WildCode: An Empirical Analysis of Code Generated by ChatGPT

Kobra Khanmohammadi[1][0009−0004−1414−2111], Pooria Roy[2][0009−0004−3990−9905], Raphael Khoury[3][0000−0002−7625−3384], Abdelwahab Hamou-Lhadj[4][0000−0002−3319−5006], and Wilfried Patrick Konan[3]

[1] Sheridan College, Ontario, Canada; kobra.khanmohammadi@sheridancollge.ca
[2] School of Computing, Queen's University, Kingston, Canada; pooria.roy@queensu.ca
[3] Université du Québec en Outaouais (UQO), Canada; raphael.khoury@uqo.ca
[4] Concordia University, Montreal, Canada; wahab.hamou-lhadj@concordia.ca
[5] konk14@uqo.ca

Abstract. LLM models are increasingly used to generate code, but the quality and security of this code are often uncertain. Several recent studies have raised alarm bells, indicating that such AI-generated code may be particularly vulnerable to cyberattacks. However, most of these studies rely on code that is generated specifically for the study, which raises questions about the realism of such experiments. In this study, we perform a large-scale empirical analysis of real-life code generated by ChatGPT. We evaluate code generated by ChatGPT both with respect to correctness and security and delve into the intentions of users who request code from the model. Our research confirms previous studies that used synthetic queries and yielded evidence that LLM-generated code is often inadequate with respect to security. We also find that users exhibit little curiosity about the security features of the code they ask LLMs to generate, as evidenced by their lack of queries on this topic.

Keywords: Secure coding · software vulnerabilities · LLM · human-AI interaction · coding queries.

1 Introduction

In the span of only a few years, LLMs went from an emerging technology to an everyday tool, widely used by both tech-savvy programmers and ordinary users alike. Of particular interest is the use of LLMs to generate code. Recent surveys indicate that most developers rely on LLMs to generate code [5], a trend so pronounced that it is even blamed for a slowdown in the hiring of programmers.

This rapid change in the practice of computer programming took place with little consideration of the security of the code being produced. The preliminary findings on the degree of security of the code produced by LLMs are alarming [12, 7, 23]. Such studies usually find that LLMs produce code that falls below even modest expectations of security and which may require extensive modifications before it can be run safely in an untrusted context. This is especially the case if the programmer does not explicitly request that the code contains security checks or that it be resistant to specific categories of attack [12].

Most initial studies on the topic proceeded by asking an LLM to generate a series of programs, often guided by specific scenarios, and analyzing the resulting code, either manually or using an automated tool [26, 9, 14, 3, 18]. While such studies provide useful insights, there is a threat to validity because the scenarios chosen may not be representative of the actual interaction programmers have with LLMs.

In this paper, we present the first empirical study on the security of code generated by ChatGPT, one of the most widely used LLMs. Our analysis is based on data extracted from WildChat [27], a publicly available dataset containing more than one million real-world conversations with ChatGPT, from which we extract all conversations that include the code generated by the model. Unlike previous work that simulates user interactions by querying LLMs with synthetic prompts, our study leverages authentic user–ChatGPT interactions. This enables us to examine not only the security of the generated code, but also what users intend to ask, how they follow up on ChatGPT responses, and how they react when encountering buggy or insecure code.

In addition, we provide a curated set of annotated conversations and corresponding code snippets in which ChatGPT produced buggy or insecure responses. This dataset, available on our HuggingFace repository[6] and GitHub repository[7], contains the full list of annotated conversations and related code samples used in this study, as well as the rules used to extract patterns from the conversations, allowing reproducibility and facilitating further research in this area. The dataset includes syntactically correct code for Python, JavaScript, C/C++, Java, PHP, and C#, as well as unchecked or potentially erroneous code for other languages.

[6] https://huggingface.co/datasets/regularpooria/wildcode
[7] https://github.com/regularpooria/wildcode

The remainder of this paper is organized as follows. Section 2 reviews related works. Section 3 details the process by which we created the dataset of code used in this study. In Section 4, we analyze this code using a number of tools to determine its security level. Then, in Section 5, we examine the intentions of the users who request code from the model. Section 6 discusses observations and insights that can be gleaned from our results. Concluding remarks are given in Section 7.

2 Review of the Literature

There is significant interest in using generative AI for coding, as evidenced by the growing number of subscriptions to tools such as GitHub Copilot [8] and the reported usage of Amazon CodeWhisperer [2]. According to a survey conducted by GitHub [8], over 90% of developers now utilize generative AI tools to support their coding activities.Despite their popularity, analyses reveal that Large Language Models (LLMs) trained on open-source repositories often replicate insecure coding practices. Previous studies show that in some cases, over 60% of the code generated by LLMs fails to alert users to vulnerabilities and security risks [12, 7, 23, **?**].

There are two main factors that contribute to the generation of unsafe code with LLMs. First, LLMs are evaluated using benchmarks, which do not include constructs to evaluate the security of the generated code [3]. Second, existing evaluation metrics assess models' performance with respect to their ability to produce functionally correct code while ignoring security concerns [1, 6, 16, 11, 25].

Currently, there is no systematic approach to evaluating security improvements in LLMs, largely due to the absence of labeled datasets based on real user conversations. Most existing evaluation datasets are synthetically generated [9, 14, 3, 18], and some researchers, attempting to approximate user queries, have fed Stack Overflow questions into LLMs to examine their performance [26]. This leads to two key issues: First, limited realism, as synthetic datasets often lack coverage of diverse programming languages, user intentions, and real-world coding scenarios. Second, the risk of pre-trained bias, where the model may have already been exposed to similar web-sourced content during its original training, artificially inflating performance metrics and undermining the reliability and validity of evaluation results.

In addition, traditional NLP techniques, as applied in some previous studies, are insufficient for analyzing code, since code semantics differ significantly from natural language. This challenge becomes even more pronounced when datasets include code snippets written in various programming languages, making consistent and meaningful representation more difficult. To overcome this issue, the some previous studies use the common approach of Code Abstract Syntax Tree, Control flow or Data flow graph for analyzing codes[3].

Furthermore, some past studies examining the security of code generated by LLMs have relied on various security scanning tools that are largely language-dependent [18, 9]. This not only limits the applicability of the study results to a few specific programming languages but also raises concerns that the security awareness embedded in the trained models may be language-specific and not generalizable across different programming languages.

Recent research has increasingly focused on enhancing LLM models to improve the security of their generated code. The first group of previous studies on LLM-generated code has primarily focused on fine-tuning prompts to incorporate security considerations into user queries [4, 24, 26, 12]. However, relying solely on prompt engineering is inadequate as a security safeguard, as these approaches remain vulnerable to emerging adversarial threats such as data poisoning. Additionally, fine-tuned prompts can be subverted by attackers through prompt injection techniques, effectively bypassing the intended security improvements.

A second group of studies leverages Retrieval-Augmented Generation (RAG) models to inform LLMs about deprecated or insecure APIs, as well as recent vulnerability disclosures. However, these approaches are typically limited to specific Python libraries and narrowly scoped vulnerability reports. A third line of work, such as [14], explores training LLMs on curated secure coding datasets. While this method can enhance the model's awareness of secure practices, it requires frequent retraining to remain effective, a process that is both computationally expensive and time-consuming, thus limiting its practicality for real-world deployment.

All of these previous studies rely heavily on synthetic prompts or benchmark datasets, limiting the ecological validity of their findings. Our study fills this gap by leveraging the WildChat dataset to conduct the first empirical investigation grounded in authentic user conversations with ChatGPT. By examining how users request, interpret, and iterate on generated code—including cases involving insecure outputs, we offer insights not only into the model's security shortcomings but also into user behavior and intent. This dual perspective allows for a more comprehensive understanding of the risks and real-world dynamics surrounding LLM-assisted programming.

