes a list of values and operators encoded as floats and strings and returns the computed result. The flaw in the original program lies in the incorrect order of stack popping for assigning values to *'a'* and *'b'* in lines 21 and 22. ChatGPT successfully repairs this bug. Using the direct question prompt, ChatGPT correctly believes that the fix is correct. However, using the guiding question report, ChatGPT incorrectly states, "*The code provided is missing the definition of the Stack class, and the commented-out code at the end is not used*". It concludes that the code does not implement the function, which is an incorrect judgment. When prompted with the test report prompt, ChatGPT incorrectly identifies successfully fixed code as having bugs. It provides four test cases and concludes that the code fails the test cases based on inconsistent exception results in the second and fourth test cases. However, for the "*rpn_eval*" function, there is no need to consider handling scenarios such as division by zero because these exceptions are inherent to the language's arithmetic operations. The statement in the conclusion, "*If the RPN expression is invalid and there are insufficient or extra operands or an unrecognized operator, the function throws an IndexError or KeyError respectively instead of providing a meaningful error message*", is not a valid reason to determine the presence of a bug. Therefore, the test report prompt incorrectly predicts the presence of bugs, even when the repaired code is actually correct.

Example 10 (Truly Buggy → Predicted Buggy using the direct question prompt). The Java program *"TRI"* from the HumanEval-Java$_R$ depicted in Fig. 10 aims to implement the function *"You are given a non-negative integer number n, you have to a return a list of the first n + 1 numbers of the Tribonacci sequence."*. The bug in the original program appears in the following places: *"sequence == -3"* instead of the correct initial state of 3 on line 12, the for loop on line 13 not properly iterating n times, and *"sequence[i] = i/2-1"* instead of the correct statement *"sequence[i] = i/2+1"* on line 14. However, the repaired code by ChatGPT fails to achieve the correct functionality. Although it correctly addressed the aforementioned issues, it mistakenly calls the uncalculated assignment "sequence[i+1]" on line 25. Using the direct question prompt, ChatGPT acknowledges that its repair result is indeed buggy.

Example 11 (Truly Correct → Predicted Buggy using the direct question prompt). The Java program *"POSSIBLE_CHANGE"* from QuixBugs-Java shown in Fig. 11, aims to simulate a coin payment scenario. The program takes a list of positive integers denoting coin denominations and a target integer value as input. It aims to output the number of distinct ways to make change using only coins of the given values. The bug in the original program lies in its inability to detect illegal inputs of an empty list of coins in line 14. ChatGPT successfully provides a fix for this bug. However, using the direct question prompt, ChatGPT predicts that the repaired code

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

```
/**
* Everyone knows Fibonacci sequence, it was studied deeply by
mathematicians in the last
* couple centuries. However, what people don't know is Tribonacci sequence.
Tribonacci
* sequence is defined by the recurrence: 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. You are given a non-negative
integer number
* n, you have to a return a list of the first n + 1 numbers of the Tribonacci
sequence.
*/
public class TRI {
    public static int[] tri(int n) {
        int[] sequence = new int[n + 1];
        if(n == 0) { sequence[0] = 1; return sequence; }
        sequence[0] = 1; sequence[1] = -3;
        for(int i = 2; i < n - 1; i--) {
            if(i % 2 == 0) sequence[i] = i / 2 - 1;
            else sequence[i] = sequence[i - 1] + sequence[i - 2] – (i + 3) / 2;
        }
        return sequence;
}}
```

```
public class TRI {
    public static int[] tri(int n) {
        ...
        sequence[0] = 1; sequence[1] = 3;
        for(int i = 2; i <= n; i++) {
            if(i % 2 == 0) sequence[i] = 1 + i / 2;
            else sequence[i] = sequence[i - 1] + sequence[i - 2] + sequence[i + 1];
        ...
}}
```

Buggy repaired code

Buggy.

Direct Question

Fig. 10. The truly buggy repaired code being predicted buggy (i.e., Example 10).

```
/**
* Input:
*   coins: A list of positive ints representing coin denominations
*   total: An int value to make change for
* Output:
*   The number of distinct ways to make change adding up to total using only
*    coins of the given values.
*/
public class POSSIBLE_CHANGE {
    public static int possible_change(int[] coins, int total)
    if (total == 0) {
        return 1;
        }
        if (total < 0) {
            return 0;
        }
        int first = coins[0];
        int[] rest = Arrays.copyOfRange(coins, 1, coins.length);
        return possible_change(coins, total - first) + possible_change(rest, total);
}}
```

```
public class POSSIBLE_CHANGE {
    public static int possible_change(int[] coins, int total) {
        ...
        if (total < 0 || coins.length == 0) { return 0; }
        ...
}}
```

Location of repair

Correct repaired code

Buggy.

Direct Question

Fig. 11. The truly correct repaired code being predicted buggy (i.e., Example 11).

may still contain the bug. When asked for the reasons, ChatGPT mentions issues such as not handling negative totals correctly and not handling empty coin arrays properly. However, it is evident that the repaired program has already considered and addressed these problems.

> **Finding 3**: The direct question often causes ChatGPT to verify its failed program repairs as successful erroneously. The guiding question prompt helps identify more failed repairs but also increases false alarms. The test report prompt also identifies more failed repairs, but the explanations provided are mostly incorrect. Additionally, self-contradictory hallucinations occur, where ChatGPT initially outputs what it believes to be a successful repair but later predicts it as buggy during self-verification.

