# Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code

Fang Liu[†], Yang Liu[†], Lin Shi[‡], Zhen Yang[§], Li Zhang[†]*, Xiaoli Lian[†], Zhongqi Li[¶], Yuchi Ma[¶]
[†]State Key Laboratory of Complex & Critical Software Environment (SKLCCSE)
School of Computer Science and Engineering, Beihang University, Beijing, China
[‡]School of Software, Beihang University, Beijing, China
[§]School of Computer Science and Technology, Shandong University, Qingdao, China
[¶]Huawei Cloud Computing Technologies Co., Ltd, China
{fangliu, liuyang26, shilin}@buaa.edu.cn, zhenyang@sdu.edu.cn, {lily, lianxiaoli}@buaa.edu.cn
{lizhongqi7, mayuchi1}@huawei.com

***Abstract*—The rise of Large Language Models (LLMs) has significantly advanced various applications on software engineering tasks, particularly in code generation. Despite the promising performance, LLMs are prone to generate hallucinations, which means LLMs might produce outputs that deviate from users' intent, exhibit internal inconsistencies, or misaligned with the real-world knowledge, making the deployment of LLMs potentially risky in a wide range of applications. Existing work mainly focuses on investigating the hallucination in the domain of Natural Language Generation (NLG), leaving a gap in comprehensively understanding the types, causes, and impacts of hallucinations in the context of code generation. To bridge the gap, we conducted a thematic analysis of the LLM-generated code to summarize and categorize the hallucinations, as well as their causes and impacts. Our study established a comprehensive taxonomy of code hallucinations, encompassing 3 primary categories and 12 specific categories. Furthermore, we systematically analyzed the distribution of hallucinations, exploring variations among different LLMs and benchmarks. Moreover, we perform an in-depth analysis on the causes and impacts of various hallucinations, aiming to provide valuable insights into hallucination mitigation. Finally, to enhance the correctness and reliability of LLM-generated code in a lightweight manner, we explore training-free hallucination mitigation approaches by prompt enhancing techniques. We believe our findings will shed light on future research about code hallucination evaluation and mitigation, ultimately paving the way for building more effective and reliable code LLMs in the future. The replication package is available at https://github.com/Lorien1128/code_hallucination.**

***Index Terms*—code generation, hallucination, large language models**

## I. Introduction

Code generation is the process of automatically generating source code based on provided specifications or requirements, which enables developers to save time by reducing manual coding efforts and allows them to focus on higher-level tasks and problem-solving. Moreover, it can aid in ensuring consistency and reducing the risk of human error during development [59, 67]. Automatic code generation has been a longstanding challenge in both software engineering and artificial intelligence communities. The recent advancements in Large Language Models (LLMs) have significantly propelled this field forward [4, 55, 63]. For example, OpenAI's Codex [4], released in 2021, has achieved a success rate of 28.8% in solving a set of 164 hand-written programming problems. Microsoft's Copilot, a code generation tool powered by Codex, has captured the interest of over 1 million professional developers [15]. Moreover, it has shown the potential to speed up coding tasks by up to 55% [30]. Subsequently, various code LLMs emerged in both academia and industry, such as Incoder [18], StarCoder [39], CodeGen [52, 53], Code Llama [63], DeepSeek-Coder [19], DeepSeek-R1 [20], ChatGPT [55], *etc*. These models have shown impressive performance in many code intelligence tasks, especially in code generation.

Despite the remarkable success of LLMs, their predictive modeling nature makes them susceptible to generate results that are not correct and reliable [64, 88]. They are prone to generate *hallucinations* across various tasks [28]. Hallucinations refers to the phenomenon that LLMs might produce nonsensical or unfaithful outputs to the provided source content [24, 28], which can poses a potential risk in deploying LLMs across various applications. Most existing work mainly focuses on investigating the hallucination for natural language generation (NLG) tasks, for instance, generative question answering [36], abstractive summarization [49], dialogue generation [25], *etc*. However, there is still a lack of clarity regarding the specific types of content that LLMs tend to hallucinate during code generation, as well as their causes and potential consequences they may have. In the integration of the generated code into their development, developers may have the following concerns: What types of hallucinations does a code LLM typically produce? What are the possible causes and impact of different hallucinations? How are these hallucinations distributed across widely-used LLMs and popular benchmarks?

We argue that such hallucinations also occur in the domain of LLM-based code generation, primary manifesting in two main categories: *knowledge hallucination* and *faithfulness hallucination*, similar to the hallucination categories found in NLG domain [24]. Given the differences between natural language generation and code generation—where code possesses unambiguous semantics and must adhere to stricter constraints due to its structured nature and specific requirements—**we**

* Corresponding author: Li Zhang

**define code hallucination as a phenomenon where an explicit semantic conflict between the generated code and established facts due to the failure of the LLM to retain, recall, or process information accurately from users' requirements, contextual code, or real-world knowledge**. Consequently, compared to NLG, there may be variations in specific hallucination categories, their causes and impacts in code generation. The occurrence of code hallucinations may undermine the correctness, performance, maintenance, and even security of the developed software. As a result, the widespread adoption of Code LLMs for code recommendation has the inherent potential to compromise the overall quality and reliability of software. Hence, it is imperative to thoroughly investigate hallucinations in LLM-powered code generation. This will allow us to gain insights into the specific weaknesses that the LLM model may generate, helps us identify code snippets or patterns that are likely to be incorrect or unreliable, providing valuable feedback for improving LLMs. To clarify terminology, a *code* (or a solution program) refers to a complete, functional program that solves the given problem. A *code snippet* refers to a self-contained segment of code that may constitute either a partial or complete solution program. In this paper, it generally denotes a part (or the entirety) of the LLM-generated code.

Driven by the critical gaps in understanding hallucinations in LLM-generated code, we propose three research questions focused on code hallucination. First, while prior studies have explored hallucinations in natural language generation, the unique characteristics of code (*e.g.*, syntax constraints, functional correctness) necessitate a specialized investigation into the categories and distribution of code hallucinations. This motivates **RQ1 (Distribution of Code Hallucinations)**, which systematically categorizes and quantifies hallucinations across diverse LLMs and benchmarks to establish a foundational understanding of their prevalence. Furthermore, through this research question, we aim to reveal whether certain hallucination categories are model-specific or task-dependent, providing a foundation for targeted improvements. Moreover, **RQ2 (Hallucination Cause Analysis)** is motivated by the practical imperative to identify factors contributing to code hallucinations. While prior work attributes NLG hallucinations to issues like training data biases or model overconfidence, code generation introduces unique triggers such as ambiguous/incomplete requirements, or knowledge gaps in programming APIs. Investigating these causes helps developers and researchers prioritize mitigation strategies. Finally, understanding the downstream consequences of code hallucinations is crucial for risk assessment and quality review when using LLMs for code generation. Unlike the focus in NLG hallucination, which concentrates on the impact of text consistency, **RQ3 (Impact of Hallucination)** evaluates the broader impacts of code hallucinations on software quality, including functional correctness, efficiency, and maintainability. These research questions form a holistic framework to characterize code hallucinations, uncover their root causes, assess their implications, and inform future solutions. The interplay between these questions ensures a comprehensive analysis: RQ1 identifies "what" code hallucinations exist, RQ2 explains "why" they occur, and RQ3 evaluates "how" they affect real-world code quality.

To answer these questions, we conducted a thematic analysis [9] of the LLM-generated code to summarize and categorize the hallucinations presented in it based on the unique characteristics of the code, and also analyze the causes and possible impacts of the code hallucinations. Specifically, we collected 3,120 code samples generated by different LLMs on two widely-used code generation benchmarks, and employ the open coding process to analyze these samples. Finally, we establish a comprehensive taxonomy of hallucinations, which comprises 3 primary categories (*Requirement Conflicting* hallucinations, *Code Inconsistency* hallucinations, and *Knowledge* hallucinations) and 12 specific types of hallucinations, and also summarized their causes and impacts. Then we conducted a comprehensive investigation and various statistical analyses of these hallucinations from diverse perspectives to gain a deeper understanding of their distributions, causes, and impacts, aiming to explore the prevailing challenges and opportunities in the field of code generation with LLMs. The analysis reveals that ❶ code LLMs are frequently influenced by a diverse range of hallucinations, with *Requirement Conflicting* being the most prevalent hallucination across all studied LLMs and benchmarks. ❷ Both the model's inherent capabilities and the prompt quality contribute to the hallucination. ❸ Regarding the impact, incorrect functionality is the most common and direct consequence of a majority of hallucinations, while code readability and efficiency are also typically affected. Therefore, it is imperative to develop effective techniques to mitigate hallucinations during code generation.

Based on our findings, we performed a preliminary experiment to explore training-free hallucination mitigation approaches by prompt enhancing techniques. The results suggest that ❹ prompting the LLM to refine requirements or perform chain-of-thought reasoning helps mitigate *Requirement Conflicting* hallucinations, while supplementing prompts with relevant domain knowledge reduces hallucinations stemming from knowledge conflicting hallucinations.

In summary, this paper makes the following contributions:

- **Hallucination taxonomy of LLM-generated code**: We conducted a comprehensive study to analyze the types of content LLMs may tend to hallucinate in code generation, and established a taxonomy of code hallucination categories.
- **Thorough analysis of code hallucinations**: We systematically analyzed the distribution of hallucinations across various LLMs and code generation tasks, and also investigated the causes and impact of code hallucinations.
- **Hallucination mitigation exploration:** We evaluated three widely-used prompt enhancing strategies to assess their effectiveness in reducing LLM hallucinations, complementing this analysis with in-depth qualitative case studies to uncover the underlying mechanisms.
- **Implications for addressing code hallucinations**: We deliberated on the findings and implications associated with evaluating and mitigating hallucinations in LLM-generated code.

## II. Preliminary

### A. Terminology

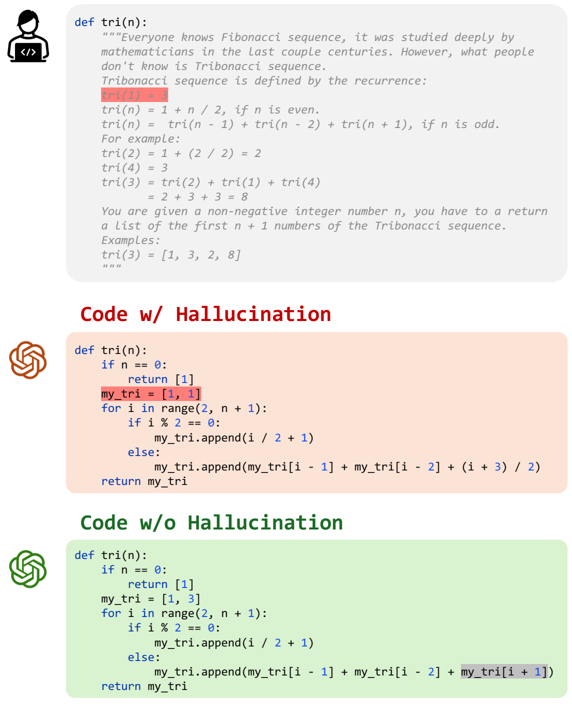
To facilitate our definition of hallucination in the context of code generation, we first clarify several key concepts.