3 Construction of the Dataset

The basis for this study is the conversations related to the code or coding tasks present in the WildChat data set [27]. WildChat is a database of 1 million real-life conversations with different versions of ChatGPT collected between April 2023 and May 2024. We extracted the relevant codes and conversations from this dataset using the process illustrated in Figure 1. WildChat has already been used in more than 300 recent studies, and it has been shown to offer diverse, multilingual, and natural user prompts compared to other conversation datasets. The data set is anonymized, has undergone ethical review, and includes metadata such as timestamps and model versions, making it a reliable source for research. Its scale and diversity capture realistic conversational patterns, including ambiguous instructions, incomplete prompts, and a mixture of technical and non-technical queries. These qualities are particularly important for analyzing coding-related conversations in settings that reflect how users actually interact with LLMs.

Each conversation in WildChat is identified by a unique *conversation_hash* and consists of a sequence of user queries and ChatGPT responses within a session. As an initial step, we extracted the subset of conversations that contain code. In the dataset, code snippets are delimited by a single backtick (`) or by triple backticks (```), which are often, but not always, followed by the programming language used. Snippets enclosed in a single backtick are typically short fragments, often a single line of code, whereas those enclosed in triple backticks are longer and more complex code snippets. For the purpose of our analysis, we concentrated on the latter. Of a total of 837,989 conversations in the WildChat dataset, 82,843 contain code generated by ChatGPT. These code-containing conversations span multiple model versions, including `gpt-4-0314`(6.4%), `gpt-3.5-turbo-0301`(23.3%), `gpt-3.5-turbo-0613`(44.3%), `gpt-4-1106-preview`(12%), `gpt-3.5-turbo-0125` (6.9%), and `gpt-4-0125-preview`(7.4%).

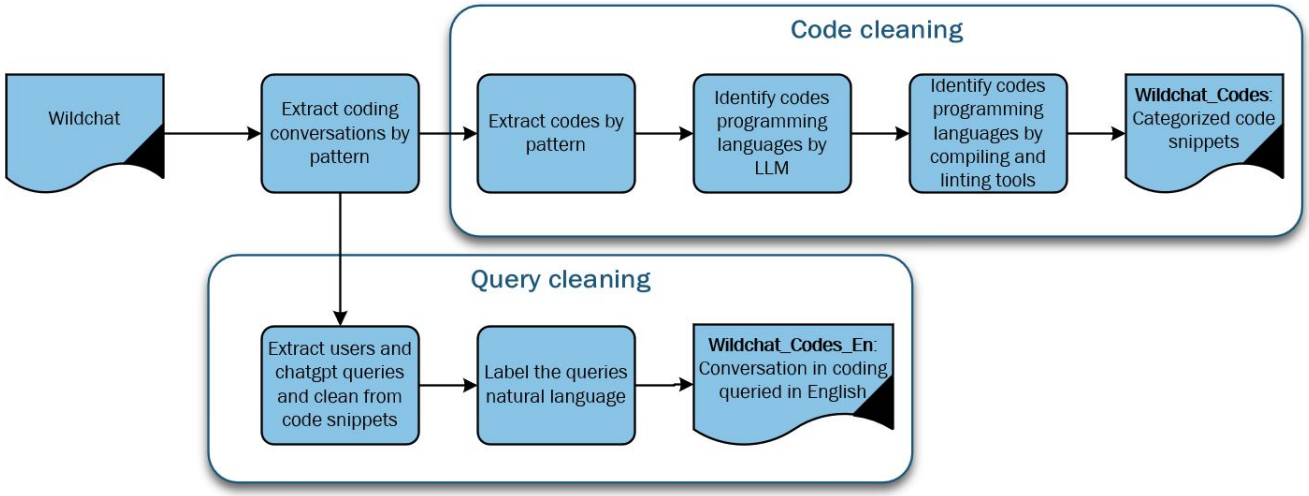


Fig. 1: Dataset generation pipeline

In the WildChat conversations, code snippets delimited by triple backticks usually begin with a programming language tag, followed by the code itself. However, these tags are often missing or incorrectly specified, so they cannot be relied on. To handle snippets without valid annotations, we applied a programming language identification model.[8] to automatically classify the code snippets by language. The model supports 26 programming languages with a reported accuracy of 95%, which implies that up to 5% of labels may still be incorrect. To further enhance labeling quality, we validated snippets for the six programming languages (those above the middle line in Table 1) using language-specific syntax checkers: we employed the py_compile module[9] for Python, eslint[10] for JavaScript, javac for Java, gcc for C and C++, the php -l command for PHP, and Microsoft's Roslyn compiler platform[11] for C#. This validation ensured syntactic correctness and improved the accuracy of the labeling, which is essential

[8] https://huggingface.co/philomath-1209/programming-language-identification
[9] https://docs.python.org/3/library/py_compile.html
[10] https://github.com/eslint
[11] https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis

for reliable downstream analysis. The resulting coding fragments, labeled by programming language, constitute the dataset named *WildChat_ Codes*, as illustrated in Figure 1.

As noted above, we also investigated users' coding intentions and their inquiries about coding issues. Since code-related conversations were not always conducted in English, it was necessary to first isolate English conversations for analysis. The WildChat dataset provides a 'Language' column; out of 82,843 conversations that contain code, 48,391 are labeled as being in English. However, we observed that many of these were, in fact, not in English, particularly when the conversation text contained numerous short code snippets (see, for example, the conversation with `conversation_hash` `e8e1274f7c1253299fa6c7865ee45703`). To ensure reliable language identification, we removed all code snippets from the conversation text and subsequently applied the Python langdetect library to uncover the true language of each conversation. Among the 82,843 conversations that contain code, 34,478 were identified as having queries in English. We focused on English conversations due to the broader availability of language processing tools and to enable the authors to manually verify and double-check the results with confidence. These English coding conversations constitute the dataset named *WildCode_ EN*, as illustrated in Figure 1.

Each conversation consists of a sequence of pairs: each consisting of a user query followed by a ChatGPT reply. To study the intentions of users and their interactions with ChatGPT's responses, we refer to the first user query as the **initial query**, with all subsequent user queries are referred to as **follow-up queries**. Similarly, the first ChatGPT response is defined as the **initial response**, and all subsequent replies are considered **follow-up responses**. An example sequence of queries and responses is shown in Table **??**, with the user's initial query and the initial ChatGPT response reproduced in the first row, and the follow-up queries and responses reported in subsequent rows.

4 Code Analysis

4.1 Overview of the Code Snippets

Table 1: Code stats

Language	Code Snippets	Conversations	Avg Snippets/Conv.	Avg. Lines/Snippet	Stddev. Lines/Snippet	Avg Comments/Snippet
C/CPP	7,526	3,911	1.92	43.73	38.61	3.18
C#	14,138	5,895	2.40	28.39	26.81	2.27
Java	18,680	7,228	2.58	27.43	27.89	1.94
JavaScript	15,943	7,217	2.21	23.39	25.15	2.23
PHP	449	340	1.32	20.60	13.42	3.05
Python	60,451	22,949	2.63	26.26	24.75	3.44
Rust	1,919	1,001	1.92	19.15	17.46	1.78
COBOL	1,378	1,022	1.35	15.84	20.00	0.06
Fortran	679	477	1.42	15.40	17.83	0.42
jq	2,805	1,146	2.45	14.23	6.46	0.06
Ruby	1,507	1,097	1.37	14.30	16.34	1.31
AppleScript	192	128	1.50	16.36	17.89	1.45
Kotlin	1,559	580	2.69	22.95	20.45	1.79
ARM Assembly	1,174	804	1.46	22.91	28.68	2.74
Erlang	2,225	1,491	1.49	15.36	17.55	0.59
Swift	819	552	1.48	14.77	12.82	1.11
R	1,555	730	2.13	13.94	13.02	2.38
PowerShell	6,375	2,856	2.23	16.68	10.47	0.49
Scala	1,692	1,111	1.52	13.33	11.87	0.94
Lua	401	288	1.39	12.34	9.95	2.05
Pascal	3,623	2,325	1.56	17.01	17.37	0.72
Go	1,631	768	2.12	35.49	34.43	2.89
Perl	1,692	1,255	1.35	12.55	13.05	0.84
Wolfram	1,225	783	1.56	15.80	15.86	1.20
.NET	3,707	1,637	2.26	27.45	20.47	4.49

Table 1 presents detailed statistics on code snippets extracted from the WildCode (shown in Figure 1) dataset, broken down by programming language. In this context, each code snippet within a conversation has been analyzed independently, and a single conversation may contain multiple code snippets. The table reports the average number

of snippets per conversation, the average number of lines per snippet, the average number of comments per snippet, and the standard deviation of the number of lines per snippet.