VI. DISCUSSION

A. The Impact of Different Temperatures

In the self-verification phase of the aforementioned experiments, we set the temperature to 0 to minimize randomness and encourage more deterministic responses regarding code correctness, vulnerability detection, and repair success. However, even at a temperature of 0, ChatGPT still exhibits some degree of randomness. To explore the impact of temperature, we conduct the experiment by running the same prompt five times at a temperature of 0.8 using the GPT-3.5 model. Tables IV and V respectively show the average values and standard deviations for each metric across the five runs on code generation, program repair, and code completion tasks. Despite some differences in ChatGPT's average performance across the five self-verifications at a temperature of 0.8 compared to 0, the standard deviations of the metrics are very low. For example, in the code generation task, as shown in Table IV, when using the direct question prompt on the HumanEval-Python and MBXP-Python datasets, the standard deviation of the F1-score is only 0.04 and 0.03, respectively. Similarly, in Table V, when using the direct question prompt on the HumanEval-Java$_R$ dataset for program repair and the code completion dataset, the standard deviation of the F1-score is 0.04 and 0.05, respectively. These indicate that ChatGPT's responses during self-verification are relatively consistent across multiple runs, even with a higher temperature setting.

B. Assessing the Self-Verification Capability of GPT-4

OpenAI offers the ChatGPT API with several models including GPT-3.5 and GPT-4. We primarily use the GPT-3.5-turbo model for our experimental evaluation because it is more cost-effective, offers faster access rates, and provides greater stability, making it better suited for large-scale experiments compared to GPT-4. However, we still conduct small-scale experiments on GPT-4 to verify whether the results would indeed differ from those of GPT -3.5. Specifically, we choose HumanEval-Python and HumanEval-Java for the code generation dataset, HumanEval-Java$_R$ for the program repair dataset, and continue to use the dataset provided by Pearce et al. [9] for code completion task. The results of the self-verification capability of GPT-4 in small-scale experiments are presented in Table VI.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
THE AVERAGE AND STANDARD DEVIATION OF ACCURACY, PRECISION, RECALL, AND F1-SCORE FOR CHATGPT IN THE CODE GENERATION TASK RUNNING AT A TEMPERATURE OF 0.8 FOR 5 TIMES

Dataset	Prompt	Acc	Prec	Rec	F1
H-Python	DQ	0.72 ±0.01	0.85 ±0.13	0.06 ±0.02	0.12 ±0.04
	GQ	0.67 ±0.02	0.42 ±0.05	0.32 ±0.03	0.37 ±0.04
	TR	0.70 ±0.01	0.20 ±0.04	0.004 ±0.01	0.01 ±0.01
H-Java	DQ	0.66 ±0.01	0.60 ±0.08	0.09 ±0.03	0.16 ±0.04
	GQ	0.61 ±0.03	0.46 ±0.03	0.59 ±0.03	0.51 ±0.02
	TR	0.65 ±0.002	0.30 ±0.02	0.01 ±0.01	0.03 ±0.02
H-JS	DQ	0.65 ± 0.02	0.58 ± 0.10	0.09 ± 0.03	0.12 ± 0.02
	GQ	0.56 ± 0.02	0.43 ± 0.04	0.37 ± 0.02	0.40 ± 0.03
	TR	0.63 ± 0.01	0.46 ± 0.03	0.09 ± 0.01	0.15 ± 0.02
H-Go	DQ	0.59 ± 0.01	0.72 ± 0.12	0.18 ± 0.02	0.27 ± 0.03
	GQ	0.57 ± 0.02	0.57 ± 0.03	0.21 ± 0.02	0.27 ± 0.02
	TR	0.56 ± 0.01	0.21 ± 0.08	0.04 ± 0.02	0.06 ± 0.03
H-C++	DQ	0.58 ± 0.02	0.63 ± 0.13	0.11 ± 0.02	0.15 ± 0.02
	GQ	0.56 ± 0.01	0.50 ± 0.04	0.37 ± 0.01	0.42 ± 0.03
	TR	0.53 ± 0.004	0.53 ± 0.01	0.01 ± 0.003	0.04 ± 0.01
M-Go	DQ	0.79 ± 0.01	0.29 ± 0.05	0.16 ± 0.01	0.21± 0.02
	GQ	0.71 ± 0.01	0.20 ± 0.03	0.36 ± 0.01	0.31 ± 0.02
	TR	0.83 ± 0.01	0.23 ± 0.02	0.02 ± 0.01	0.02 ± 0.01
M-Python	DQ	0.76 ± 0.01	0.22 ± 0.09	0.05± 0.02	0.06 ± 0.03
	GQ	0.57 ± 0.01	0.19 ± 0.03	0.37 ± 0.01	0.27 ± 0.03
	TR	0.78 ± 0.02	0.29 ± 0.02	0.03 ± 0.02	0.06 ± 0.04
M-C++	DQ	0.62 ± 0.01	0.60 ± 0.14	0.05 ± 0.02	0.09 ± 0.04
	GQ	0.60 ± 0.02	0.43 ± 0.03	0.33 ± 0.02	0.38 ± 0.03
	TR	0.59 ± 0.01	0.73 ± 0.02	0.01 ± 0.004	0.05 ± 0.02
M-C#	DQ	0.61 ± 0.01	0.59 ± 0.06	0.08 ± 0.01	0.09 ± 0.05
	GQ	0.58 ± 0.02	0.54 ± 0.05	0.26 ± 0.02	0.35 ± 0.04
	TR	0.60 ± 0.01	0.49 ± 0.03	0.01 ± 0.002	0.01 ± 0.01
M-Java	DQ	0.58 ± 0.01	0.64 ± 0.03	0.11 ± 0.004	0.19 ± 0.03
	GQ	0.55 ± 0.02	0.49 ± 0.03	0.35 ± 0.01	0.39 ± 0.02
	TR	0.57 ± 0.01	0.54 ± 0.02	0.01 ± 0.003	0.03 ± 0.01
M-Kotlin	DQ	0.59 ± 0.02	0.72 ± 0.11	0.06 ± 0.02	0.09 ± 0.03
	GQ	0.56 ± 0.02	0.54 ± 0.03	0.45 ± 0.01	0.49 ± 0.02
	TR	0.57 ± 0.01	0.56 ± 0.02	0.04 ± 0.01	0.07 ± 0.02
M-JS	DQ	0.56 ± 0.03	0.79 ± 0.07	0.09 ± 0.02	0.15 ± 0.03
	GQ	0.56 ± 0.01	0.61 ± 0.05	0.43 ± 0.02	0.50 ± 0.03
	TR	0.54 ± 0.01	0.55 ± 0.02	0.09 ± 0.01	0.15 ± 0.02
M-TS	DQ	0.56 ± 0.04	0.73 ± 0.13	0.08 ± 0.02	0.13 ± 0.04
	GQ	0.57 ± 0.02	0.60 ± 0.03	0.33 ± 0.02	0.40 ± 0.04
	TR	0.55 ± 0.01	0.47 ± 0.02	0.03 ± 0.01	0.08 ± 0.02
M-Scala	DQ	0.54 ± 0.01	0.71 ± 0.05	0.06 ± 0.02	0.10 ± 0.04
	GQ	0.59 ± 0.03	0.67 ± 0.06	0.24 ± 0.02	0.34 ± 0.04
	TR	0.55 ± 0.02	0.71 ± 0.09	0.09 ± 0.04	0.11 ± 0.05
M-PHP	DQ	0.53 ± 0.02	0.77 ± 0.08	0.05 ± 0.03	0.09 ± 0.04
	GQ	0.58 ± 0.02	0.57 ± 0.04	0.37 ± 0.02	0.49 ± 0.03
	TR	0.55 ± 0.01	0.44 ± 0.06	0.02 ± 0.004	0.09 ± 0.02
M-Swift	DQ	0.52 ± 0.02	0.74 ± 0.10	0.07 ± 0.02	0.08 ± 0.03
	GQ	0.56 ± 0.01	0.60 ± 0.03	0.39 ± 0.01	0.48 ± 0.02
	TR	0.48 ± 0.01	0.85 ± 0.02	0.03 ± 0.01	0.06 ± 0.01
M-Perl	DQ	0.50 ± 0.02	0.71 ± 0.05	0.13 ± 0.03	0.27 ± 0.04
	GQ	0.53 ± 0.03	0.61 ± 0.04	0.48 ± 0.02	0.52 ± 0.05
	TR	0.49 ± 0.02	0.69 ± 0.06	0.04 ± 0.01	0.08 ± 0.03
M-Ruby	DQ	0.32 ± 0.03	0.81 ± 0.12	0.10 ± 0.02	0.19 ± 0.03
	GQ	0.51 ± 0.03	0.76 ± 0.09	0.41 ± 0.03	0.54 ± 0.05
	TR	0.29 ± 0.01	0.83 ± 0.04	0.05 ± 0.02	0.09 ± 0.02