**Definition 1: Requirement.** The requirement refers to the natural language description about the programming task, as provided in the user's input (prompt) to the LLM.

**Definition 2: Contextual Code.** The contextual code encompasses both the code included in the user's input (prompt) and prior code generated by the LLM.

**Definition 3: Real-world Knowledge.** The real-world knowledge refers to publicly available and verifiable information derived from human experience or natural activities.

**Definition 4: Established Fact.** The established fact refers to objective, clear, verifiable, and context-bounded units of information derived from: (1) requirements (*e.g.*, user-specified input/output format constraints), (2) contextual code (*e.g.*, variable names, API usage given in the contextual code), and (3) real-world knowledge (*e.g.*, mathematical laws).



```
def tri(n):
    """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.
    For example:
    tri(2) = 1 + (2 / 2) = 2
    tri(4) = 3
    tri(3) = tri(2) + tri(1) + tri(4)
           = 2 + 3 + 3 = 8
    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.
    Examples:
    tri(3) = [1, 3, 2, 8]
    """
```

```
def tri(n):
    if n == 0:
        return [1]
    my_tri = [1, 1]
    for i in range(2, n + 1):
        if i % 2 == 0:
            my_tri.append(i / 2 + 1)
        else:
            my_tri.append(my_tri[i - 1] + my_tri[i - 2] + (i + 3) / 2)
    return my_tri
```

```
def tri(n):
    if n == 0:
        return [1]
    my_tri = [1, 3]
    for i in range(2, n + 1):
        if i % 2 == 0:
            my_tri.append(i / 2 + 1)
        else:
            my_tri.append(my_tri[i - 1] + my_tri[i - 2] + my_tri[i + 1])
    return my_tri
```

Fig. 1. Illustration of code with or without hallucination.

### B. Definition of Code Hallucination

When generating code, LLMs may involve contextual information from various sources, including user requirements, contextual code, and real-world knowledge. For various reasons, the generated code may not always remain consistent with the contextual information and instead exhibit semantic conflicts. We argue that these semantic conflicts can be divided into two main types: (a) conflicts arising from failure to retain or recall contextual information, and (b) conflicts arising from errors in logical reasoning when converting retained information into actual code. In order to maintain consistency with the core meaning of hallucination in the NLG domain, we regard the first type of conflict as *code hallucination*. Given the black-box nature of LLMs, we typically can only analyze the final output (*i.e.*, the generated code), making it often difficult to precisely attribute a conflict to one type or the other. Therefore, for the feasibility of practical classification, we further restrict code hallucination to "explicit semantic conflict" between generated code and established facts, where the conflict is immediate and obvious, not dependent on complex logical reasoning. In such cases, the probability that the conflict originates from the second type is small, allowing us to reasonably and clearly classify it as the first type.

Based on the above analysis, we formally define code hallucination as **a phenomenon where an explicit semantic conflict between the generated code and established facts due to the failure of the LLM to retain, recall, or process information accurately from users' requirements, contextual code, or real-world knowledge**. This discrepancy could arise due to flawed training data sources [34, 54], suboptimal choices in training and decoding [23, 84], *etc.*, ultimately producing code that exhibits semantic conflicts with established facts. Such misalignment manifests through explicit semantic contradictions, serving as a traceable indicator of hallucination.

Specifically, code hallucination can be categorized into two main types: *Faithfulness Hallucinations* and *Knowledge Hallucinations*, analogous to the faithfulness and factuality hallucinations in NLG [24, 28]. *Knowledge Hallucinations* primarily highlights the discrepancies between the generated code and real-world knowledge, and *Faithfulness Hallucination* involves two situations where the generated code conflicts with either the task requirement (*Requirement Conflicting*) or its contextual code (*Code Inconsistency*), both of which, together with *Knowledge Hallucination*, form the three fundamental types of hallucinations in our taxonomy. Simultaneously, these three types also correspond to the input-conflicting hallucination, context-conflicting hallucination, and fact-conflicting hallucination in NLG [85]. Given the differences between natural language generation and code generation, there may exist variations in specific hallucination categories, causes, and impacts.

Under this definition, we do not consider indirect (implicit) conflicts that require certain logical reasoning or calculations to manifest as hallucinations, as they are inherent to errors in inference or deduction, rather than failures to retain, recall, or process established facts accurately. In contrast, code hallucinations reflect a direct misalignment between the model's output and the information that is provided or expected to know. For example, as illustrated in Figure 1, The requirement of the programming task is to calculate the "Tribinacci" sequence according to the defined recurrence rule. Neither of the two samples in the figure fulfill the the requirement correctly. However, only the first code exhibits hallucination. Specifically, the first code incorrectly initializes `tri(1)` to 1, which directly conflicts with the requirement for `tri(1) = 3` in the task description. Thus, it belongs to a hallucination. In contrast, the second code fails due to incorrectly accesses out-of-range indexes, or alternatively, obfuscating the logical order of the sequence generation, mistakenly assuming that the subsequent item has already been produced. Nonetheless, it is semantically consistent with the requirement, so we do not consider it as an hallucination. In summary, code hallucination focuses on direct semantic conflict without considering naive syntactic errors or logical reasoning/planning errors. The detailed differences between them will be explained in Section II-C.

### C. Difference between Code Hallucinations and Errors

In this section, we highlight the difference between code errors and hallucinations in code generation. Code with hallucinations does not always mean that the code contains errors. For instance, if the code's semantic does not conflict with the functional requirements, such as when it includes useless statements, it is unlikely to result in actual errors in most cases. In addition, semantic conflicts do not necessarily result in conflicts in execution results. On the other hand, not all code errors relate to hallucinations. As mentioned in II-B, code hallucination focuses on direct semantic conflicts. Therefore, implicit conflicts that require logical deduction to manifest are not within the scope of code hallucination. Furthermore, if the error is not even caused by semantic conflicts, such as a simple syntactic error, it will certainly not be classified as a hallucination. To illustrate this more intuitively, below we summarized several typical code error patterns that do not qualify as hallucinations, and we provide an example for each pattern in the online Appendix[1], which is available in our replication package.

- **Syntactic Error:** Errors that are not related to the semantics of the code. These errors can usually be detected during the compilation phase.
- **Logical Reasoning Error:** LLMs might interpret the requirements in a way that leads to incorrect logic. This can result in logical deductions or calculations in the generated code that, while not directly conflicting with the requirements, are still incorrect.
- **Incomplete Implementation:** This error occurs when the generated code partially fulfills requirements without directly conflicting with their semantics. For example, a problem might have multiple specific requirements, but the code addresses only a subset of them.
- **Invalid Generation:** The generated code is structurally invalid due to syntactic incompleteness or lack of coherent semantics. Unlike *Incomplete Implementation* (where code is structurally sound but fails to meet all functional requirements), *Invalid Generation* involves formally incomplete code, such as missing critical syntax elements (*e.g.*, half-formed function definitions or absent control structures).

## III. Related Work

### A. LLMs for Code Generation

In recent years, a significant number of LLMs for code-related tasks, especially for code generation, have been proposed [4, 39, 55, 63]. Codex [4] is the earlier representative work to use large generative pre-trained models with up to 12 billion parameters to generate code. It enabled Copilot to deliver real-time coding suggestions, revolutionizing the coding experience. The success of Codex has captured the interest of both academia and industry groups in this particular field. As a consequence, various models have emerged. DeepMind proposed AlphaCode [40], which is trained for generating code in real-world programming competitions. Meta proposed InCoder [18] and Code Llama [63], Amazon provided CodeWhisperer [2], BigCode project proposed StarCoder [39], DeepSeek-AI introduced DeepSeek-Coder series, OpenAI introduced GPT and ChatGPT series. The emergence of these models yields remarkable enhancements in the effectiveness of code generation. To assess the capabilities of LLMs in code generation tasks, researchers have developed various benchmarks. Notable examples of these benchmarks include HumanEval [4], DS-1000 [33], MBPP [3], APPS [22], CoderEval [80], DevEval [37], *etc*. These benchmarks typically consist of multiple test cases, and are usually accompanied by a few instances to aid in understanding the task description.

### B. Hallucination in LLMs

The term "hallucination" has been widely used within the NLG community to describe the generation of text that is nonsensical or deviates from the original source content [24, 28]. In the field of NLP, hallucinations are primarily categorized into two types: *intrinsic hallucinations*, where the generated content contradicts the source content, and *extrinsic hallucinations*, where the generated content cannot be verified from the source input [28]. Huang et al. [24] further redefined the taxonomy into two main groups: *factuality hallucination* and *faithful hallucination*, aiming to offer a more tailored framework for LLM applications. Factuality hallucination emphasizes the discrepancy between generated content and verifiable real-world facts, typically manifesting as factual inconsistency or fabrication. On the other hand, faithfulness hallucination refers to the divergence of generated content from users' requirements, as well as self-consistency within generated content. Considering the versatility of LLMs, Zhang et al. [85] further refined the definition by categorizing hallucination within the context of LLMs as follows: (1) Input-conflicting hallucination, occurring when the generated content deviates from the original source input; (2) Context-conflicting hallucination, where the generated content contradicts previously generated information; (3) Fact-conflicting hallucination, arising when LLMs produce content that lacks fidelity to established real-world knowledge. In the context of LLM-powered code generation, the hallucination taxonomy proposed by Zhang et al. [85] naturally aligns with the fundamental components of programming tasks: task requirements, contextual information, and real-world knowledge. Building upon this framework, we adapt their taxonomy with code-specific refinements to better characterize and categorize code hallucinations.

### C. Code Hallucination

Recent research has increasingly focused on hallucination phenomena in code generation tasks. Zhang et al. [86] studied

[1]https://github.com/Lorien1128/code_hallucination/blob/master/appendix.pdf

the phenomena, mechanism, and mitigation of LLM hallucinations in repository-level generation scenario, identifying key hallucination categories such as Task Requirement Conflicts, Factual Knowledge Conflicts, and Project Context Conflicts. In a similar vein, Spracklen et al. [66] focused on a very specific type of hallucination: package hallucinations, where the generated code erroneously references non-existent libraries, thereby highlighting how model settings, such as training data recency and decoding strategies, can influence these errors. Other studies have also approached the phenomenon from various angles. For example, Agarwal et al. [1] defined hallucinated code as code that exhibits one or more defects (*e.g.*, syntactic or logical errors, dead code, security vulnerabilities) and proposes a comprehensive taxonomy of these defects. In contrast, Jiang et al. [29] introduced a benchmark that predicts hallucination occurrences by identifying the divergence between LLM outputs and canonical solutions. Rahman and Kundu [58] further distinguished hallucinated output from merely incorrect output by emphasizing that hallucinations often include elements that are completely or partially irrelevant to the provided context. Furthermore, Tian et al. [69] categorized code hallucinations into types such as mapping, naming, resource, and logic hallucinations based on execution-based verification, while Mesbah [50] underscores the importance of contextual analysis to improve the accuracy of AI-driven code generation.