As can be seen in the table, Python dominates with 60,451 code snippets across 22,949 conversations, making it by far the most common language in the dataset. C/C++ code snippets are the longest, averaging 43.7 lines per code snippet with high variability (std. dev. 38.6). Moreover, among all languages, .NET has the highest average comments per block (4.49), and Python also ranks high (3.44), reflecting their widespread use and strong interest among users. The results show that LLMs predominantly generate short programs, though with substantial variation in length. Popular programming languages exhibit both a higher frequency of code blocks and a greater density of comments within the code. On average, each conversation contains between 1.5 and 2.5 code snippets, indicating that users frequently prompt the LLM for iterative refinements of previously generated code. The Intent section of this paper examines this phenomenon in greater detail and analyzes the corresponding follow-up categories.

4.2 Syntax Check

As explained in Section 3, we conducted a syntax check on a subset of code snippets in WildCode. This subset includes code written in the six programming languages, as listed above the dividing line in Table 1. More specifically, only on the snippets that were not labaeled by ChatGPT and had to be labeled by a classifier.

Table 2 reports the number of code snippets labeled by the model for each language. These correspond to the snippets for which ChatGPT did not initially provide a valid language tag in the ```` ```LANGUAGE ```` format, and thus required the classification model to determine their language. For each language, we also report the subset of snippets containing syntax errors identified by the linting process, and from this we infer the number of valid snippets, i.e. those without syntax errors. Note that no single PHP code snippet was valid, as none started with the `<?php` tag. A manual inspection further showed that the classified code snippets were all related to command-line tools associated with PHP rather than actual PHP source code.

Table 2: Number of snippets labelled by the model and valid (syntax-error-free) messages per language.

Language	**Labelled by ChatGPT**	**Labelled by Model**	**Code Snippets W/ Syntax Errors**	**Valid Code Snippets**
Python	60,451	57,371	19,805	37,566
Java	18,680	19,257	20	19,237
JavaScript	15,943	26,442	15,725	10,717
C#	14,138	13,893	6,517	7,376
C/CPP	7,526	21,510	18,323	3,187
PHP	449	1,935	1,935	0

To classify linting messages into syntax error categories, we first embedded the natural-language descriptions of the 20 predefined categories[12]. Each linting message was then embedded and assigned to the category with the highest cosine similarity to its description embedding. Table 3 presents the distribution of messages in categories and programming languages. We observe that C/C++ and C# dominate categories related to general programming issues (e.g. parsing errors or missing semicolons), while JavaScript, Python, and PHP contribute primarily to language-specific parsing errors.

Table 3: Aggregated syntax error categories across languages (total 177,732 rows).

Error Category	**Messages**	**C/CPP**	**C#**	**Java**	**JavaScript**	**Python**	**PHP**
Syntax Error	95,432	48,598	30,032	0	7,212	9,585	0
Declaration Error	33,026	15,683	11,013	0	2,801	3,529	0
Access Control Error	20,852	8,106	7,837	0	2,168	2,741	0
Preprocessor Error	18,377	5,390	5,172	100	2,759	3,021	1935
Lexical Error	2,829	1,244	885	0	323	377	0

[12] https://github.com/regularpooria/WildCode/blob/master/utils/error_categories.json

4.3 Security Analysis

We used OpenGrep[13], an open source tool to analyze the security issues of the code in our WildCode dataset. There is a repository of rules for OpenGrep[14] that includes security-related rules. These rules are defined using regular expressions that identify common insecure coding patterns across multiple programming languages. For our study, we selected the subset of security rules for the six programming languages represented in the WildCode dataset, resulting in a total of 648 rules. Each rule has a CWE mapping [15]. The distribution of these rules across programming languages is presented in Table 4 in column `Rules` for each category and for each language, and the list of rules is available in this file[16] on our GitHub

Table 4: Possible vulnerabilities by category and language.

Language	Hash Function			SQL Injection			RNG			Deserialization		
	CS	TO	Rules	CS	TO	Rules	CS	TO	Rules	CS	TO	Rules
C#	24	56	3	124	624	1	91	133	1	56	82	11
Java	32	93	16	223	1373	11	150	232	1	273	634	15
JavaScript	22	72	4	163	932	9	352	987	1	463	805	5
PHP	17	73	5	183	873	4	N/A	N/A	N/A	N/A	N/A	N/A
Python	180	577	22	883	6985	17	2810	12791	3	2258	4833	8

CS: Code Snippets | TO: Total occurrences | Rules: Number of detection rules

Table 4 presents the distribution of code snippets mapped to regular expression rules in four vulnerability categories: hash functions, SQL injection, random number generation (RNG) and deserialization. In the table, **TO** (Total Occurrences) denotes the total number of rule matches, while **CS** (Code Snippets) indicates the number of unique code snippets in which at least one rule from the corresponding category was identified. Because a single code snippet can match (i.e., violate) multiple rules, the values in **TO** are always greater than or equal to those in **CS**. Note that entries marked N/A indicate that no corresponding rules exist for that programming language and that C/C++ does not apply to the table. These rules are detailed in the following subsections.

Note: In tables 5, 6, 7, 8; The sum of unique conversation hashes across individual rules may be greater than the overall unique hashes for the language, because some conversation hashes violate multiple rules. Total occurrences are additive.

4.3.1 Weak Cryptographic hash functions: A total of 264 unique ChatGPT-generated conversations contain code snippets referencing hash functions across Python, Java, C, C#, JavaScript, and PHP. Using a set of 50 static analysis rules tailored to detect insecure or improper use of hash functions, we found that 54 conversations triggered at least one rule, representing a vulnerability rate of 20.61%. Notably, only 13 out of the 50 rules were activated, with the majority of violations stemming from continued use of MD5, SHA1, or cryptographic algorithms lacking authentication guarantees. Table 5 shows all vulnerabilities found in the said conversations.

[13] https://github.com/opengrep/opengrep
[14] https://github.com/regularpooria/opengrep-rules
[15] https://cwe.mitre.org/
[16] https://github.com/regularpooria/WildCode/blob/master/utils/rules.json

Table 5: Top Hash Function Vulnerabilities in ChatGPT-Generated Codes By Language

Language	Rule	CWE	Unique Files	Total Occurrences
Java	use-of-md5	CWE-328	7	8
	desede-is-deprecated	CWE-326	1	1
	des-is-deprecated	CWE-326	1	2
	ecb-cipher	CWE-327	1	2
	use-of-aes-ecb	CWE-327	1	2
	use-of-default-aes	CWE-327	1	2
	Overall		**10 (31.25%)**	**17**
PHP	weak-crypto	CWE-328	2	2
	openssl-decrypt-validate	CWE-252	1	1
	Overall		**3 (17.65%)**	**3**
Python	insecure-hash-algorithm-md5	CWE-327	33	59
	insecure-hash-algorithm-sha1	CWE-327	4	6
	crypto-mode-without-authentication	CWE-327	4	5
	insecure-cipher-algorithm-des	CWE-327	1	2
	md5-used-as-password	CWE-327	2	2
	Overall		**41 (22.78%)**	**74**

4.3.2 SQL Injection: We examined 970 LLM-generated conversations that contained SQL-related code snippets through regex patterns, applying 42 rules targeting common patterns of SQL injection vulnerabilities. Only seven rules were triggered, resulting in 61 logged conversations, a vulnerability rate of 3. 93%. The most frequently violated rule involved the execution of raw SQL queries in SQLAlchemy, followed by tainted string concatenation and JDBC usage patterns. Table 6 shows the rules and their occurrences.