TABLE V
THE AVERAGE AND STANDARD DEVIATION OF ACCURACY, PRECISION, RECALL, AND F1-SCORE FOR CHATGPT IN THE PROGRAM REPAIR AND CODE COMPLETION TASKS RUNNING AT A TEMPERATURE OF 0.8 FOR 5 TIMES

Dataset	Prompt	Acc	Prec	Rec	F1
QB-Python	DQ	0.81 ± 0.02	0.69 ± 0.11	0.15 ± 0.03	0.22 ± 0.04
	GQ	0.61 ± 0.04	0.24 ± 0.10	0.33 ± 0.04	0.28 ± 0.05
	TR	0.73 ± 0.03	0.37 ± 0.09	0.35 ± 0.02	0.34 ± 0.03
QB-Java	DQ	0.61 ± 0.01	0.23 ± 0.14	0.10 ± 0.04	0.13 ± 0.06
	GQ	0.57 ± 0.02	0.33 ± 0.06	0.31 ± 0.02	0.34 ± 0.04
	TR	0.59 ± 0.03	0.41 ± 0.04	0.35 ± 0.03	0.37 ± 0.05
H-Java$_R$	DQ	0.65 ±0.01	0.54 ±0.07	0.16 ±0.02	0.24 ±0.04
	GQ	0.44 ±0.01	0.33 ±0.01	0.57 ±0.02	0.42 ±0.01
	TR	0.65 ±0.02	0.52 ±0.09	0.12 ±0.02	0.20 ±0.04
D$_{completion}$	DQ	0.73 ±0.03	0.56 ±0.15	0.14 ±0.03	0.22 ±0.05
	GQ	0.35 ±0.03	0.27 ±0.01	0.78 ±0.03	0.40 ±0.02
	TR	0.33 ±0.04	0.25 ±0.03	0.75 ±0.09	0.38 ±0.04

TABLE VI
THE RESULTS OF THE SELF-VERIFICATION CAPABILITY OF GPT-4 IN SMALL-SCALE EXPERIMENTS

Dataset	Prm	Acc	Prec	Rec	F1	TN	FN	FP	TP
H-Python	DQ	**0.80**	**0.64**	0.25	0.36	123	27	5	9 (0/0)
	GQ	**0.80**	0.55	**0.64**	**0.59**	109	13	19	23 (20/23)
	TR	0.79	0.60	0.08	0.15	126	33	2	3 (1/3)
H-Java	DQ	**0.82**	**0.68**	0.45	0.54	118	21	8	17 (0/0)
	GQ	0.77	0.50	**0.61**	**0.55**	103	15	23	23 (18/23)
	TR	0.76	0.00	0.00	0.00	125	38	1	0 (0/0)
H-Java$_R$	DQ	**0.73**	**0.27**	0.36	0.31	109	18	27	10 (0/0)
	GQ	0.61	0.26	**0.68**	**0.37**	81	9	55	19 (11/19)
	TR	0.74	0.23	0.21	0.22	116	22	20	6 (0/6)
Dataset$_{completion}$	DQ	**0.67**	**0.44**	0.27	0.33	29	11	5	4 (0/0)
	GQ	0.47	0.34	0.80	0.48	11	3	23	12 (10/12)
	TR	0.53	0.38	**0.87**	**0.53**	22	10	12	5 (5/5)

Despite the numerical disparities of these metrics between GPT-4 and GPT-3.5, our overall conclusion regarding GPT-4 and GPT-3.5 remains unaltered. GPT-4 also frequently misclassifies its generated incorrect code as correct, its vulnerable code as non-vulnerable, and its failed program repairs as successful when using the direct question prompt, resulting in low recall. Employing the guiding question prompt improves the detection of buggy and vulnerable code and failed repairs, thereby increasing recall and F1 scores. However, using the test report prompt does not significantly enhance the identification of incorrect code or failed repairs. Additionally, instances of self-contradictory hallucinations are also observed in GPT-4's behavior.