Our research advances previous work in several key aspects. On the one hand, we explicitly distinguish code hallucinations from general code errors by emphasizing direct and traceable semantic conflicts with established facts, and addressing the ambiguity in prior studies that overgeneralize [1] or underspecify [58] the concept. On the other hand, our taxonomy categorizes hallucinations based on their semantic conflict sources (requirements, contextual code, and real-world knowledge), offering finer granularity and more comprehensive scope than repository-level [86] or package-specific [66] classifications. Furthermore, we conduct a systematic investigation into how hallucinations manifest across different LLMs and code generation tasks, analyzing their prevalence, patterns, underlying causes, and practical implications. Building on these insights, we explore and discuss several feasible prompt-enhancing strategies for hallucination mitigation. In summary, our study advances the understanding of hallucination phenomena in code generation by providing a comprehensive analysis that bridges key gaps in current research. Through systematic investigation of previously underexplored dimensions, we offer new insights that both complement and extend the existing body of knowledge in this field.

### D. Evaluation of LLM-Generated Code

With the emergence of code LLMs, many researchers began to focus on the generation capabilities of code LLMs. Driven by this, a wide variety of benchmarks for evaluating LLM-generated code have been proposed, including standalone function benchmarks [3, 4, 5, 6, 22, 26, 32], multi-dependency task benchmarks [33, 89], repository-level task benchmarks [37, 80, 82], competition-level task benchmarks [22, 26], and dynamically updated benchmarks [5, 6, 26], among others. Based on these benchmarks, numerous studies have examined the quality of the LLM-generated code from different aspects, including security, usability, and especially correctness [27, 46, 68, 71, 79]. Jesse et al. [27] explored the extent to which code LLMs are inclined to produce simple, stupid bugs [31]. Nguyen and Nadi [51] assessed the correctness and comprehensibility of GitHub Copilot's code suggestion. Tambon et al. [68] analyzed the bug patterns in LLM-generated code and their prevalence. Liu et al. [45] further conducted an empirical study of ChatGPT-generated code to evaluate its quality and reliability, which also includes an exploration of ChatGPT's self-debugging capability. Similarly, Liu et al. [46] examined the code generated by ChatGPT, with a specific focus on three aspects: correctness, understandability, and security. Dou et al. [13] analyzed the bug types in code generated by LLMs. In contrast to existing research, we conducted the first comprehensive analysis from the perspective of hallucinations to examine the deviations inherent in the LLM-generated code, and also analyzed the code quality issues that can arise from these hallucinations, encompassing most of the quality issues identified in current research [45, 68].

## IV. Hallucinations in LLM-Generated Code

### A. Hallucination Taxonomy Construction

To construct the code hallucination taxonomy, we first collected code samples generated by widely-used LLMs, then adopted the open coding process to analyze these samples, and finally established a taxonomy of code hallucinations.

*1) Data Collection:* We collect code samples generated by four LLMs on two code generation benchmarks.

**Benchmark Selection.** We use two widely-used code generation benchmarks, HumanEval [4] and CoderEval [80], including two popular programming languages and covering common scenarios in current code generation: standalone function and repository-level function generation.

- **HumanEval** consists of 164 standalone hand-written Python coding tasks, covering mathematics, algorithms, reasoning, *etc*. Each task includes a function signature, docstring, ground truth function body as reference, and several unit tests.
- **CoderEval** consists of 230 functions from 43 Python projects and 230 methods from 10 Java projects. These projects are all selected from high star open source projects of various domains, featuring realistic development scenarios. Each task contains the original docstring, the signature, the solution code as reference, and several unit tests to assess the functional correctness of the generated code.

**LLM Selection.** We select four well-known LLMs that have demonstrated great performance in code generation, including CodeLlama [63], DeepSeekCoder [19], GPT-4 [55] and DeepSeek-R1 [20]. This selection encompasses both open-source and closed-source models, as well as general-purpose and code-specialized models. Notably, we include DeepSeek-R1 for its enhanced reasoning capabilities. The diverse characteristics of these models ensure that our evaluation comprehensively represents the current state of advanced LLMs in code generation.

- **CodeLlama** [63] is an open-sourced code LLM based on the LLaMA [70] architecture, specifically designed for code comprehension and generation tasks. We employ CodeLlama-7B[2] to generate code for each problem.
- **GPT-4** [55] is a large-scale, multimodal model developed by OpenAI, exhibiting human-level performance on various tasks, including code generation. We employ GPT-4 preview version (gpt-4-0125-preview version[3]) in our evaluation. The specific number of parameters for this version is not publicly disclosed.
- **DeepSeek-Coder** [19] is a series of code language models trained from scratch, pretrained on 87% of code and 13% of natural language. We adopt 1.3B[4] and 7B[5] versions of DeepSeek-Coder to generate code.
- **DeepSeek-R1** [20] is an open-source reasoning LLM developed by the Chinese startup DeepSeek. It employs advanced reinforcement learning techniques to enhance reasoning capabilities, achieving performance comparable to OpenAI's o1 model across various benchmarks. We used the online API of DeepSeek-R1 (671B)[6] for code generation.

**LLM Decoding Parameter Settings.** To minimize the non-deterministic nature of LLMs and reduce variability of our results, for each LLM, we use greedy decoding to generate one code for each problem of the datasets by setting the `do_sample` parameter to `False`. In this case, the LLM's output is deterministic and remains unaffected by parameters such as `temperature` and `top_p`. Except for the parameters mentioned above, all other parameters of these model use the recommended values in their official documentations.

**Prompt Selection.** To avoid introducing additional variables, the prompts fed into the LLMs consisted only of a brief role description and the default problem description from the dataset, with specific details available in the Appendix. Then we run these LLMs to generate code for each problem from the above two benchmarks. As illustrated in Table I, we finally collected 3,120 (820+1,150+1,150) code samples generated by these LLMs from HumanEval and CoderEval. Each of the problems in two benchmarks has 5 solutions generated by different LLMs. The last column also presents the average pass@1[7] results for each model across the benchmarks.[8]

*2) Manual Analysis:* To analyze the hallucination in LLM-generated code, we conducted a thematic analysis [9]. According to our previous definition of code hallucination, there are three primary code hallucination categories, *i.e.*, *Requirement Conflicting*, *Code Inconsistency* and *Knowledge hallucinations*, where the first two belongs to the faithfulness hallucinations. We initially conducted a pilot analysis by randomly sampling 500 code snippets (around 20% of the total 3,120) to enrich the categories, establish the codebook, and develop an initial taxonomy for code hallucination. After obtaining the codebook, the remaining 80% of the code snippets were labeled by ten annotators to further refine and expand the taxonomy. In the first phase of labeling (the initial 20%), both labelers have over four years of experience in Java and Python programming. In the second phase (the remaining 80%), all the ten labelers possess at least two years of experience in Java or Python programming.

TABLE I
DATA STATISTICS IN MANUAL ANALYSIS.

| Models | HumanEval | CoderEval | | Avg. pass@1 |
|---|---|---|---|---|
| | *Python* | *Java* | *Python* | |
| DeepSeek-Coder-1.3B | 164 | 230 | 230 | 33.33 |
| DeepSeek-Coder-7B | 164 | 230 | 230 | 35.42 |
| CodeLlama-7B | 164 | 230 | 230 | 26.28 |
| GPT-4 | 164 | 230 | 230 | 42.47 |
| DeepSeek-R1 | 164 | 230 | 230 | 48.08 |
| Total | 820 | 1150 | 1150 | - |

We implemented strict quality control procedures throughout the analysis process. In the initial pilot analysis stage, two experts (co-authors of this paper) with rich Python and Java programming experience and code LLM research experience independently evaluated 500 samples: Given the problem description, reference code provided by the original dataset, execution results (correct or error with detailed error message), they independently review each generated code and identify the existence of hallucinations, including both the specific types as well as the position of the hallucinatory code snippet. This process takes approximately 60 person-hours. In order to answer RQ2 & RQ3 and better find ways to deal with code hallucinations, except for the hallucination types, the experts are also asked to identify the possible causes and the impact of the hallucinatory code snippet. **It is worth noting that one code may contain multiple hallucinatory code snippets. A single hallucinatory code snippet can have various causes and impact, and annotators are allowed to annotate multiple causes/effects for each hallucinatory code snippet.** During the labeling process, for the disagreement annotations, the experts and other two of the authors convened for a joint discussion to incorporate more diverse perspectives and enhance annotation reliability until a consensus was reached by an absolute majority. If consensus could not be achieved, the decision was made by the expert with the longest programming experience. Based on the discussion, we grouped similar categories, resolving conflicts and discrepancies. Finally, we organized the annotation results and established the preliminary version of the codebook, illustrating various hallucination types related to LLM-generated code, possible causes and impact of the code hallucinations.

After obtaining the codebook, the remaining 80% of the samples are labeled independently by 10 participants with Python and Java programming experience, and also familiar with code LLMs, with $\sim$240 person-hours. Each code is annotated by two different participants to eliminate the subjective factors of the annotator. Specifically, they label the samples independently following the same procedure as

[2]https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf
[3]https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4
[4]https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-instruct
[5]https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5
[6]https://api-docs.deepseek.com/zh-cn/guides/reasoning_model
[7]We calculate the pass@k metric using the method proposed by Chen et al. [4]. Specifically, for pass@1, we obtain the result by dividing the number of passed samples by the total number of samples and expressing it as a percentage.
[8]For the HumanEval dataset, we employ the upgraded EvalPlus[43] test suite for evaluation.

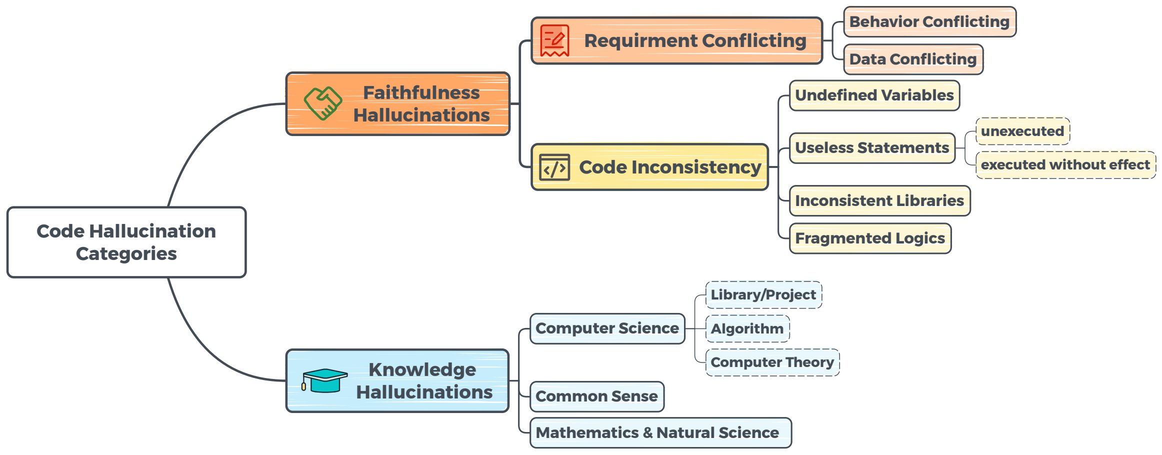


Fig. 2. Taxonomy of code hallucinations.