Table 6: SQL Injection Vulnerabilities in ChatGPT-Generated Codes By Language

Language	Rule	CWE	Code Snippets	Total Occurrences
C#	csharp-sqli	CWE-89	3	3
	Overall		**3 (2.42%)**	**3**
Java	jdbc-sqli	CWE-89	4	9
	Overall		**4 (1.79%)**	**9**
JavaScript	tainted-sql-string	CWE-915	2	3
	Overall		**2 (1.23%)**	**3**
PHP	tainted-sql-string	CWE-915	7	9
	Overall		**7 (12.50%)**	**9**
Python	tainted-sql-string	CWE-915	2	7
	sqlalchemy-execute-raw-query	CWE-89	42	70
	psycopg-sqli	CWE-89	5	5
	sql-injection-db-cursor-execute	CWE-89	3	15
	avoid-sqlalchemy-text	CWE-89	2	2
	Overall		**45 (5.40%)**	**99**

4.3.3 Weak random number generation: We examined 3032 conversations that contained code that uses random number generation. If the generated random number is used for a security-sensitive task, such as creating a password or a cryptographic nonce, then the underlying random number generation algorithm must be cryptographically secure. Otherwise, a vulnerability is present in the code. We applied this analysis to Java, C#, JavaScript, PHP and Python and found 17 instances where weak random number generation was found. Of these, 15 were present in Java code and 2 in Python. Thus, only 0.47% of the code snippets contained this specific vulnerability, a result that is significantly better than the one obtained for other classes of vulnerabilities.

4.3.4 Deserialization attacks: A deserialization vulnerability is present whenever the code processes serialized data from an untrusted source, without including proper validation and security checks. This gives a malicious adversary the ability to input arbitrary data, perform a denial of service, or even execute arbitrary code [20]. This

is one of the main security issues in Java programs, a fact that was made evident by the devastating Log4Shell vulnerability in 2021 [10].

Java programs present in the Wildchat dataset include 30 instances of deserialization. A manual inspection revealed that every single case seemed vulnerable to deserialization attacks, as none contained security checks. Furthermore, in none of the conversations associated with these programs did ChatGPT discuss the risks inherent in deserializing data.

Table 7: Unsafe Deserialization in ChatGPT-Generated Codes By Language

Language	Rule	CWE	Scripts	Total Occurrences
Java	documentbuilderfactory-disallow-doctype-decl-missing	CWE-611	3	3
	object-deserialization	CWE-502	21	33
	transformerfactory-dtds-not-disabled	CWE-611	1	1
	saxparserfactory-disallow-doctype-decl-missing	CWE-611	5	7
	use-snakeyaml-constructor	CWE-502	1	3
	Overall		**30 (10.99%)**	**47**
JavaScript	grpc-nodejs-insecure-connection	CWE-502	7	20
	Overall		**7(1.51%)**	**20**
Python	marshal-usage	CWE-502	2	3
	Overall		**2(0.09%)**	**3**

4.3.5 Memory Safety: In our previous paper [12], ChatGPT exhibited particular difficulty with memory corruption vulnerabilities in C/C++ programs. This is also the case for programs present in the Wildchat data set. We focus on 6 rules, which forbid the use of constructs that are known to be easily exploitable, namely `scanf`, `strcpy`, `memset`, `strcat`, gets `printf`. Using any of these functions incurs a risk of a buffer overflow, unless accompanied by thorough boundary checks, or unless the input is completely controlled by the program. As shown in Table 8, the C/C++ programs combine 1807 distinct violations of these 6 rules, across 581 distinct code blocks. In general, 14.85 % of the C/C++ programs contain at least 1 violation. Many code fragments contained multiple occurrences, often several dozen distinct occurrences. The continued use of these function calls is particularly disappointing, since in most cases memory-safe alternatives exist.

Table 8: Unsafe Memory in ChatGPT-Generated Codes By Language

Language	Rule	CWE	Code Snippets	Total Occurrences
C/CPP	insecure-use-scanf-fn	CWE-676	378	1182
	insecure-use-memset	CWE-14	117	263
	insecure-use-string-copy-fn	CWE-676	120	273
	insecure-use-strcat-fn	CWE-676	18	56
	insecure-use-gets-fn	CWE-676	11	19
	insecure-use-printf-fn	CWE-134	5	14
	Overall		**581 (14.85%)**	**1807**

This is almost certainly an undercount of the actual number of vulnerable programs, since even in the absence of these functions, memory corruption can still occur because of errors in pointer arithmetic or memory management. We also found that programs that do not contain memory management errors are substantially shorter than those that do, with an average of 48 lines in the former case (Std. dev. 40) versus 106 (std. dev. 58) in the latter case. That said, a program of 100 lines is hardly a "large " program by any definition, and the inability of the model to create a memory-safe program of even such a short size is disheartening.

Interestingly, despite the fact that multiple C-rules are triggered hundreds of times, 3 rules are never triggered: a rule forbidding freeing a pointer twice and two rules related to use-after-free of pointers. Violations of these rules usually occur in larger code-bases, when the same pointer variable is used in several different functions or in several different files. LLMs are still limited in the size of the programs they can produce, and these two types of vulnerability can usually be easily avoided when writing a small code fragment. If a novice programmer intends to create a larger program by separately requesting several fragments from the LLMs and joining them together, it is likely that these vulnerabilities may be present in the final code.

4.4 ReDoS (Regular Expression Denial of Service)

In our previous research [12], one of the security issues for which ChatGPT seemed to struggle the most is the problem of ReDoS attacks [19]. It is one of a handful of cases in which the model is unable to recognize the presence of the vulnerability even when explicitly prompted on the topic.

To estimate the prevalence of vulnerable regular expressions in code generated by ChatGPT, we extracted every regexe from the programs in our dataset. We then analyzed these expressions using four different reDoS vulnerability detection tools. The tools used are:

- `Saferegex` [13] : A static analysis tool developed by Microsoft which reject regex that contain any one of a number of patterns known to be vulnerable;
- `Rescue` [21] : A dynamic testing tool that generates instances of malicious input using a gray-box approach. Rescue may either mark a regex as vulnerable or invulnerable, or timeout (T.O.), an indication that the regex is likely vulnerable;
- `Redoshunter` [15]: A hybrid (static and dynamic) tool that detects up to 5 types of vulnerable patterns (EOD, EOA, POA, NQ and SLQ), defined by the author;
- `Revealer` [17]: A hybrid (static and dynamic) tool that classifies vulnerable regexes as either susceptible to polynomial (poly.) or exponential (expo.) exploitation.

According to this analysis, about a third of the regex present in the code in our dataset is susceptible to ReDoS attacks. Despite the possibility that these tools may generate false positives, this analysis is likely an overcount of the actual incidence of ReDoS vulnerabilities in the generated code snippet. Code can safely use a vulnerable regex if it does not manipulate untrusted user input, if the regex validation algorithm is not susceptible to ReDoS attacks, or if the validation is bounded by a timeout. However, this analysis provides us with a baseline indication of the prevalence of ReDoS vulnerabilities in our data set.

The results are presented in Table 9. For each programming language, we report the number of extracted regexes and the vulnerabilities identified by each detection tool.

Only two English-language conversations explicitly mentioned the ReDoS attack, and never in the context of writing secure code. In one case, a user reproduced a `npm` report which referred to CVE-2022-3517 (a ReDoS vulnerability) and asked how to respond to it. In the second case, ReDoS was part of a list of 100 vulnerability types for bug bounty programs. There was no instance of an LLM commenting that a regex (either produced by the user or by itself) is susceptible to this class of attack.