C. The Impact of Different Guiding Questions

In the aforementioned experiments, the guiding question prompt asks ChatGPT to indicate its agreement or disagreement regarding assertions that (1) the code does NOT implement the function based on the requirement description, (2) the completed code HAS vulnerabilities, and (3) the repaired code

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

TABLE VII
THE COMPARISON RESULTS OF THE SELF-VERIFICATION CAPABILITY OF CHATGPT USING THE NEGATIVE AND POSITIVE GUIDING QUESTIONS IN SMALL-SCALE EXPERIMENTS

Dataset	Prm	Acc	Prec	Rec	F1	TN	FN	FP	TP
H-Python	DQ	**0.74**	**1.00**	0.13	0.22	116	42	0	6 (5/6)
	N-GQ	0.64	0.39	**0.42**	**0.40**	85	28	31	20 (9/17)
	P-GQ	0.72	**1.00**	0.04	0.08	116	46	0	2 (1/2)
H-Java	DQ	**0.70**	**0.82**	0.16	0.26	105	48	2	9 (8/9)
	N-GQ	0.63	0.46	**0.33**	**0.39**	85	38	22	19 (5/15)
	P-GQ	0.65	0.50	0.04	0.07	105	55	2	2 (1/2)
H-Java$_R$	DQ	0.65	0.50	0.10	0.17	100	52	6	6 (0/0)
	N-GQ	0.51	0.39	**0.69**	**0.50**	44	18	62	40 (9/13)
	P-GQ	**0.66**	**0.63**	0.09	0.15	103	53	3	5 (4/5)
D$_{completion}$	DQ	**0.75**	**1.00**	0.08	0.14	35	12	0	1 (0/0)
	N-GQ	0.29	0.24	**0.77**	**0.37**	4	3	31	10 (9/10)
	P-GQ	0.65	0.39	0.54	0.45	24	6	11	7 (6/7)

does NOT correctly implement the function. In other words, we employ a negative guiding question. The experiment shows that using the negative guiding question compared to the direct question enables ChatGPT to identify more buggy or vulnerable generated code. To explore the impact of different guiding questions, in this subsection, we utilize a positive guiding question, which asks ChatGPT to indicate its agreement or disagreement regarding assertions that (1) the code CORRECTLY implements the function based on the requirement description, (2) the completed code does NOT have vulnerabilities, and (3) the repaired code CORRECTLY implements the function. The comparison results of the self-verification capability using the negative and positive guiding questions in small-scale experiments are presented in Table VII.

Compared to the negative version, using the positive version reduces false positives, resulting in higher precision. However, this leads to lower recall and F1-score, as ChatGPT more frequently incorrectly predicts buggy code as correct and vulnerable code as non-vulnerable. For example, in the HumanEval-Python dataset, when employing the positive version, ChatGPT erroneously predicts 96% of incorrectly generated code as correct (compared to 58% for the negative version) and identifies only 4% (compared to 42% for the negative version) of incorrect code as buggy. Similarly, in the HumanEval-Java$_R$ dataset, using the positive version results in ChatGPT erroneously predicting 91% of unsuccessfully repaired programs as successfully repaired (compared to 31% for the negative version) and recognizing only 9% (compared to 69% for the negative version) of unsuccessfully repaired programs. In the code completion dataset, employing the positive version leads to ChatGPT erroneously predicting 46% of generated vulnerable code as non-vulnerable (compared to 23% for the negative version) and identifying 54% (compared to 77% for the negative version) of truly vulnerable code as vulnerable. These findings underscore ChatGPT's unreliability, as altering the self-verification prompt results in significant changes in responses. Due to the importance of identifying code with bugs or vulnerabilities during actual development, we have opted for the negative guiding question.

D. The Robustness of ChatGPT's Self-Verification Capabilities Against Prompt Perturbations

Previous studies have evaluated the robustness of LLMs in program-related tasks against prompt perturbations [22], [23], [24], [25], [26], showing that sentence-level prompt rewriting has a greater impact on LLMs such as Codex [25]. To better understand ChatGPT's robustness in self-verification, we apply the prompt rewriting perturbation strategy to the direct question prompt in a small-scale experiment using the GPT-3.5 model, as shown in Table VIII. We do not apply perturbations for the guiding question and test report prompts since these are specifically tailored questioning methods. The results of ChatGPT's self-verification capability after the prompt perturbations to the direct question prompt are shown in Table IX.

After rewriting perturbations, ChatGPT's self-verification capability shows only minor changes. For example, in the code generation task on the HumanEval-Java, compared to the results before perturbation, the number of self-contradictory hallucinations is 13 (previously 11), the accuracy is 0.67 (previously 0.70), the recall is 0.14 (previously 0.16), and the F1-score is 0.23 (previously 0.26). For the code repair task on the HumanEval-Java-Repair dataset, the number of self-contradictory hallucinations is 13 (previously 12), the accuracy is 0.59 (previously 0.65), the recall is 0.04 (previously 0.10), and the F1-score is 0.06 (previously 0.17). On the code completion dataset, the number of self-contradictory hallucinations is 2 (previously 1), the accuracy is 0.77 (previously 0.75), the recall is 0.15 (previously 0.08), and the F1-score is 0.26 (previously 0.14). These results suggest that ChatGPT is relatively insensitive to prompt rewriting perturbations, demonstrating reliable robustness in self-verification.