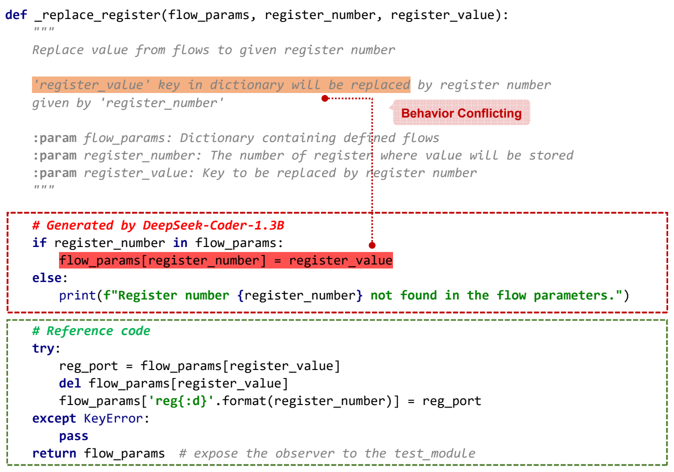
the previous pilot analysis, aiming to refine and enlarge the taxonomy. If a new hallucination category/cause/impact occurs that the codebook does not cover, the annotator needs to write a description of the hallucination, for further discussions to establish new codes and enhance the codebook and taxonomy. We calculated Cohen's Kappa scores pairwise for annotators who independently labeled identical subsets of the dataset across three annotation dimensions: hallucination categories, causes, and impacts. *All Kappa scores ranged from 0.76 to 0.96, exceeding the 0.6 threshold, indicating substantial agreement* [72]. For the disagreements in annotation, the corresponding participants and a senior author (one of experts in the initial stage, with extensive Python/Java programming experience and code LLM research expertise) convened for a joint discussion until a consensus was reached. Similarly, if consensus could not be reached, the final label was determined by the senior author. More labeling details can be found in Appendix.

Finally, we located 1,212 hallucinatory code snippets (173 from HumanEval, 452 from CoderEval-Python, and 587 from CoderEval-Java) from 1,134 samples out of a total of 3,120 code samples, resulting in a code hallucination taxonomy comprising 3 primary categories and 12 specific categories as leaf nodes, as shown in Figure 2. Detailed explanations of each hallucination category will be provided in Section IV-B. Additionally, the manual study also identified the possible causes and impacts of these hallucinations, which will be discussed in Section V-B and Section V-C.

## *B. Taxonomy of Code Hallucinations*

### ***I. Requirement Conflicting (39.60%)***

This is the most prevalent and consequential category, referring to the cases where the semantics of the generated code directly conflicts with the given requirements. It consists of two subcategories: *Behavior Conflicting* (35.40%) and *Data Conflicting* (4.21%). The primary difference between these two categories lies in the specific content of the requirement that is conflicted. Specifically, *Behavior Conflicting* refers to the cases where the code's functional logic or execution flow deviates from the intended behavior outlined in the requirement (*e.g.*, implementing "summation" as "multiplication"). For example, in Figure 3, the problem clearly states that `register_value` is a **key** in the dictionary, but the code generated by DeepSeek-Coder-1.3B incorrectly uses `register_value` as a **value** instead, which directly conflicts with the expected behavior specified in the requirements. This issue may stem from the LLM mistakenly determining the variable's semantic based solely on the meaning of its name. On the other hand, *Data Conflicting* arises when the data entities in the generated code directly conflicts the requirement-specified values or formats. For example, the requirements might specify a fixed return string (*e.g.*, "SUCCESS") under specific conditions, but the generated code instead returns an incorrect value (*e.g.*, "OK").

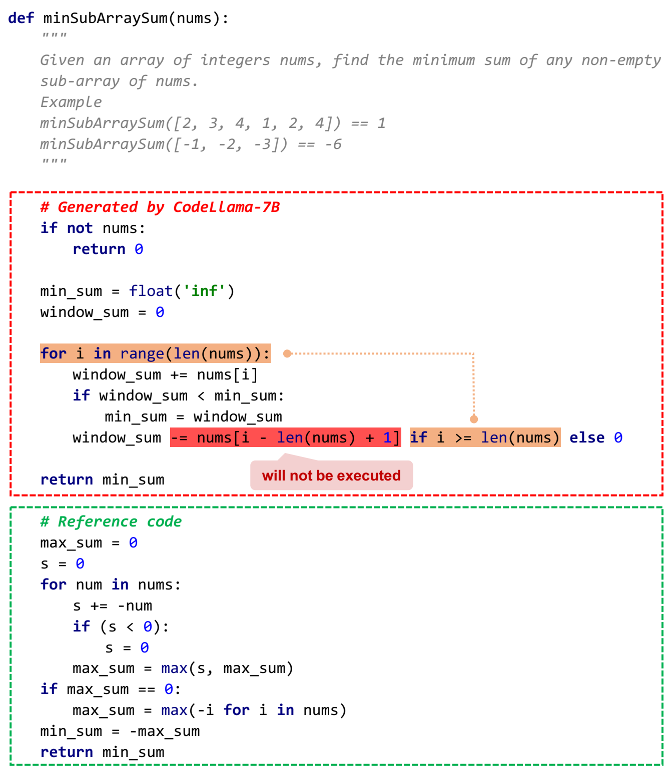


```
def _replace_register(flow_params, register_number, register_value):
    """
    Replace value from flows to given register number

    'register_value' key in dictionary will be replaced by register number
    given by 'register_number'

    :param flow_params: Dictionary containing defined flows
    :param register_number: The number of register where value will be stored
    :param register_value: Key to be replaced by register number
    """

    # Generated by DeepSeek-Coder-1.3B
    if register_number in flow_params:
        flow_params[register_number] = register_value
    else:
        print(f"Register number {register_number} not found in the flow parameters.")

    # Reference code
    try:
        reg_port = flow_params[register_value]
        del flow_params[register_value]
        flow_params['reg{:d}'.format(register_number)] = reg_port
    except KeyError:
        pass
    return flow_params  # expose the observer to the test_module
```

Fig. 3. An example of code with *Behavior Conflicting* hallucination.

### ***II. Code Inconsistency (25.50%)***

This category refers to situations where direct semantic conflicts arise within the generated code. The *code inconsistency* hallucination typically manifests as *Undefined Variables* (16.91%), *Useless Statements* (6.60%), *Inconsistent Libraries* (0.25%), and *Fragmented Logics* (1.73%). *Undefined Variables* occurs when there is a reference to an undefined variable in the generated code. *Useless Statements* typically manifests as two



```python
def minSubArraySum(nums):
    """
    Given an array of integers nums, find the minimum sum of any non-empty
    sub-array of nums.
    Example
    minSubArraySum([2, 3, 4, 1, 2, 4]) == 1
    minSubArraySum([-1, -2, -3]) == -6
    """
```

```python
    # Generated by CodeLlama-7B
    if not nums:
        return 0

    min_sum = float('inf')
    window_sum = 0

    for i in range(len(nums)):
        window_sum += nums[i]
        if window_sum < min_sum:
            min_sum = window_sum
        window_sum -= nums[i - len(nums) + 1] if i >= len(nums) else 0

    return min_sum
```

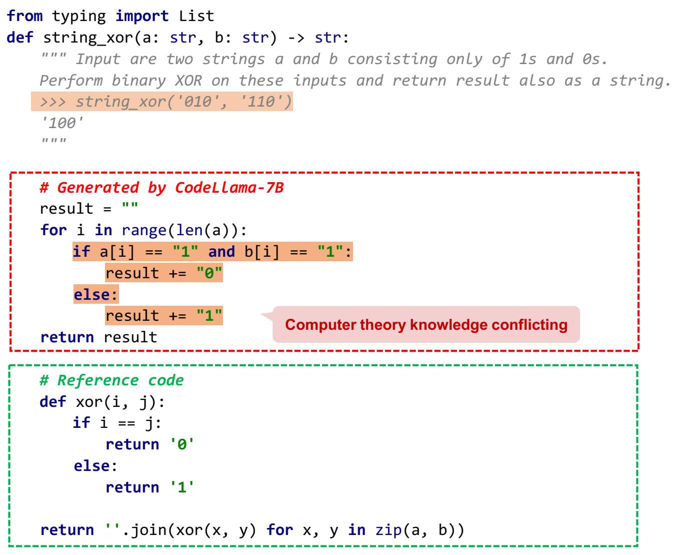
```python
    # Reference code
    max_sum = 0
    s = 0
    for num in nums:
        s += -num
        if (s < 0):
            s = 0
        max_sum = max(s, max_sum)
    if max_sum == 0:
        max_sum = max(-i for i in nums)
    min_sum = -max_sum
    return min_sum
```

Fig. 4. An example of code with *Useless Statements* hallucination.

scenarios: (1) the generated code contains statements that will not be executed (0.58%), or (2) the statements will be executed but has no impact on the output (6.02%). *Inconsistent libraries* refer to the use of libraries with similar but inconsistent functionality or the use of different API versions of the same library in different locations of the code, leading to the failure to achieve the desired objective. For example, `Matplotlib` and `Plotly`, as third-party libraries for Python, both have powerful drawing capabilities. LLM may mistakenly mix the two libraries when generating drawing code, using one library in one part of the code and the other in a different part, resulting in semantic conflicts within the code. *Fragmented Logics* refer to situations where the logical flow of the generated code is disjointed, making it difficult to understand or follow, and increasing the likelihood of errors. Figure 4 shows an example of unexecuted hallucination from the *Useless Statements* category. In the code generated by CodeLlama-7B, the loop condition, `for i in range(len(nums))`, indicates that the maximum value of `i` is `len(num) - 1`, but the condition of the `if` branch in the last line of the loop is `i >= len(nums)`. This conflict prevents the statement `-= nums[i - len(nums) + 1]` from being executed.

### *III. Knowledge Hallucinations (34.90%)*

Unlike the previous two categories, *Knowledge Hallucinations* relate to the direct semantic conflict between the generated code and real-world knowledge. Our manual analysis identified three main types of knowledge conflicts: *Computer Science* knowledge conflicting (33.25%), *Mathematics and Natural Science* knowledge conflicting (1.40%), and *Common Sense* conflicting (0.25%). *Computer Science* knowledge can be further divided into three sub-categories, including



```python
from typing import List
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
```

```python
    # Generated by CodeLlama-7B
    result = ""
    for i in range(len(a)):
        if a[i] == "1" and b[i] == "1":
            result += "0"
        else:
            result += "1"
    return result
```

```python
    # Reference code
    def xor(i, j):
        if i == j:
            return '0'
        else:
            return '1'

    return ''.join(xor(x, y) for x, y in zip(a, b))
```

Fig. 5. An example of code with *Computer Theory* knowledge conflicting hallucination.