Table 9: Detection of Vulnerable Regexes by Different Tools.

Language	Total Regex	SafeRegex	Rescue		ReDoSHunter	Revealer		Total Vuln.
			Vuln.	T.O.		Poly.	Expo.	
C/CPP	80	40	2	14	10	1	0	36 (45%)
C#	46	27	0	5	5	0	0	27 (58.7%)
Java	226	160	2	37	19	2	0	47 (20.7%)
JS	50	16	2	9	11	1	2	16 (32%)
PHP	38	11	0	8	1	0	0	11 (28.9%)
Python	753	309	20	290	222	10	0	212 (28.1%)
Total	1,203	568	26	375	268	14	2	354 (29.4%)

4.5 Hallucinations

A important security vulnerability in code generated by LLMs is the persistent presence of 'package hallucinations', package names that the LLM includes in the code, but do not actually exist. This opens pathways for malicious adversaries to exploit the program by creating libraries with the names hallucinated by the LLM, and including malicious code in those libraries. Alternatively, the code may require manual modifications in order to run properly, with the attendant risk of introducing vulnerabilities, as discussed in [12, 22].

We focus our analysis on Python and JavaScript due to their widespread adoption and the strong ecosystem support provided by package repositories, PyPI for Python and NPM for JavaScript. These repositories allow for

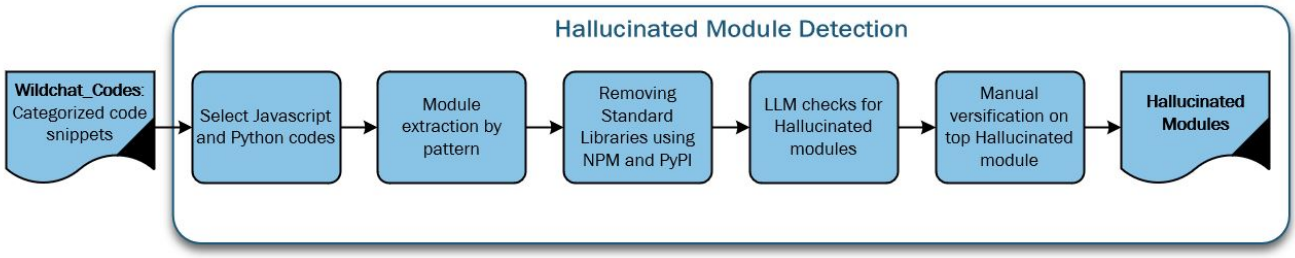


Fig. 2: Hallucination detection pipeline

systematic validation of third-party modules. Our pipeline for detecting hallucinated modules, illustrated in Figure 2, begins by extracting imported module names using a regular expression. We then filter these candidates against the official PyPI and NPM package lists, discarding any modules not present in the repositories. Standard library modules are similarly excluded. Finally, as an additional safeguard, we query an LLM with Web Search capabilities to flag potentially invalid modules, which we manually review and remove if confirmed to be incorrect.

Our analysis identified 285 distinct hallucinated Python modules among 1,984 modules used in ChatGPT responses (approximately 14.4%), with 210 (73.7%) appearing only once. In comparison, 21 distinct JavaScript packages were hallucinated out of 606 (approximately 3.5%), with 13 appearing only once. This pattern indicates that the model predominantly generates unique fictitious modules in Python, particularly when handling less familiar coding tasks. Interestingly, module names that occur in hundreds of code snippets are never hallucinations. Instead, the model tends to produce imaginary modules when faced with less common or user-defined functionalities. Each hallucinated module name occurs less than 50 times in the dataset. Tables 10 and 11 list the names of the most frequently hallucinated modules.

Table 10: Hallucinated Python libraries generated by ChatGPT and their properties

Name	Occurrences	Notes
`your...`	53	Examples: `your_module`, `your_gui`, `yourapp`
crm_app	19	A generic name for a Customer Relationship Manager
Autodesk	13	The only official python SDK for Autodesk is `shotgunsoftware`
some_module	11	A placeholder name that the LLMs uses when generating ambiguous code
create_image_with_text	8	Non-existent; hallucinated for text-to-image generation libraries
openai_secret_manager	7	Non-existent
ENM_class	6	Non-existent
radar_tracking	5	Similar libraries exist but not this specific name
universo_fisico	5	Non-existent
mail	4	Non-existent; Similar packages exist but none with this name

Table 11: Hallucinated JavaScript libraries generated by ChatGPT and their properties

Name	Occurrences	Notes
Penduel	15	Non-existent; Seems to be a .Sol related use case but no package exist with this name. Might be a local file
your-keycloak-library	8	Non-existent; Similar libraries exist for Keycloak integration but not with this exact name
Shares	3	Non-existent; Similar packages exist but none with this name
LoanNFT	2	Non-existent; NFT-related packages exist but not with this exact name
@colyseus/nomination	2	"nomination" does not exist in the colyseus package
PlayerManagement	2	Non-existent; Similar game/player management libraries exist but not this specific one

5 User Intent

In this section, we investigate how users engage with ChatGPT in code-related conversations, focusing on four complementary dimensions: (i) ChatGPT's implicit programming language preferences when users do not specify a language, (ii) the intentions that underlie user queries, (iii) the relationship between user intent and the total length and depth of conversations, and (iv) the extent to which users address security concerns when interacting with the code generated by ChatGPT. By analyzing these aspects in the conversations related to codes in Wildchat, our goal is to uncover patterns in user behavior, identify gaps in security awareness, and better understand how ChatGPT mediates coding practices across multi-turn dialogues.

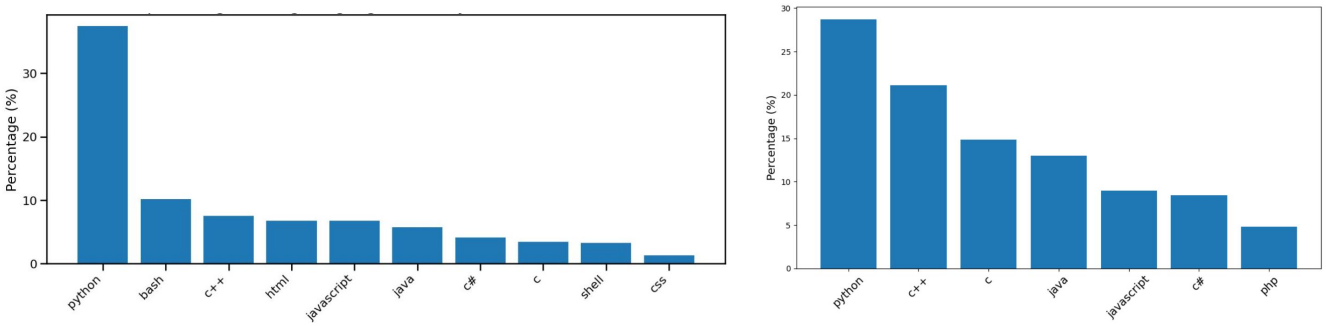
To better understand the intentions of users and their interpretation of coding issues, we restrict our analysis to queries formulated in English. These queries span a wide spectrum of concerns, including code understanding, bug identification and resolution, debugging practices, security vulnerabilities, and performance optimization. For this purpose, we have used the WildCode_EN dataset, which contains the English conversations from the WildChat dataset that also contain code, as explained in Section 3.

5.1 ChatGPT programming language preference

Our first analysis investigates ChatGPT's default programming language choices when users omit specifying a language in their requests. The goal is to understand the implicit defaults of ChatGPT in code generation, as these defaults shape the coding environment presented to users, influence the accessibility of generated solutions, and may reveal underlying model biases toward certain languages (e.g., Python). For this analysis, we use the *WildCode* dataset. The programming languages of the code snippets in these conversations were determined based on the language labels assigned to the code snippets in *WildCode*, as described in Section 3. We extracted all conversations in which the user did not specify any programming language in their *initial query*, while ChatGPT's response included a code snippet.