VII. IMPLICATIONS

We recognize that querying ChatGPT, a probabilistic tool, may yield incorrect or varied responses. However, these responses are not purely random; rather, they reflect biased predictions learned from extensive data. Nevertheless, our findings reveal several key points: (a) Altering the self-verification prompt frequently leads to changes in ChatGPT's responses regarding code correctness, with only two possible outcomes: correct or incorrect. (b) ChatGPT often exhibits instances of self-contradictory hallucination. The diverse responses and frequent self-contradictions underscore the need to address these issues to enhance the reliability of the user experience with ChatGPT. From our observations, several implications can be drawn:

(1) **It is not advisable to rely only on ChatGPT as both a developer and a tester in software development, and human expertise and judgment play an indispensable role in the process.** The inaccuracies and self-contradictory hallucinations encountered during ChatGPT's self-verification emphasize the need for caution and a thorough evaluation of its output. ChatGPT should be viewed as a tool that assists developers rather than replacing their role as autonomous software developers and testers. It is crucial to combine the capabilities

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
THE REWRITING PERTURBATIONS OF DIRECT QUESTIONS PROMPT FOR THREE CODE-RELATED TASKS

Task		Prompt text
Code Generation	Original prompt	Does the code correctly implement "$check_point$" function based on the description of "$function_desc$"? Answer the question (Yes or No).
	Rewritten prompt	According to the description of "$function_desc$", does the implementation of "$check_point$" function match the requirements? Answer Yes or No.
Code Completion	Original prompt	Label the code as either "vulnerable" or "non-vulnerable" based on whether the code has vulnerabilities.
	Rewritten prompt	Does the code contain vulnerabilities? Label it as "vulnerable" if it does, otherwise label it as "non-vulnerable".
Program Repair	Original prompt	Label the $lang$ code as either "buggy" or "bug-free" based on whether the code correctly implements the function "$func_desc$".
	Rewritten prompt	Determine if the $lang$ code correctly implements the function "$func_desc$" and label it as "buggy" or "bug-free" accordingly.

TABLE IX
THE RESULTS OF THE SELF-VERIFICATION CAPABILITY OF CHATGPT AFTER REWRITING PERTURBATIONS TO THE DIRECT QUESTION PROMPT

Dataset	Acc	Prec	Rec	F1	TN	FN	FP	TP
H-Python	0.74	0.71	0.21	0.32	112	38	4	10
H-Java	0.67	0.62	0.14	0.23	102	49	5	8
H-Java$_R$	0.59	0.15	0.04	0.06	95	56	11	2
D$_{completion}$	0.77	1.00	0.15	0.26	35	11	0	2

of ChatGPT with human expertise to ensure the quality and reliability of the generated code.

(2) **Collecting instances of self-contradictory hallucinations can be a valuable approach to refine ChatGPT.** The instances of self-contradictory hallucinations highlight the limitations and potential risks associated with ChatGPT. Similar studies on testing question-answering models [27], [28], [29], [30], [31] have demonstrated the effectiveness of fine-tuning models using self-contradictory samples identified through software testing methods. By drawing inspiration from these studies, developers can proactively gather and analyze user feedback, specifically focusing on instances of self-contradictory hallucination. By incorporating these instances into the fine-tuning process, developers can improve the capability of ChatGPT and effectively eliminate self-contradictory hallucinations, leading to a more reliable experience for users.

(3) **Prompt engineering is a potential approach to improve ChatGPT's self-verification capability.** By incorporating the guiding question prompt, ChatGPT's ability for self-verification in code generation, code completion, and program repair can be improved, albeit with an increased risk of false alarms. Additionally, the use of the test report prompt can further enhance ChatGPT's self-verification capability in the code completion task. Therefore, in multi-agent systems where ChatGPT acts as a tester, the guiding question prompt can be utilized to detect more buggy codes generated by ChatGPT acting as a developer, thereby enhancing the overall correctness of the generated code. However, there is still a long way to go in designing the perfect prompt to maximize ChatGPT's self-verification capabilities. Further research in prompt engineering, particularly focused on self-verification purposes, is necessary to continue advancing the field.

VIII. THREATS OF VALIDITY

Datasets. We have opted for HumanEval and MBXP for code generation tasks, as well as QuixBugs-Python/Java and HumanEval-Java$_R$ for program repair, due to their prevalent use in recent studies on code generation and program repair. We acknowledge the widespread use of two classic program repair datasets, Defects4J [32] and ManyBugs [33] in program repair research. These two datasets consist of real-world programs, but the buggy functions/methods lack function descriptions. Using ChatGPT without specifying the intended requirement and expecting it to generate repaired code is unfair. In such scenarios, the results produced by ChatGPT may be either random or memorized if it encountered these programs in these datasets during its training. Therefore, we do not select the Defects4J and ManyBugs datasets.

Evaluation Metrics. Similar to existing studies on bug prediction and vulnerability detection [34], [35], [36], [37], [38], we use four common binary classification metrics—accuracy, precision, recall, and F1-score—to assess ChatGPT's self-verification capabilities. Accuracy can be misleading in imbalanced datasets, where one class significantly outnumbers the other. To address this limitation, we also consider precision and recall. Precision helps us determine how much of the predicted buggy/vulnerable code is actually correct, thereby reducing the risk of false positives. Recall, on the other hand, measures the model's ability to identify all actual buggy/vulnerable code, ensuring that fewer true issues are missed (i.e., reducing false negatives). However, high precision may still overlook much actual buggy/ vulnerable code, while high recall might lead to an increase in false positives. Therefore, we use the F1-score, which balances precision and recall, though it assumes both are equally important, which may not be the case in all scenarios.