*Library/Project* knowledge (25.99%), *Algorithm* knowledge (4.95%), and *Computer Theory* knowledge (2.31%). Figure 5 presents an example of *Computer Theory* knowledge conflicting hallucination, where the model incorrectly assumes that the XOR value of `a` and `b` is 1 only when both `a` and `b` are 1, which conflicts with the binary calculation rule established in computer theory. It is worth noting that, although it could also be seen as a logical reasoning error (*i.e.*, the model correctly recalls the XOR definition but errs in translating it into code), this possibility is obviously less likely. *Library/Project* knowledge primarily refers to the information contained within the standard library, commonly used third-party libraries, or the specific context of the current project. Unlike the *Inconsistent Libraries* category, this type of hallucination focuses on single-point conflicts in library/project knowledge, whereas the former concerns multi-point contextual inconsistencies. More detailed descriptions of each type of hallucination can be found in the codebook provided in the Appendix of our replication package.

## V. Code Hallucination Analysis

Based on previous manual analysis results, we further conducted an in-depth analysis to investigate the following research questions.

### *A. RQ1. Distribution of Code Hallucinations.*

In this RQ, we first analyze the overall distribution of various code hallucination categories and their occurrence across different LLMs. We then delve deeper into the hallucination distribution across different benchmarks.

*1) Overall Distribution of Hallucinations:* Figure 6 and Figure 7 illustrate the overall distribution of code hallucination categories in different LLMs. We can observe that *Behavior Conflicting* is the most frequently occurring type of hallucination across various LLMs, except for DeepSeek-R1. In DeepSeek-R1, *Behavior Conflicting* rank only third among all

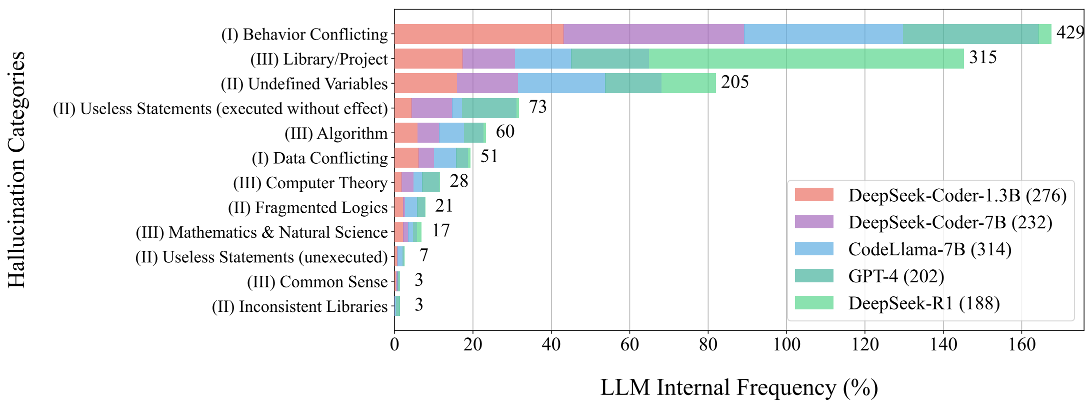


Fig. 6. Overall distribution of code hallucinations. Frequency denotes the percentage of hallucinations in the specified category relative to the total number of hallucinations produced by a single LLM (denoted by the same color). The number behind each bar represents the total number of this hallucination category, and the number behind each LLM legend represents the total number of hallucinations generated by the model.

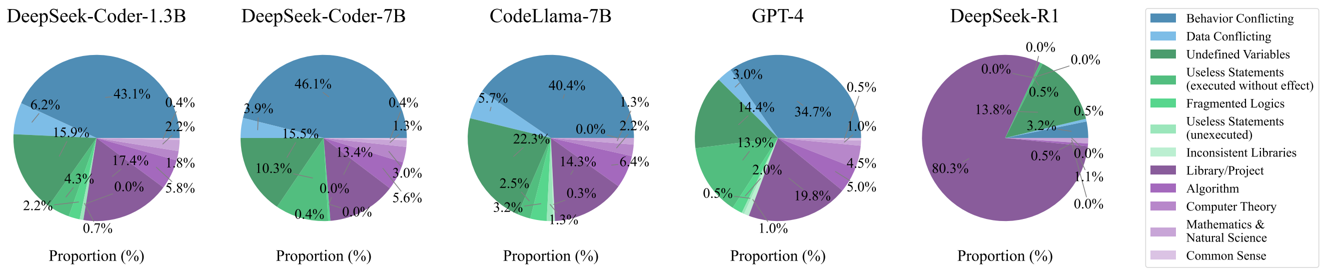


Fig. 7. Distribution of code hallucination in different models.

hallucination categories and are fewer in number than the top two categories. Additionally, *Data Conflicting* is also relatively rare. This suggests that improving reasoning abilities could substantially enhance an LLM's ability to adhere to specified requirements. Following closely behind are *Library/Project* and *Undefined Variables* hallucinations, both of which are largely identified from CoderEval dataset, as the generated function mostly depends on the contextual information within its own class, project, or other libraries. Compared with the *Library/Project* knowledge, the hallucination related to the other two subcategories of *Computer Science* knowledge, *i.e.*, *Algorithm* and *Computer Theory*, are relatively less common. There are also a certain amount of executed *Useless Statements* hallucination, demonstrating that LLMs struggle to fully capture execution semantics. However, these issues are less prevalent in DeepSeek-R1, where such hallucinations constitute only a small fraction. This finding implies that increasing model parameters or enhancing reasoning capabilities can effectively alleviate such hallucinations.

Although *Inconsistent Libraries* occur the least frequently, they particularity warrants additional attention. As previously defined, this category of hallucination refers to the mixed API use of similar libraries or different versions of the same library. These hallucinations are relatively subtle and difficult to detect, especially for beginners. Moreover, such inconsistencies can cause runtime errors or prevent the code from achieving its intended functionality. As a result, *Inconsistent Libraries* may lead to higher debugging costs and greater potential risks compared to other categories of hallucinations.

Regarding the hallucination distribution across LLMs, the current results suggest that the frequency of code hallucinations may be negatively correlated with the model's parameter size. Specifically, the number of hallucinations identified from the code generated by DeepSeek-Coder-1.3B, DeepSeek-Coder-7B, and DeepSeek-R1-671B decreases as the parameter size increases. Compared to DeepSeek-Coder-7B, the code generated by CodeLlama-7B exhibit more hallucination issues, even surpassing those found in DeepSeek-Coder-1.3B. This aligns with their performance in code generation, where the pass rate of CodeLlama-7B is lower than both the 1.3B and 7B versions of DeepSeek-Coder, as shown in Table I. The inferior performance of CodeLlama-7B may be attributed to deficiencies in its training data or model design/training strategies, leading to weaker instruction-following capabilities and more frequent hallucinations in generated code compared to a similarly sized model (DeepSeek-Coder-7B). Additionally, the superior performance of DeepSeek-R1 underscores the importance of more advanced LLM training strategies in mitigating code hallucinations. These observations suggest that code hallucinations are not solely influenced by the parameter size of the LLMs, and the underlying mechanisms—including how various factors such as model architecture, training data quality, and parameter size collectively affect hallucination frequency—require further systematic investigation. Besides, it is interesting to note that, among the five models, DeepSeek-R1, despite exhibiting the fewest hallucinations overall, has the

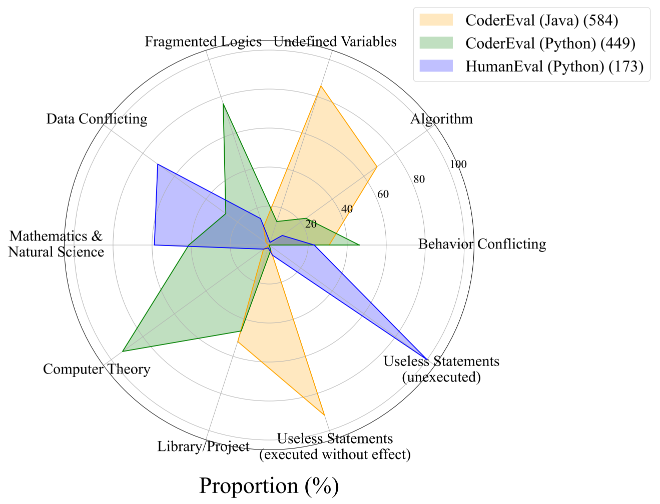


Fig. 8. Distribution of code hallucination on different benchmarks. Proportion indicate the percentage of hallucinations found in each benchmark relative to the total number of hallucinations in that category. The number behind each legend represents the total number of hallucinatory code snippets within the dataset.

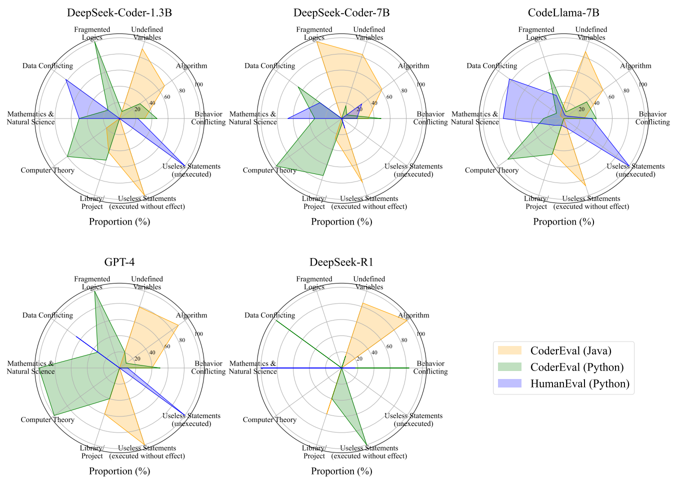


Fig. 9. Distribution of code hallucinations on different benchmarks, divided into separate subplots by model.

highest proportion of hallucinations in *Library/Project*. This observation further underscores the challenges LLMs face in terms of missing real-world knowledge. **Since the number of hallucinations in the last two categories—*Inconsistent Libraries* and *Common Sense*—is quite small, we have omitted them from the following analysis.**

*2) Hallucination Distribution across Benchmarks:* To investigate the hallucination in different as well as real-world code generation scenarios, we employ two representative code generation benchmarks in our manual analysis, covering both standalone function (HumanEval) and repository-level function generation scenarios across Python and Java languages (CoderEval). We present a breakdown distribution of code hallucinations on these two benchmarks in Figure 8. Specifically, for each benchmark, the results shown in the figure are calculated based on the results of the code generated by all LLMs corresponding to that benchmark. We can observe that *Behavior Conflicting* hallucinations commonly occur among all benchmarks. The programming tasks in HumanEval relate to mathematics and algorithms, which can be solved by standalone functions by accessing only built-in functions and standard libraries in Python. Therefore, most *Mathematics & Natural Science knowledge conflicting* hallucinations are found in HumanEval dataset. Besides, solving these tasks requires strong logical reasoning ability, LLMs may produce *Useless Statements* when they make incorrect logical deductions, as illustrated in Figure 4. Additionally, since LLMs lack a real-time compiler or interpreter during generation, they cannot detect such semantic issues beforehand. The programming tasks in CoderEval come from real-world development scenarios, where the solution function could depend on the contextual information within its class, file, project, or the third-party libraries. Therefore, the *Library/Project* hallucinations are mostly found in CoderEval benchmark. Besides, as the prompt only consists of the task requirement, without including the essential contextual information, LLMs tend to generate the *Undefined Variables* or *Useless Statements* as they are unable to determine the correct ones and end up making guesses. This can also lead to *Fragmented Logic* and conflicts with specific *Algorithm* knowledge. There are also differences in the Python and Java projects in CoderEval benchmark, where more mathematics and computer theory related knowledge is required for solving problems in CoderEval-Python. As a result, LLMs might generate hallucinatory code conflicting the above knowledge in CoderEval-Python.