Figure 3a illustrates the distribution of programming languages in the code snippets generated by ChatGPT during conversations where the user did not specify any programming language. As can be seen, ChatGPT exhibits a marked preference for writing Python code, which accounts for more than one-third of all generated code, followed by Bash, C++, HTML, and JavaScript. This suggests a strong default preference or widespread applicability of Python in initial coding tasks. A sample conversation is provided in the project's Github.

Figure 3b shows the distribution of the programming languages requested by the users in *follow-up queries*. While Python remains the most frequently mentioned language; C++, C, and Java are also commonly requested when the code initially generated by ChatGPT was in a different language. This pattern indicates that users often shift programming languages during multi-turn interactions, possibly due to evolving task requirements or preferences.



(a) Top 10 programming languages used by ChatGPT in initial code generation.

(b) Languages newly requested by users in follow-up queries.

Fig. 3: Comparison of programming languages in initial code generation vs. user-requested follow-up.

5.2 Users' intentions in code related queries

In the next stage of our analysis, we investigate users' intentions in code-related queries, with particular emphasis on follow-up messages. Understanding these intentions is crucial for characterizing how users engage with ChatGPT

beyond their initial requests, as follow-up queries often reveal deeper goals such as clarifying outputs, fixing errors, or adapting code to new requirements.

To understand users' intentions behind their code-related queries, we defined a set of categories that represent common types of coding requests. The definitions of these categories, along with representative keywords expected in user queries, are provided in Table 12. For classification, we first removed code snippets from user messages, retaining only the natural language text. We then applied zero-shot classification using the `bart-large-mnli`[17] model, leveraging the category definitions and example keywords as candidate labels for intent detection. The model produced a probability distribution over the predefined categories for each query. To ensure fairness when multiple categories received nearly identical confidence levels, the probabilities were rounded to two decimal places, and all categories that shared the maximum rounded probability were selected. This tie-aware approach captures cases where the model could not clearly differentiate between categories, instead of arbitrarily selecting only one. Using this methodology, we derived three types of labels for each conversation: (i) the initial category based on the user's first query, (ii) the primary follow-up category corresponding to the second query in the query sequence in a conversation with ChatGPT, and (iii) aggregated follow-up categories in subsequent queries except for the initial.

Table 12: Categories of User Code Queries with Explanations and Example Keywords

Category	**Explanation**	**Example Keywords**
Bug Fixing	Requests to identify, explain, or fix bugs and incorrect behavior in existing code.	bug, error, exception, crash, traceback, debug, fix, issue
Code Explanation	Inquiries aimed at understanding how a specific piece of code works or what it does.	explain, understand, what, how, walkthrough, describe, purpose, logic, meaning
Code Generation	Requests to write or generate new code from scratch, often with a specific task or functionality in mind.	write, develop, implement, create, build, construct, design, compose, make
Code Translation	Requests to convert or rewrite code from one programming language to another.	convert, translate, migrate, port, adapt, switch, language change
Setup/Deployment	Questions related to deploying code, using DevOps tools, CI/CD pipelines, Docker, or cloud infrastructure.	deploy, set up, docker, install, ci/cd, container, cloud, heroku, kubernetes, configure, virtualenv
Library/API Usage	Questions on how to use specific libraries, modules, or APIs in code.	api, module, package, library
Optimization	Requests to improve the performance, readability, or efficiency of existing code.	improve, enhance, optimize, reduce, fast, efficient, time, complexity
Secure Coding	Requests to enhance the security of code, identify vulnerabilities, or follow best security practices.	secure, validation, verify, prevent, protect, sanitize, vulnerability, security, attack, encryption, safe code

Figure 4 presents the distribution of user intents across these three contexts. *Bug Fixing* and *Code Generation* emerge as the most frequent categories, reflecting that practical coding support is the predominant concern at the start of user interactions with ChatGPT. Similarly, in follow-up queries, *Bug Fixing*, *Code Generation*, and *Setup/Deployment* remain dominant. In contrast, categories such as *Secure Coding* and *Optimization* appear much less frequently, suggesting that these considerations are less commonly prioritized during user–assistant exchanges.

Table 13: Most Common Follow-up Categories by Initial Query Type

Initial Category (Count)	**Most Common Follow-up Category**
Bug Fixing (8017)	Bug Fixing (5034), Code Generation (2983)
Code Explanation (510)	Bug Fixing (292), Code Generation (218)
Code Generation (8017)	Code Generation (4126), Bug Fixing (3891)
Code Translation (783)	Bug Fixing (456), Code Generation (327)
Library/API Usage (593)	Bug Fixing (360), Code Generation (233)
Optimization (383)	Bug Fixing (312), Secure Coding (71)
Secure Coding (21)	Code Explanation (11), Bug Fixing (10)
Setup/Deployment (1535)	Bug Fixing (949), Code Generation (586)

[17] https://huggingface.co/facebook/bart-large-mnli

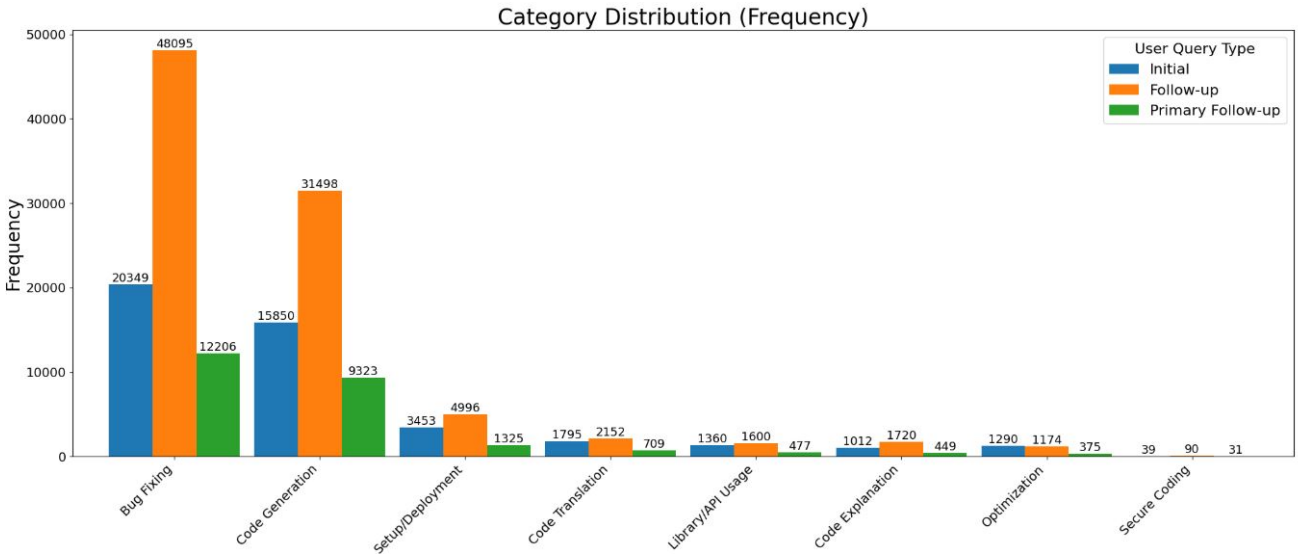


Fig. 4: Predicted category distribution for user queries.

We also examined the relationship between users' initial query categories and the categories of their follow-up messages. Table 13 presents the most common follow-up categories associated with each initial query type. A clear pattern emerges: *Bug Fixing* and *Code Generation* dominate the follow-up space across nearly all initial categories. This indicates that, regardless of the original intent, users frequently transition toward refining existing code or requesting new code during their interactions. Notably, *Code Generation* appears frequently both as an initial intent and as a follow-up category, highlighting the iterative nature of coding workflows with ChatGPT.