Data Leakage. Data leakage presents a potential concern where testing samples become visible during model training,

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

potentially leading to its output generated through memorization. In our experiments, we primarily utilize GPT-3.5 API, which was trained on data up until January 2022. Our experimental datasets, including HumanEval-Python, MBXP-Python, and QuixBugs-Python/Java, were created before January 2022, while others were generated afterward. We find the results between the two types of datasets are similar. For example, the code generation accuracy is 71% and 65% for HumanEval-Python and HumanEval-Java. During self-verification, ChatGPT erroneously predicts incorrectly generated code as correct, with error rates of 88% and 84% for HumanEval-Python and HumanEval-Java using the direct question, 58% and 67% using the guiding question prompt, and 94% and 95% using the test report prompt (shown in Table I). In other words, the impact of data leakage on ChatGPT's self-verification capability appears to be minimal.

Context Size Limitation. The official OpenAI documentation[5] specifies that the maximum size of the context window for the GPT-3.5-turbo model is 16,385 tokens, which includes both user input and model-generated output. Additionally, the model's output can have up to 4,096 tokens. Therefore, in a single conversation, the actual user input can have up to 12,289 tokens (16,385 - 4,096) to ensure the model can generate an output of up to 4,096 tokens. Based on the token encoding method of the GPT-3.5-turbo model, we have analyzed all the conversations in our experiments. The results indicate that in the code generation tasks for the Scala language on the MBXP dataset, the self-verification in the Test Report conversation had a maximum input size of 4,188 ($<$ 12,289) tokens. Thus, the context size limit of the model did not pose any issues for our experimental process.

Reproducibility of Our Findings. The responses generated by ChatGPT are non-deterministic, meaning that even the same prompt can produce different answers. Therefore, we have conducted several experiments to demonstrate the robustness and reproducibility of our findings, utilizing both GPT-3.5 and GPT-4 models to ensure that our results are not confined to a specific model. We also examine the impact of changing the temperature parameter and rewriting task prompts. Although some fluctuations occur, they are generally minor and confirm the consistency of our findings under different conditions. To address these concerns further and enhance transparency and reproducibility, we have made our experimental data and code replication package publicly accessible online.

IX. Related Work

LLMs for Software Engineering. Recently, researchers have proposed various fine-tuning and prompt engineering approaches to further enhance the performance of LLMs on specific coding tasks. For the code generation task, Chen et al. [39] proposed to employ Codex to create test cases in a zero-shot manner, which were employed to improve the correctness of its generated code. Madaan et al. [40] employed program trajectories to fine-tune LLMs for enhancing code generation. Jiang et al. [41] introduced a two-phase approach to code generation, involving LLMs generating code-writing plans before the final implementation. Dong et al. [1] and Qian et al. [3] suggested a cooperative approach where LLMs like ChatGPT adopt multiple roles (analyst, coder, and tester) to collaboratively tackle code generation tasks. Although these studies enabled ChatGPT to function as a tester by generating test reports for the generated code and resolving bugs based on these reports, they did not explicitly evaluate the efficacy of these generated test reports in validating the generated code (i.e., ChatGPT's self-verification capability). Ni et al. [42] fine-tuned the model by training a validator to determine whether programs sampled from LLMs were correct to improve language-to-code generation. Ali et al. [43] reduced memory usage during training by optimizing the rank decomposition matrix of the base model. Thakur et al. [44] fine-tuned LLMs by evaluating the functional correctness of the generated Verilog code. For code completion, Zhang et al. [6] and Liu et al. [45] have proposed the increased usage of intra-repository information. They employed a retrieve-then-generate model to assist ChatGPT with retrieved code examples. Guo et al. [46] pre-trained on a corpus of high-quality project-level code to enhance code-completion capabilities. For the program repair task, Chen et al. [47] and Kang et al. [48] allowed LLMs to generate both explanations and code patches based on the execution results derived from code interpreters. Jin et al. [49] involved mining referential bugs and their respective fixes from real software development workflows, and utilized this information to enable Codex to repair programs. Xia et al. [50], Jiang et al. [19], and Fan et al. [51] employed collected program repair datasets to fine-tune LLMs and observed that using fine-tuned LLMs resulted in improved program repair performances. Berabi et al. [52] improved the performance of fixing security vulnerabilities by reducing code length and utilizing LLMs.

The Hallucination of LLMs. Rawte et al. [53] classified types of hallucination phenomena of LLMs and established evaluation criteria. Shen et al. [54] conducted a comprehensive measurement of ChatGPT's reliability in generic question-and-answer scenarios, and revealed that ChatGPT often presented opinions as facts or included imaginary specifics without proper qualification. Liu et al. [55] and Tian [56] summarized and categorized the code hallucinations in LLM-generated code. Mündler et al. [11] conducted an analysis of self-contradiction hallucinations in which LLMs generate two contradictory sentences within the same context, thereby revealing the lack of factuality in these models. Jang et al. [57] found self-contradictory hallucination in terms of semantic consistency, negation consistency, and symmetric consistency. They observed that ChatGPT perceived two sentences with the same meaning as expressing identical intentions. Interestingly, even when one of the sentences was modified with a negation word, ChatGPT still predicted that both sentences conveyed the same meaning. In the context of code-related tasks, Ma et al. [58] highlighted ChatGPT's susceptibility to hallucination when interpreting code semantic structures and fabricating nonexistent facts. White et al. [59] discovered that ChatGPT had a tendency to confidently and enthusiastically hallucinate incorrect output, underscoring the need for careful scrutiny by human users. Different from the aforementioned studies, our research presents

[5]https://platform.openai.com/docs/models/gpt-3-5-turbo

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

the first exploration of self-contradiction hallucinations in code-related tasks.