In addition, we plotted a separate subplot for each model, as shown in Figure 9. It can be observed that the distribution of code hallucination types across benchmarks remains largely consistent among different models, indicating that our taxonomy is robust and the type distribution is not significantly affected by model variations. The few noticeable discrepancies, such as those between DeepSeek-R1 and the other four models, primarily arise because of the small number of certain hallucination type, making its proportions more sensitive to minor fluctuations in hallucination counts.

We also apply statistical tests on the above results that exhibit obvious differences, and the results demonstrate that the observed differences are significant. Detailed information can be found in the Appendix included in our online replication package.

**Answer to RQ1:** Code LLMs are frequently influenced by a range of hallucinations, with *Behavior Conflicts* being the most prevalent hallucination across all LLMs and benchmarks. The frequency of hallucinations may be related to various factors of the LLM, including its parameter size—generally speaking, models with larger parameter sizes tend to hallucinate less. The distribution of hallucination types varies across different code generation scenarios and programming languages.

## B. *RQ2. Hallucination Cause Analysis.*

Based on our manual analysis results, we have summarized the causes and impacts of hallucinations as shown in Figure 10. Code generation fundamentally involves two components,

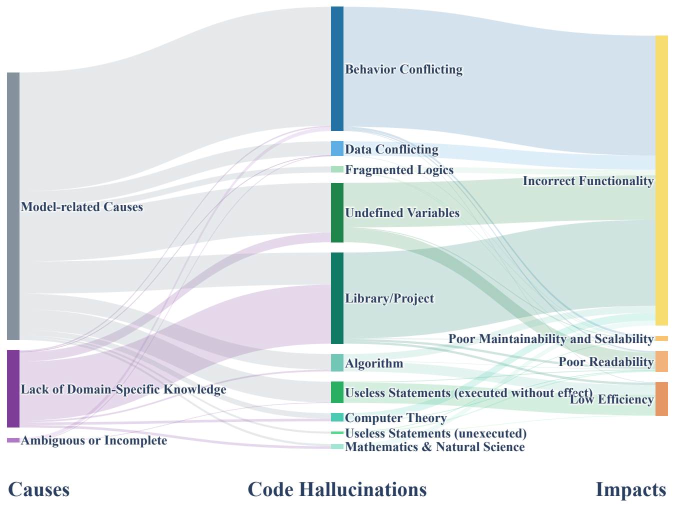


Fig. 10. A visualization of the causes and impacts of various code hallucinations.[1]

*i.e.*, the model and the prompt. Therefore, the causes of hallucinations can stem either from model-related or prompt-related factors. We categorize the causes of hallucinations into three types: (a) ambiguous or incomplete requirements, (b) lack of domain-specific knowledge necessary for code generation, and (c) model-related factors. The first two causes can be classified as deficiencies in the prompt design, that is, how requirements are articulated to the LLM, representing relatively direct attributions. Specifically, these two respectively correspond to defects in two sources of established facts, user requirements and real-world knowledge. As for the third source, contextual code, is itself part of the model's generated output and thus not considered a cause related to the prompt. Beyond these, other characteristics of prompts, such as text complexity and length, may also indirectly influence the occurrence of hallucinations. To address this, we further conducted a unified modeling analysis of these factors. While diverse model-related factors (*e.g.*, architecture, training data, optimization objectives, *etc.*) result in inherent limitations in the LLMs, given the complexity and impracticality of manually identifying these factors, we categorize them broadly as *Model-related Causes*. Generally speaking, code generation fundamentally involves only two components, the model and the prompt. Therefore, the above cause categories are sufficient to cover the vast majority of cases.

*1) Distribution of Hallucination Causes:* The detailed information for each cause and its proportion are as follows. In this RQ, the proportion of each cause category and, in the next RQ, the proportion of each impact category are calculated as the proportion of samples containing hallucinations caused by that cause or resulting in that impact to the total number of samples. More specific examples can be found in the Appendix.

- **Ambiguous or Incomplete (1.90%).** Ambiguity or lack of necessary steps or constraints in the requirement. Ambiguity refers to content that can be interpreted in multiple ways. For example, "merging two integer lists" could be understood as concatenating the lists or as performing element-wise addition. This can lead to incomplete or incorrect code implementation, as the model may guess the intended behavior, increasing the likelihood of generating misaligned or incorrect code.
- **Lack of Domain-Specific Knowledge (26.24%).** Code generation often depends on domain-specific knowledge, such as details about classes or functions from a particular codebase. However, this knowledge may have been introduced after the LLM's training period or could pertain to a highly specialized domain not covered in its training data. In such cases, the model's parameterized knowledge remains incomplete, resulting in gaps when handling domain-specific code generation tasks. Consequently, if the prompt does not provide the necessary domain-specific knowledge for code generation, the LLM may produce code that conflicts with real-world knowledge. It is important to note that the "incomplete" in *Ambiguous or Incomplete* refers to the logical incompleteness of the task description, whereas this type of cause focuses more on the incompleteness of relevant knowledge.
- **Model-related Causes (80.86%).** The limitations of the LLM itself may contribute to the occurrence of hallucinations, and these limitations can stem from various factors, such as poor-quality training data, a small model parameter size, inherent deficiencies in optimization objective or model architecture [24], *etc*. Given the complexity and impracticality of manually identifying these factors, we collectively classify them as *Model-related Causes*. In the labeling process, if the prompt shows no obvious flaws, labelers classify the hallucination under this category.

According to the proportion statistics, *Model-related Causes* accounts for the majority of code hallucinations, significantly more than other factors. This suggests that improving the inherent capability of LLMs could be an effective strategy for substantially reducing these hallucinations and enhancing the overall quality of code generation. Specifically, the capability of LLMs can be enhanced through various approaches, such as improving the quality of training data, refining model architecture, optimizing decoding strategies, *etc*. In addition, *Lack of Domain-Specific Knowledge* is also a significant cause of code hallucinations, particularly in code generation for repo-level scenarios.

Furthermore, we also investigate the contributions of these causes to various categories of hallucinations. As shown in Figure 10, *Model-related Causes* are the primary contributors to almost all categories of hallucinations. In certain hallucination categories (such as *Undefined Variables* and *Library/Project*), factors related to the requirements also play a relatively significant role. This is especially true for *Library/Project*, as many of the requirements in the CoderEval dataset, based on our manual analysis, do not provide sufficient project-specific knowledge. Therefore, it is equally important to write requirements as clearly, accurately, and comprehensively as possible to reduce the occurrence of code

[1]For the convenience of drawing, for hallucinations containing multiple causes or impacts, we randomly retain one of the causes and impacts. The precise data statistics can be found in our replication package.

hallucinations when using LLMs to generate code.

To further validate the reliability of our cause annotations, we conducted additional experiments focusing on two prompt-related causes, *Ambiguous or Incomplete* and *Lack of Domain-Specific Knowledge*. Specifically, we selected the Deepseek-Coder-1.3B model for verification. For *Lack of Domain-Specific Knowledge*, we randomly sampled 144 tasks (with 95% confidence level and 5% margin of error) from the 230 Python tasks in CoderEval. Among these, 9 hallucinatory samples were identified as partially or fully caused by this cause, and 7 of which involved *Library/Project* hallucinations. After enriching the prompts with relevant repository code retrieved via a RAG-based method, only 2 samples remained exhibiting *Library/Project* hallucinations. Similarly, for *Ambiguous or Incomplete*, we manually optimized all 4 corresponding prompts in HumanEval whose generated code contained hallucinations arising from this cause, and after regeneration, only 1 sample remained hallucinatory. The reduction of hallucinations in the newly generated samples confirm that the identified causes are accurate, further supporting the soundness of our analysis and conclusion in this RQ.

*2) Impact of Prompt Length and Complexity:* In addition to the three direct causes of hallucinations mentioned above, code hallucinations may also be indirectly influenced by other characteristics of the prompt. Among these, the complexity and length of the prompt are two fundamental features. Prompts that are too complex or too long may exacerbate the LLM's tendency to forget or confuse established facts, thereby increasing the likelihood of code hallucinations. Our manual analysis (Section IV-A2) also requires labelers to label the text complexity of prompts on a scale of 1-3, with higher values indicating greater complexity [16, 17]. Considering the characteristics of the code generation task specifically, we established four criteria for evaluating complexity: whether the prompt (1) necessitates repeated readings to comprehend its intent, (2) involves multiple reasoning steps, (3) requires specialized domain knowledge or newly introduced definitions, or (4) exhibits syntactical complexity (e.g., nested clauses or prepositional phrases). Labelers then determined the complexity level based on these criteria. The specific criteria and examples can be found in the codebook in the Appendix. Regarding the prompt length, we used the `tiktoken` library to calculate the token length of each prompt.

To investigate the potential relationship between prompt complexity and length, we conducted a joint analysis of these factors' influence on code hallucinations using the scatter plot presented in Figure 11. It can be observed that the probability of hallucinatory code occurrence is indeed influenced by both the complexity and length of the prompt. Furthermore, the distribution of the scatter plot indicates a strong positive correlation between length and complexity. Regarding the impact of prompt length, the probability of hallucinatory code occurrences initially fluctuates and decreases as prompt length increases. This decline may be attributed to the fact that longer prompts allow for more detailed and clarified problem descriptions, which helps LLM better understand the requirement and reduce the likelihood of generating hallucinations. However, once the length reaches approximately 350, the probability

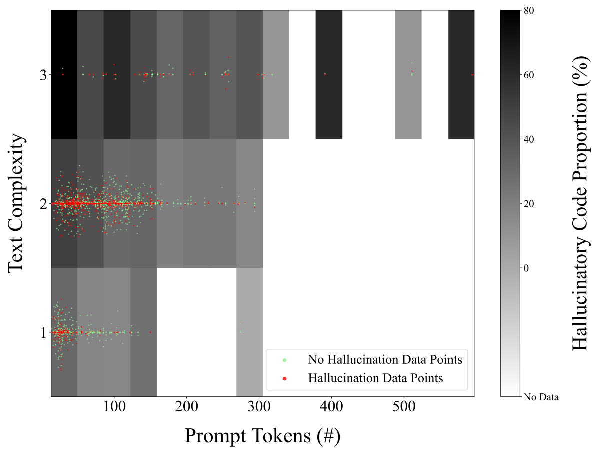


Fig. 11. Heatmap of the proportion of hallucinatory code across different text complexity and length of prompts, along with a scatter plot of the distribution of prompts corresponding to hallucinatory and non-hallucinatory code. The darker the color, the higher the proportion of hallucinatory code, while the white areas represent regions with no data points. To improve visual discrimination of discrete complexity levels (1–3), we slightly dispersed the text complexity values of identical data points. The corresponding figure generated from raw data can be found in the Appendix.

of hallucinations then rises significantly. This may be due to the LLM's difficulty in capturing key points when the prompt becomes too lengthy. In contrast, the impact of prompt complexity shows a clearer trend: higher complexity levels consistently correlate with increased hallucination rates. This suggests that complex logic poses a greater challenge for LLMs in accurately grasping the intent of the requirements. We performed a p-test on the correlations mentioned above between these three variables to support our qualitative analysis, with detailed results provided in the Appendix.