Interestingly, when *Secure Coding* is the initial focus, it rarely results in continued security-related discussions; suggesting a gap in users' sustained engagement with secure development practices. Conversely, follow-up queries to *Optimization* tasks occasionally involve *Secure Coding*, implying that some users perceive a connection between performance and security considerations in code quality.

To statistically assess these patterns, we applied the chi-square test of independence. The test results revealed no significant association between initial and follow-up categories, indicating that despite the apparent trends, such as the dominance of *Bug Fixing* and *Code Generation*, these follow-up intent categories are not strongly dependent on the user's initial query category.

5.3 Impact of Query Category on Conversation Length

We next examine how the category of a user's initial or follow-up query influences the overall length of the conversation with ChatGPT. This analysis provides insight into whether certain types of coding requests, such as bug fixing, code explanation, or secure coding, tend to generate more extended interactions, while others can be resolved more quickly. Here, **conversation length** is measured as the number of user–ChatGPT query–response pairs within a conversation. By linking query categories to conversation depth, we aim to better understand the dynamics of multi-turn dialogues and the factors that drive longer or shorter exchanges.

Figure 5 illustrates the distribution of conversation lengths, measured by the total number of messages exchanged per conversation, grouped by initial and follow-up categories, respectively. These visualizations provide insight into how the nature of a user's request influences the depth and complexity of the ensuing interaction. As shown in Figure 5, conversations that begin with *Secure Coding* tend to have the highest median and widest range in message counts, suggesting that security-focused topics often prompt more extensive discussions. In contrast, queries related to *Bug Fixing* and *Code Translation* are typically resolved in shorter conversations, indicating these are more concise or well-scoped tasks.

Figure 5 shows that when the follow-up category is *Code Explanation* or *Bug Fixing*, conversations tend to be longer, potentially due to the need for iterative clarification or detailed reasoning. Meanwhile, follow-up requests involving *Secure Coding* exhibit fewer messages, implying that even when security is addressed later in a conversation, users do not typically engage in extended dialogue on that topic.

Overall, the number of messages exchanged appears to reflect the perceived complexity or ambiguity of the task, with explanatory and security-related queries tending to foster longer, more in-depth interactions.

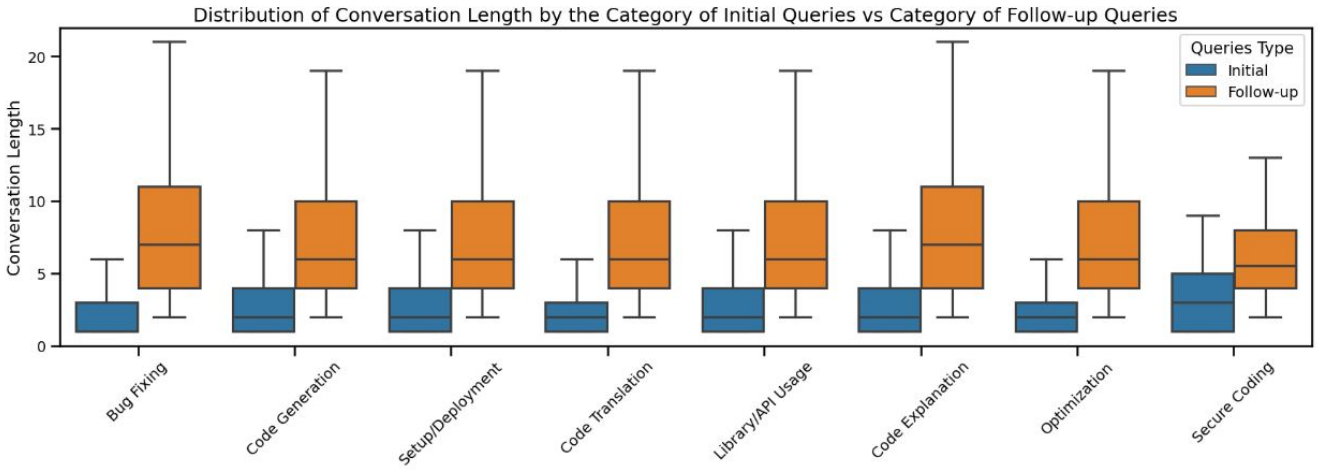


Fig. 5: Distribution of conversation length grouped by initial and followup message category.

5.4 Security Awareness in Code-Related Conversations

We now turn our attention to the extent to which users address security concerns in their interactions with ChatGPT-generated code. While prior analyses have shown that users frequently focus on practical tasks such as code generation and bug fixing, it remains unclear how often security considerations enter these conversations. By examining both initial query intents and follow-up queries, we aim to identify whether users explicitly engage with secure coding practices, how often security arises in multi-turn dialogues, and whether it is sustained throughout the interaction. This analysis provides critical insight into the role of security awareness in AI-assisted coding workflows.

As mentioned in Section 4, we analyzed code snippets generated by ChatGPT using *OpenGrep* to identify instances containing errors. Out of 48,391 conversations that include code, the code generated by ChatGPT in 1,562 conversations were flagged by OpenGrep as having at least one error. Among these, 1,214 conversations were conducted in English, which we used as the basis for our intent analysis.

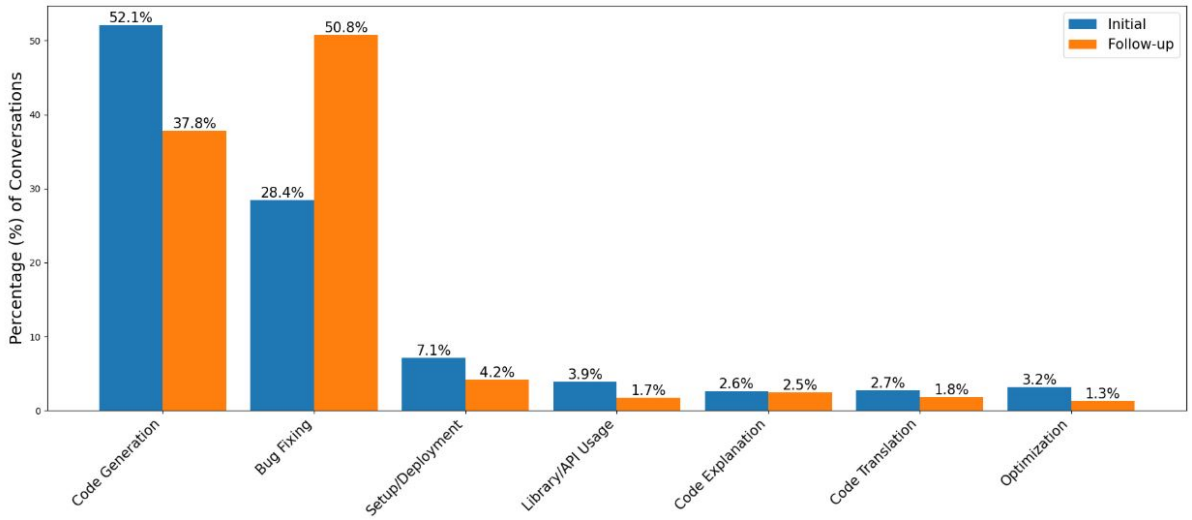


Fig. 6: Predicted Category for User Query in Conversations with Buggy Codes

Figure 6 presents the distribution of the predicted intent category for initial and follow-up queries from the user in these buggy code conversations. A key observation is that while *Code Generation* and *Bug Fixing* dominate both initial queries and follow-up actions, *Secure Coding* is extremely rare, with only 6 instances in all follow-ups. This suggests a noticeable gap in user emphasis on security-related intents, highlighting the need for increased security awareness and better integration of secure coding practices in AI-assisted development workflow.