Bug detection and vulnerability detection using LLMs. Recently, LLMs have also been used for bug detection [60], [61], [62], [63] and vulnerability detection [36], [37], [38] tasks. For example, Kang et al. [61] proposed AutoFL, a GPT-4-based fault localization method that not only identified fault locations but also generated explanations for the bugs. The results showed that AutoFL outperformed existing techniques at the method level, achieving a Precision@5 value of 71% and providing accurate explanations for 56.7% of all bugs. Zhou et al. [36] found that when directly asked to identify vulnerabilities in methods, GPT-4 achieved an accuracy of 60.3%, a precision of 67.3%, a recall of 40.2%, and an F1 score of 50.3%. The aforementioned works verified the performance of LLMs in bug detection and vulnerability detection on task-specific datasets (externally produced code solutions). However, with the increasing popularity of using LLMs in code-related tasks within multi-agent scenarios [1], [3], [64], there is a lack of research on how well LLMs can simultaneously act as both developers and testers to self-verify the correctness of their own generated code. Therefore, our study investigates ChatGPT's self-verification capabilities in code-related tasks and explores whether ChatGPT, when acting in this dual role, exhibits self-contradictory hallucinations.

AI-Generated Content Detection. Detecting Artificial Intelligence-Generated Content (AIGC) is crucial for promoting responsible and ethical use of such content. To effectively detect whether a given natural language text is generated by AI, numerous text detection methods [65], [66], [67] have been developed in academia and industry. For example, Liu et al. [67] proposed a detector using a support vector machine and RoBERTa, successfully distinguishing AI-generated essays from human ones. Similarly, Liao et al. [68] demonstrated the efficacy of a BERT-based model in detecting medical texts generated by ChatGPT. However, detecting AI-generated code presents a challenge. Wang et al. [69] conducted the first empirical study to evaluate existing AIGC detectors in identifying whether code-related content is generated by ChatGPT. The study revealed that existing detectors exhibited lower performance on code-related data compared to natural language data. Our work focuses on utilizing ChatGPT to self-verify whether the code content it produces meets the requirements, instead of relying on external detectors, which differs from Wang et al.'s study [69].

X. Conclusion

In this paper, we conduct the first empirical study that evaluates ChatGPT's self-verification capabilities in code completion, code generation, and program repair. We first ask ChatGPT to generate the correct code, complete the code and ensure that the completed code has no vulnerabilities, and repair the buggy code. Then, we ask ChatGPT to self-verify the correctness of the generated code, the presence of vulnerabilities in code completions, or the success of code repairs. We employ three types of verification prompts for this study: direct question, guiding question, and test report. The results reveal that (1) ChatGPT frequently makes erroneous predictions during self-verification, incorrectly labeling its generated code as correct, completed code as non-vulnerable, and program repairs as successful. (2) There are some instances of self-contradictory hallucinations in ChatGPT's behavior, where it initially generates code or completions that it deems correct or secure but later contradicts this belief during self-verification. (3) The self-verification capability of ChatGPT can be enhanced by asking the guiding question, which queries whether ChatGPT agrees with assertions about incorrectly generated or repaired code and vulnerabilities in completed code. (4) Using a test report generated by ChatGPT can identify more vulnerabilities in completed code, but the explanations in the test report are mostly incorrect for incorrectly generated code and failed repairs. Based on these results, several implications are derived for further research. Our source code and experimental results are available at https://figshare.com/s/4b51f0b8a2cda17d08be.

References

[1] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via ChatGPT," 2023, *arXiv:2304.07590*.

[2] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating code generation performance of ChatGPT with crowdsourcing social data," in *Proc. 47th IEEE Comput. Softw. Appl. Conf.*, 2023, pp. 876–885.

[3] C. Qian et al., "Communicative agents for software development," 2023, *arXiv:2307.07924*.

[4] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. Chi Conf. Human Factors Comput. Syst. Extended Abstr.*, 2022, pp. 1–7.

[5] B. T. Hammond Pearce, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can OpenAI codex and other large language models help us fix security bugs," 2021, *arXiv:2112.02125*.

[6] F. Zhang et al., "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Proc. 28th Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 2471–2484.

[7] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," 2023, *arXiv:2304.00385*.

[8] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT," in *Proc. IEEE/ACM Int. Workshop Autom. Program Repair.*, 2023, pp. 23–30.

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions," in *Proc. 43rd IEEE Symp. Secur. Privacy*, 2022, pp. 754–768.

[10] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, "Security implications of large language model code assistants: A user study," 2022, *arXiv:2208.09727*.

[11] N. Mündler, J. He, S. Jenko, and M. Vechev, "Self-contradictory hallucinations of large language models: Evaluation, detection and mitigation," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024.

[12] F. Cassano et al., "MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3675–3691, Jul. 2023.

[13] Q. Xin, "Towards addressing the patch overfitting problem," in *Proc. 39th IEEE/ACM Int. Conf. Softw. Eng.*, 2017, pp. 489–490.

[14] "CodeQL documentation," CodeQL | GitHub. Accessed: Apr. 2024. [Online]. Available: https://codeql.github.com/docs/

[15] B. Athiwaratkun et al., "Multi-lingual evaluation of code generation models," in *Proc. 11th Int. Conf. Learn. Represent.*, 2023.

[16] Q. Zheng et al., "CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X," in *Proc. 29th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, 2023, pp. 5673–5684.

[17] T. M. C. (MITRE), "2021 CWE top 25 most dangerous software weaknesses," 2021. Accessed: Nov. 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1337.html

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.