Additionally, when the complexity is fixed, it appears that changes in prompt length do not significantly affect the proportion of hallucinatory code. This suggests that the effect of prompt length on hallucination rates may be mediated by complexity. To validate this hypothesis, we performed a Bootstrap mediation effect significance test on the original data, resulting in a 95% Bootstrap confidence interval of $(0.0113, 0.0725)$, with both bounds greater than 0, indicating that the mediation effect is significant. In other words, the impact of prompt length on hallucinatory code is largely explained by its relationship with complexity.

**Answer to RQ2:** The inherent capabilities of LLMs and deficiencies in prompt design are major contributors to code hallucinations. Other prompt-related factors, particularly the complexity of the prompt, also play a crucial role in causing hallucinations. Enhancing the inherent capabilities of LLMs, such as the reasoning capability, and writing requirements clearly, comprehensively, and concisely could be effective strategies for reducing code hallucinations and improving code quality.

### *C. RQ3. Impact of Hallucination.*

As shown in Figure 10, the impacts of the hallucinations can be summarized into four categories. The detailed information for each impact and its proportion are as follows:

- **Incorrect Functionality (95.30%).** This is the most obvious and direct impact of hallucination. When the generated code conflicts with the requirement, contextual code or real-world knowledge, the functionality of the code is highly likely to be incorrect. This could fundamentally undermines the usability and reliability of the software system, as users encounter unexpected behaviors when interacting with the erroneous implementations. From the debugging perspective, such hallucinations introduce debugging significant challenges, requiring developers to cross-validate the code against requirements, contextual information, or domain-specific knowledge to locate the root cause, increasing troubleshooting complexity.
- **Poor Readability (14.27%).** Certain hallucinations may come with complex code logic or irrelevant code statements, leading to poor readability. This may hinder collaborative development, as team members struggle to understand and modify opaque implementations. Furthermore, this also extends debugging timelines and elevates the risk of introducing new defects during maintenance.
- **Low Efficiency (5.20%).** Some categories of hallucinations may not impact the functionality of the code itself but can introduce redundant logic or unnecessary variables, potentially reducing code efficiency. While remaining the functionality, these inefficiencies may degrade system responsiveness and resource utilization, which probably become critical in latency-sensitive systems. Identifying and resolving such issues demands performance profiling expertise to distinguish hallucination-induced overheads from legitimate computational costs. This additional complexity not only extends debugging efforts but also increases maintenance burdens. It is important to note that compiler optimizations may mitigate such inefficiencies during execution. However, code execution efficiency depends on numerous interacting factors, making it difficult to isolate their effects experimentally. Furthermore, since our research focuses on the semantics of the generated code rather than the compiled machine code, we do not account for compiler optimizations in our analysis.
- **Poor Maintainability and Scalability (3.88%).** Similarly, hallucinatory code may exhibit poor coding style and structure, potentially violating the principles of high cohesion and low coupling or increasing the complexity of function interfaces. This can negatively impact maintainability and scalability. These hallucinations systematically undermines system evolution, making feature extensions error-prone and system upgrades prohibitively expensive. Resolving such issues often requires comprehensive architecture review rather than localized fixes, demanding cross-component impact analysis that dramatically increases maintaining efforts.

According to the proportion statistics and Figure 10, we can observe that 8 out of 10 hallucination categories result in *Incorrect Functionality*. The most frequent hallucination category is *Behavior Conflicting*, where the generated code deviates from the expected behavior in requirements and may produce erroneous outputs, thus making incorrect functionality the most frequent impact. Besides, a large percentage of samples contain *Knowledge* and *Code Inconsistency* hallucinations, like *Library/Project* knowledge conflicting, *Algorithm* knowledge conflicting, and *Undefined Variables*, may also result in incorrect functionality.

TABLE II
2×2 CONTINGENCY TABLE OF CODE ERRORS AND HALLUCINATIONS. THE PERCENTAGES IN PARENTHESES REPRESENT THE PROPORTION OF THE TOTAL SAMPLE.

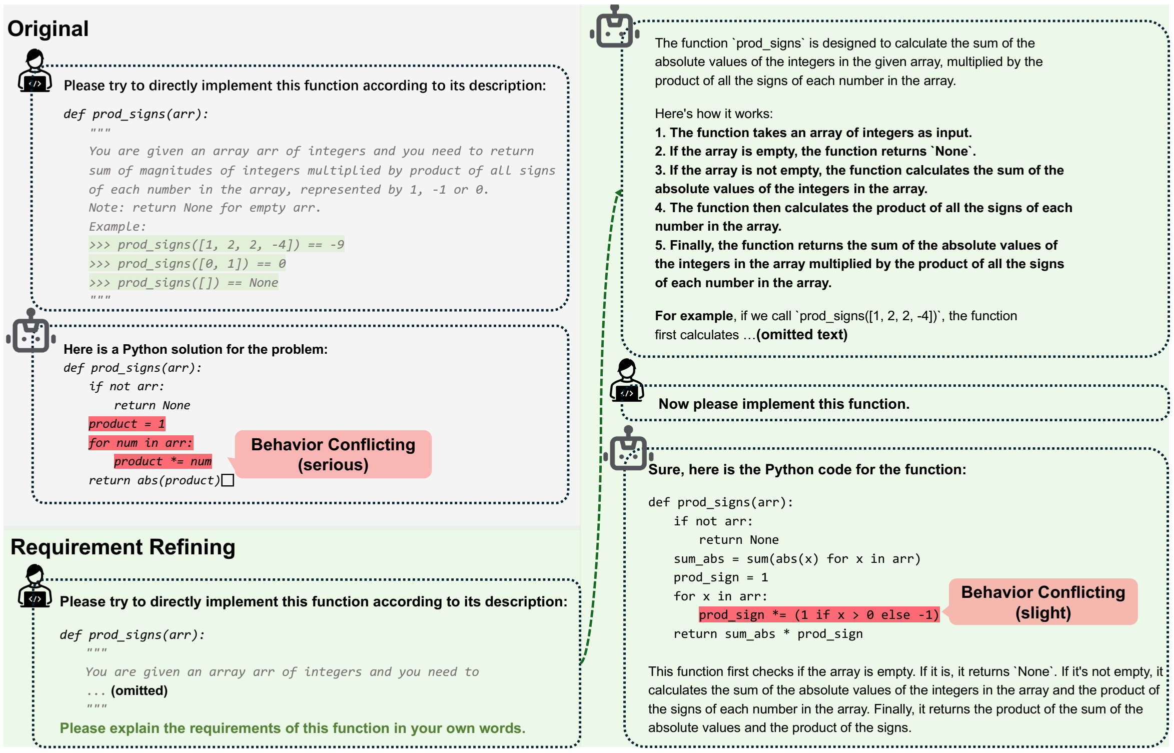
| | **w/ Error** | **w/o Error** | **Total** |
|---|---|---|---|
| **w/ Hallucination** | 1,063 (34.07%) | 71 (2.28%) | 1,134 (36.35%) |
| **w/o Hallucination** | 899 (28.81%) | 1,087 (34.84%) | 1,986 (63.65%) |
| **Total** | 1,962 (62.88%) | 1,158 (37.12%) | 3,120 |

TABLE III
PASS@1 SCORES IN DIFFERENT CATEGORIES OF HALLUCINATIONS.

| **Category** | **Pass@1** |
|---|---|
| **w/o Hallucination** | **54.73** |
| **w/ Hallucination** | **6.26** |
| Computer Theory | 0.00 |
| Common Sense | 0.00 |
| Inconsistent Libraries | 0.00 |
| Mathematics & Natural Science | 0.00 |
| Undefined Variables | 0.49 |
| Data Conflicting | 1.96 |
| Library/Project | 4.76 |
| Fragmented Logics | 4.76 |
| Useless Statements (executed without effect) | 5.48 |
| Behavior Conflicting | 7.46 |
| Algorithm | 33.33 |
| Useless Statements (unexecuted) | 57.14 |

In addition, we also present the distribution of general code errors versus hallucinations across all generated samples in Table II, and further calculate the pass@1 scores for samples with and without hallucination as well as for each individual hallucination category, as shown in Table III. The results show that code containing hallucinations is more prone to errors. Meanwhile, non-hallucinatory code may also contain errors, but these are unrelated to hallucinations. The findings intuitively illustrate the distinctions between hallucinations and general coding errors from a data-driven perspective, and demonstrate that hallucinatory code—across most hallucination categories—exhibits lower functional correctness than hallucination-free code, thereby supporting our earlier analysis. Therefore, mitigating code hallucinations will greatly improve the functional correctness of LLM-generated code.

Following *Incorrect Functionality*, the impacts of *Poor Readability* and *Low Efficiency* are also common, typically arising from hallucination categories other than *Requirement Conflicting*. For example, *Useless Statements* (executed without effect) may not lead to functional incorrectness, but their unnecessary occurrence can reduce efficiency, and these useless statements also could negatively affect the code's readability. Additionally, code with *Algorithm* knowledge conflicting hallucination may use incorrect or suboptimal algorithms, resulting in high time complexity and, consequently, low execution efficiency. Thus, while the impacts of different hallucinations may appear similar, their underlying mechanisms can differ.



Please try to directly implement this function according to its description:

```
def prod_signs(arr):
    """
    You are given an array arr of integers and you need to return
    sum of magnitudes of integers multiplied by product of all signs
    of each number in the array, represented by 1, -1 or 0.
    Note: return None for empty arr.
    Example:
    >>> prod_signs([1, 2, 2, -4]) == -9
    >>> prod_signs([0, 1]) == 0
    >>> prod_signs([]) == None
    """
```

Here is a Python solution for the problem:

```
def prod_signs(arr):
    if not arr:
        return None
    product = 1
    for num in arr:
        product *= num
    return abs(product)
```

Please try to directly implement this function according to its description:

```
def prod_signs(arr):
    """
    You are given an array arr of integers and you need to
    ... (omitted)
    """
```

Please explain the requirements of this function in your own words.

The function `prod_signs` is designed to calculate the sum of the absolute values of the integers in the given array, multiplied by the product of all the signs of each number in the array.