5.5 User Discussion on Hallucinated Modules

As discussed in Section 4.5, ChatGPT generated non-existent (hallucinated) modules in some of the produced code snippets. We manually examined the list of conversations[18] where these modules were mentioned, and the conversations were conducted in English. In none of these cases did the users suspect that the module might be fake; instead, they typically continued asking about the errors they encountered. Likewise, ChatGPT did not self-correct or indicate that the issue could originate from a non-existent module. An example of such a conversation involving a hallucinated module is presented in our GitHub.

6 Discussion and Threat to Validity

Overall, this study suggests an overall low quality to the code generated by ChatGPT, especially with respect to the security aspects of the code. It remains to be seen whether LLMs dedicated to coding (e.g. Github Copilot) , or if more recent versions of the model produce higher quality code.

In most cases, our analysis captures an overcount of the actual number of vulnerabilities of each type present in the code. For example, unsecured deserialzation, or vulnerable regexes, can be safely used if the programmer is certain that a malicious adversary can never control the input. Conversely, our analysis of C/C++ memory corruption vulnerabilities is certainly an undercount, since it excludes several classes of vulnerable patterns.

Our study of user intentions excludes conversations conducted in languages other than English. Although this ensures consistency in intent classification, it can introduce bias by omitting multilingual users and their potentially distinct patterns of interaction. In contrast, for the code security analysis, we considered all code generated by ChatGPT regardless of the natural language of the user's query. This difference in scope could limit the comparability between the two analyzes.

The WildChat dataset does not capture cases where users uploaded code as images (e.g., screenshots). As a result, some code-related interactions may be missing, potentially underestimating the overall volume and diversity of coding activity in the dataset.

When examining the conversations, we found a small number of cases in which the user specifically requests a vulnerable code. For example, a user asked for "an example of a C program vulnerable to buffer overflow attacks". The presence of code that is vulnerable by design could be a threat to the validity of our study. However, such requests are sufficiently infrequent that it is unlikely that they invalidate our results. It is worth remarking that while ChatGPT often refuses to create attack code on behalf of a user, it does not hesitate to create vulnerable code. This is a curious ethical choice: indeed, while attack code may have value as an academic tool or for pentesting, vulnerable code has no benefit.

7 Conclusion

In this paper, we present the first large-scale empirical study of code generated by ChatGPT based on real-world user interactions drawn from the WildChat dataset. Unlike previous research that relies on synthetic prompts or benchmark datasets, our analysis leverages 82,843 authentic conversations in which ChatGPT generates code, allowing us to assess both the quality of the code and the intentions of the users who request it.

Our findings highlight several concerning trends. First, code quality, particularly in terms of security, remains a significant issue. Tools such as OpenGrep and ReDoS detectors reveal that vulnerabilities such as insecure cryptographic functions, SQL injection risks, unsafe memory operations, and hallucinated package imports are widespread in ChatGPT-generated code. In particular, approximately one-third of regex expressions were found to be susceptible to ReDoS attacks, and 14.85% of C/C++ snippets contained memory safety violations. Second, our analysis of user intent shows that security is rarely prioritized in user queries. Even when users encounter buggy or vulnerable code, they seldom raise security concerns or request secure alternatives.

Together, these contributions underscore the urgent need for security-aware LLMs, better user prompting strategies, and proactive safeguards within generative coding tools.

References

1. Ashrafi, N., Bouktif, S., Mediani, M.: Enhancing llm code generation: A systematic evaluation of multi-agent collaboration and runtime debugging for improved accuracy, reliability, and latency. arXiv preprint arXiv:2505.02133 (2025)

[18] Available on our GitHub repository: https://github.com/regularpooria/WildCode

2. AWS DevOps Blog: Introducing amazon codewhisperer dashboard and cloudwatch metrics. https://aws.amazon.com/blogs/devops/introducing-amazon-codewhisperer-dashboard-and-cloudwatch-metrics/ (2024)
3. Bai, W., Xuan, K., Huang, P., Wu, Q., Wen, J., Wu, J., Lu, K.: Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. arXiv preprint arXiv:2409.16526 (2024)
4. Bruni, M., Gabrielli, F., Ghafari, M., Kropp, M.: Benchmarking prompt engineering techniques for secure code generation with gpt models. arXiv preprint arXiv:2502.06039 (2025)
5. Etsenake, D., Nagappan, M.: Understanding the human-llm dynamic: A literature survey of llm use in programming tasks. ArXiv **abs/2410.01026** (2024), https://api.semanticscholar.org/CorpusID:273026291
6. Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Lahiri, S.K.: Llm-based test-driven interactive code generation: User study and empirical evaluation. IEEE Transactions on Software Engineering (2024)
7. Fu, Y., Liang, P., LI, Z., SHAHIN, M., YU, J., CHEN, J.: Security weaknesses of copilot-generated code in github projects: An empirical study. ACM Transactions on Software Engineering and Methodology (2025)
8. GitHub: Survey reveals ai's impact on the developer experience. https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/ (2025), accessed: 2025-05-26
9. He, J., Vechev, M.: Large language models for code: Security hardening and adversarial testing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1865–1879 (2023)
10. Hiesgen, R., Nawrocki, M., Schmidt, T.C., Wählisch, M.: The race to the vulnerable: Measuring the log4j shell incident. arXiv preprint arXiv:2205.02544 (2022)
11. Huang, D., Zhang, J.M., Bu, Q., Xie, X., Chen, J., Cui, H.: Bias testing and mitigation in llm-based code generation. ACM Transactions on Software Engineering and Methodology (2024)
12. Khoury, R., Avila, A.R., Brunelle, J., Camara, B.M.: How secure is code generated by chatgpt? In: 2023 IEEE international conference on systems, man, and cybernetics (SMC). pp. 2445–2451. IEEE (2023)
13. Kutner, J.: Saferegex. https://github.com/jkutner/saferegex (2018)
14. Li, D., Yan, M., Zhang, Y., Liu, Z., Liu, C., Zhang, X., Chen, T., Lo, D.: Cosec: On-the-fly security hardening of code llms via supervised co-decoding. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1428–1439 (2024)
15. Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L., Cheung, S.C.: {ReDoSHunter}: A combined static and dynamic approach for regular expression {DoS} detection. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3847–3864 (2021)
16. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems **36**, 21558–21572 (2023)
17. Liu, Y., Zhang, M., Meng, W.: Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1468–1484. IEEE (2021)
18. Nazzal, M., Khalil, I., Khreishah, A., Phan, N.: Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 2266–2280 (2024)
19. OWASP Foundation: Regular expression denial of service (redos). https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS (2025), accessed: 2025-08-04
20. Sayar, I., Bartel, A., Bodden, E., Le Traon, Y.: An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. ACM Transactions on Software Engineering and Methodology **32**(1), 1–45 (2023)
21. Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., Lu, J.: Rescue: crafting regular expression dos attacks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 225–235. ASE '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3238147.3238159
22. Spracklen, J., Wijewickrama, R., Sakib, A.N., Maiti, A., Viswanath, B.: We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 3687–3706 (2025)
23. Tihanyi, N., Jain, R., Charalambous, Y., Ferrag, M.A., Sun, Y., Cordeiro, L.C.: A new era in software security: Towards self-healing software via large language models and formal verification. arXiv preprint arXiv:2305.14752 (2023)
24. Tony, C., Ferreyra, N.E.D., Mutas, M., Dhiff, S., Scandariato, R.: Prompting techniques for secure code generation: A systematic investigation. arXiv preprint arXiv:2407.07064 (2024)
25. Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E.: Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. arXiv preprint arXiv:2304.10778 (2023)
26. Zeng, B., Zhang, Q., Zhou, C., Go, G., Jiang, Y., Shi, H.: Inducing vulnerable code generation in llm coding assistants. arXiv preprint arXiv:2504.15867 (2025)
27. Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., Deng, Y.: Wildchat: 1m chatgpt interaction logs in the wild. arXiv preprint arXiv:2405.01470 (2024)