[18] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge," in *Proc. Companion ACM Int. Conf. Syst., Program., Languages, Appl.: Softw. Humanity*, 2017, pp. 55–56.
[19] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 1430–1442.
[20] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
[21] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?" in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2023, pp. 2445–2451.
[22] A. Mastropaolo et al., "On the robustness of code generation techniques: An empirical study on GitHub copilot," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 2149–2160.
[23] A. Shirafuji et al., "Exploring the robustness of large language models for solving programming problems," 2023, *arXiv:2306.14583*.
[24] M. Yan, J. Chen, J. M. Zhang, X. Cao, C. Yang, and M. Harman, "COCO: Testing code generation systems via concretized instructions," 2023, *arXiv:2308.13319*.
[25] T. Y. Zhuo et al., "On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex," in *Proc. 17th Conf. Eur. Chap. Assoc. Comput. Linguistics*, 2023, pp. 1090–1102.
[26] B. Wang et al., "Adversarial GLUE: A multi-task benchmark for robustness evaluation of language models," in *Proc. Neural Inf. Process. Syst. Track Datasets Benchmarks 1*, 2021.
[27] S. Chen, S. Jin, and X. Xie, "Testing your question answering software via asking recursively," in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2021, pp. 104–116.
[28] Q. Shen, J. Chen, J. M. Zhang, H. Wang, S. Liu, and M. Tian, "Natural test generation for precise testing of question answering software," in *Proc. 37th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2022, pp. 1–12.
[29] S. Gupta, P. He, C. Meister, and Z. Su, "Machine translation testing via pathological invariance," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 863–875.
[30] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *Proc. 42nd ACM/IEEE Int. Conf. Softw. Eng.*, 2020, pp. 961–973.
[31] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 303–314.
[32] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. 23rd Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.
[33] C. Le Goues et al., "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.
[34] S. Feng et al., "COSTE: Complexity-based oversampling technique to alleviate the class imbalance problem in software defect prediction," *Inf. Softw. Technol.*, vol. 129, 2021, Art. no. 106432.
[35] Z. Xu et al., "Cross project defect prediction via balanced distribution adaptation based transfer learning," *J. Comput. Sci. Technol.*, vol. 34, pp. 1039–1062, 2019.
[36] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proc. 44th ACM/IEEE Int. Conf. Softw. Eng.: New Ideas Emerg. Results*, 2024, pp. 47–51.
[37] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "ChatGPT for vulnerability detection, classification, and repair: How far are we?" in *Proc. 30th Asia-Pacific Softw. Eng. Conf.*, 2023, pp. 632–636.
[38] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *Proc. 34th IEEE Int. Symp. Softw. Rel. Eng. Workshops*, 2023, pp. 112–119.
[39] B. Chen et al., "CodeT: Code generation with generated tests," in *Proc. 11th Int. Conf. Learn. Represent.*, 2023.
[40] A. Madaan et al., "Learning performance-improving code edits," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024.
[41] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," 2023, *arXiv:2303.06689*.
[42] A. Ni et al., "LEVER: Learning to verify language-to-code generation with execution," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 26106–26128.
[43] Z. Ali et al., "Memory efficient with parameter efficient fine-tuning for code generation using quantization," in *Proc. 18th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2024, pp. 1–6.
[44] S. Thakur et al., "VeriGen: A large language model for Verilog code generation," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 29, pp. 1–31, 2023.
[45] T. Liu, C. Xu, and J. McAuley, "RepoBench: Benchmarking repository-level code auto-completion systems," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024.
[46] D. Guo et al., "DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence," 2024, *arXiv:2401.14196*.
[47] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024.
[48] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," 2023, *arXiv:2304.02195*.
[49] M. Jin et al., "InferFix: End-to-end program repair with LLMs," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 1646–1656.
[50] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. 45th Int. Conf. Softw. Eng.*, 2023, pp. 1482–1494.
[51] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, Piscataway, NJ, USA: IEEE, 2023, pp. 1469–1481.
[52] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "DeepCode AI fix: Fixing security vulnerabilities with large language models," 2024, *arXiv:2402.13291*.
[53] V. Rawte, A. Sheth, and A. Das, "A survey of hallucination in large foundation models," 2023, *arXiv:2309.05922*.
[54] X. Shen, Z. Chen, M. Backes, and Y. Zhang, "In ChatGPT we trust? Measuring and characterizing the reliability of ChatGPT," 2023, *arXiv:2304.08979*.
[55] F. Liu et al., "Exploring and evaluating hallucinations in LLM-powered code generation," 2024, *arXiv:2404.00971*.
[56] Y. Tian et al., "CodeHalu: Code hallucinations in LLMs driven by execution-based verification," 2024, *arXiv:2405.00253*.
[57] M. Jang and T. Lukasiewicz, "Consistency analysis of ChatGPT," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 15970–15985.
[58] W. Ma et al., "The scope of ChatGPT in software engineering: A thorough investigation," 2023, *arXiv:2305.12138*.
[59] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and Software Design," 2023, *arXiv:2303.07839*.
[60] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," 2023, *arXiv:2308.15276*.
[61] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of LLM-based explainable fault localization," *Proc. ACM Softw. Eng.*, vol. 1, pp. 1424–1446, 2024.
[62] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng.*, 2024, pp. 1–13.
[63] K. Liu et al., "LLM-powered test case generation for detecting tricky bugs," 2024, *arXiv:2404.10304*.
[64] L. Wang et al., "A survey on large language model based autonomous agents," *Front. Comput. Sci.*, vol. 18, no. 6, 2024, Art. no. 186345.
[65] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "DetectGPT: Zero-shot machine-generated text detection using probability curvature," in *Proc. Int. Conf. Mach. Learn.*, vol. 202, 2023, pp. 24950–24962.
[66] B. Guo et al., "How close is ChatGPT to human experts? Comparison corpus, evaluation, detection," 2023, *arXiv:*2301.07597.
[67] Y. Liu et al., "ArguGPT: Evaluating, understanding and identifying argumentative essays generated by GPT models," 2023, *arXiv:2304.07666*.
[68] W. Liao et al., "Differentiate ChatGPT-generated and human-written medical texts," 2023, *arXiv:2304.11567*.
[69] J. Wang, S. Liu, X. Xie, and Y. Li, "Evaluating AIGC detectors on code content," 2023, *arXiv:2304.05193*.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 16,2026 at 13:26:04 UTC from IEEE Xplore. Restrictions apply.