Here's how it works:
**1. The function takes an array of integers as input.**
**2. If the array is empty, the function returns `None`.**
**3. If the array is not empty, the function calculates the sum of the absolute values of the integers in the array.**
**4. The function then calculates the product of all the signs of each number in the array.**
**5. Finally, the function returns the sum of the absolute values of the integers in the array multiplied by the product of all the signs of each number in the array.**

**For example**, if we call `prod_signs([1, 2, 2, -4])`, the function first calculates …**(omitted text)**

Now please implement this function.

Sure, here is the Python code for the function:

```
def prod_signs(arr):
    if not arr:
        return None
    sum_abs = sum(abs(x) for x in arr)
    prod_sign = 1
    for x in arr:
        prod_sign *= (1 if x > 0 else -1)
    return sum_abs * prod_sign
```

This function first checks if the array is empty. If it is, it returns `None`. If it's not empty, it calculates the sum of the absolute values of the integers in the array and the product of the signs of each number in the array. Finally, it returns the product of the sum of the absolute values and the product of the signs.

Fig. 12. An example of using Self-Refine for hallucination mitigation with DeepSeek-Coder-1.3B.

**Answer to RQ3:** Incorrect functionality is the most frequent impact, as most code hallucinations can lead to this consequence, suggesting that mitigating code hallucinations will greatly improve the functional correctness of LLM-generated code. Moreover, code hallucinations can negatively affect code readability and execution efficiency, which are also important factors to address for improving overall code quality.

## VI. Discussion

### A. Hallucination Mitigation Exploration

Previous analysis have demonstrated that code generated by LLMs frequently suffers from various hallucination issues. Addressing these issues is crucial for ensuring the quality and reliability of the generated solutions. Building upon our established taxonomy of hallucinations, along with their corresponding causes and impacts, we argue that it would be effective to design tailored methods to address specific types of hallucinations. Unlike traditional code generation approaches, LLMs have the potential to learn from the prompt and solve problems without resorting to model parameter updates, such as Self-Repair [21, 83], Self-Refine [12, 48], Retrieval-Augmented Generation (RAG) [35], Chain-of-Thought (CoT) [11, 74] prompting techniques, *etc*. Compared to the training-based approaches, these methods offer a more flexible and adaptive way for addressing the issues. Thus, in this section, we explore and discuss several feasible prompting enhancing strategies for hallucination mitigation in code generation. For example, *Requirement Conflicting* hallucination, which may arise from misunderstanding requirements or poor reasoning during instruction following, can potentially be mitigated by prompting LLMs to refine requirements or apply chain-of-thought reasoning. For *Code Inconsistency* hallucination, chain-of-thought prompting techniques can also be used to help the LLM create more reasonable plans for implementations. Additionally, post-processing methods can be applied to eliminate common *code inconsistency* conflicts. For example, static code analysis tools can be used to detect *Undefined Variables* and replace them with the most similar defined variable. As for the *Knowledge hallucination*, relevant real-world knowledge can be retrieved as context and provided to the LLM with Retrieval Augmentation Generation (RAG) techniques, helping to compensate for any gaps in the model's real-world knowledge.

To demonstrate the feasibility of the above ideas and provide references for future research, we experimented with several feasible prompting enhancing strategies for hallucination mitigation, including **Self-Refine**, **CoT** and **RAG**. We chose DeepSeek-Coder-1.3B as the LLM for our experiments, with generation parameters consistent with those in Section IV-A1. For Self-Refine and CoT, as previously discussed, these strategies have stronger potential to mitigate *Requirement Conflicting* and *Code Inconsistency* hallucinations. Therefore, for these two strategies, we selected the HumanEval dataset for evaluation due to its high prevalence of these specific hallucination types. The prompt formats are the same as in Figure 12 and the first hallucination mitigation example in the Appendix respectively. Similarly, for RAG-based strategy, we employed CoderEval (Python) dataset for evaluation. We used a sliding window approach (window size: 20 lines, step size: 2 lines) with the OpenAI `p50k_base` tokenizer to retrieve semantically similar code snippets from the repository, and

TABLE IV
PASS@1 SCORES AND MANUAL ANALYSIS RESULTS FOR THE THREE PROMPTING ENHANCING STRATEGIES USED TO MITIGATE HALLUCINATION.

| Dataset | Strategy | Pass@1 | Samples | Hallucinatory Samples | | *Requirement Conflicting* | | *Code Inconsistency* | | *Knowledge Hallucinations* | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *quantity* | *proportion* | *quantity* | *proportion* | *quantity* | *proportion* | *quantity* | *proportion* |
| **HumanEval** | Origin | 61.59 | 115 | 30 | 26.09% | 28 | 84.85% | 2 | 6.06% | 3 | 9.09% |
| | Self-Refine | 54.27 7.32↓ | | 16 14↓ | 13.91% 12.18%↓ | **9** | **56.25%** | 3 | 18.75% | 4 | 25.0% |
| | CoT | 51.83 9.76↓ | | 18 12↓ | 15.65% 10.44%↓ | **15** | **83.33%** | 3 | 16.67% | 0 | 0.0% |
| **CoderEval (Python)** | Origin | 14.78 | 144 | 62 | 43.06% | 38 | 57.58% | 7 | 10.61% | 21 | 31.82% |
| | RAG | 26.52 11.74↑ | | 29 33↓ | 20.14% 22.92%↓ | **17** | **58.62%** | **3** | **10.34%** | **9** | **31.03%** |

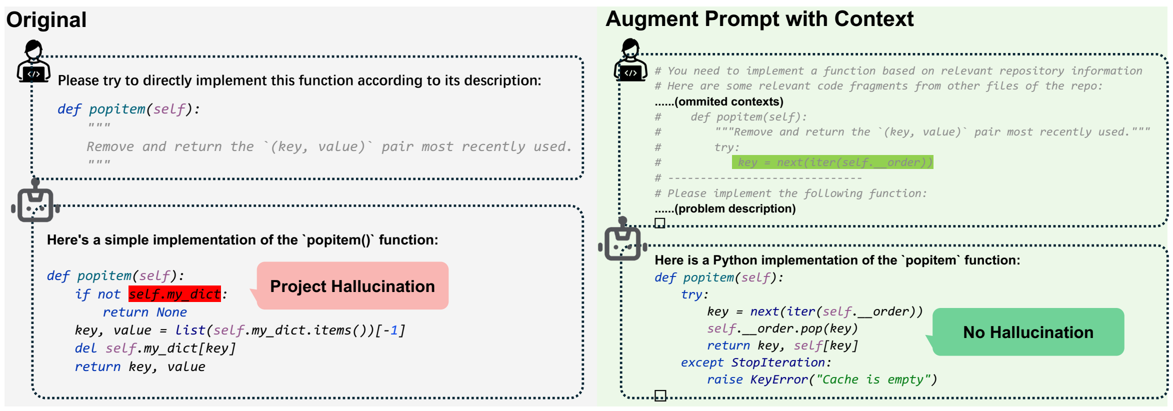
then constructed prompts with a maximum token length of 1,000 in the same format as shown in Figure 13.

We first execute the code generated under these three strategies and obtained their pass@1 results. Then, we randomly sampled a subset for manual evaluation with a 95% confidence level and a 5% confidence interval. Our quality control procedures followed the same rigorous standards established in Section IV-A2, and the results are shown in Table IV. In the table, "Origin" refers to baseline code generation using the original prompt template from Section IV-A1. For the "Hallucinatory Samples" columns, "quantity" and "proportion" refer to the number of hallucinatory samples and their percentage relative to total samples, respectively. And for the last six columns, "quantity" and "proportion" refer to the number of hallucinations of each type and the proportion of that type of hallucination among all hallucination types. The results show that these three strategies substantially reduce hallucinations in their respective target datasets, with particularly effective mitigation of *Requirement Conflicting*. For RAG, we observe a notable reduction in *Knowledge Hallucinations* on the CoderEval (Python) dataset, which aligns with our expectations. However, Self-Refine and CoT strategies exhibited decreased pass@1 performance, which is primarily attributed to the limitation to smaller LLMs' (1.3B) constrained reasoning capacity, where multi-step processing amplifies initial errors through error propagation [65]. This further explains why CoT did not have a significant impact on the number of *Code Inconsistency* hallucinations. We conducted a further investigation into the tasks that led to a decrease in pass@1. For Self-Refine, among the 17 tasks that originally passed but failed after prompt enhancement, all 17 corresponding samples under the original prompt did not contain hallucinations. For CoT, the numbers are 22 and 21, respectively. In summary, the reduction in pass@1 caused by prompt enhancement mainly originates from samples that were originally free of hallucinations.

To further understand the mechanisms behind the effectiveness of these strategies, we provide several specific examples for qualitative analysis. Figure 12 presents an example of employing Self-Refine for mitigating *Behavior Conflicting*. In this example, the task is to calculate the sum of the absolute values of each number in the given array, and then multiplying it by the product of their signs. The original generated code contains a serious *Behavior Conflicting*, which directly calculates the absolute value of the product of each number, rather than the product of their signs. To mitigate this issue, we prompt the LLM to explain the requirement, enabling it to refine the requirement on its own before generating the code. As seen from the figure, after the refinement, although the code generated still exhibits the *Behavior Conflicting* (only considering 1 or -1 as possible sign values, which conflicts with the expected code behavior specified in the problem description, where the values should be 1, -1, or 0.). However, compared to the original code, the hallucination is notably milder. Additionally, when the input list does not contain a 0, the new generated code is functionally correct.

Figure 13 illustrates an example of using RAG to augment the prompt for mitigating *knowledge hallucination*. Originally, LLM struggles to understand the requirements precisely without the relevant project knowledge. For instance, they might incorrectly use a attribute/method that does not exist in the class (*i.e.*, using `self.my_dict` instead of `self.__order`), resulting in the appearance of *knowledge hallucination*. Actually, the project-specific context can provide essential semantic information for understanding the functionality as well as offering valuable insights into the design and implementation of the target code. It is worth noting that the functionality of `popitem` method in another class in the context happens to be complementary to the target function, with much of the underlying logic and implementation being largely similar. Therefore, by referring to the context, the LLM can generate code that closely aligns with the project's requirements. More examples can be found in the Appendix included in our online replication package.

Based on the above analysis, we believe that the prompt enhancing techniques present a promising direction for hallucination mitigation. Our hallucination taxonomy serves as a useful guidance for developers in selecting optimization methods based on their specific needs and preferences. For instance, if a developer prioritizes time complexity, and as shown in Figure 10, the low efficiency is largely due to the LLM's lack of algorithmic knowledge, the developer can focus on retrieving relevant algorithms and incorporating them as context when interacting with LLMs. However, the effectiveness of the prompt enhancing techniques still heavily relies on several factors, including the LLM's inherent ability to understand and following requirements, its logical reasoning skills, and the accurate extraction of relevant real-world knowledge. Therefore, improving the LLM's inherent abilities through advanced training techniques (*e.g.* reinforcement learning [56]), improved decoding strategies [10, 38], model architecture optimization [42, 60], and multi-agent interaction [14], as well as leveraging external tools for knowledge retrieval and fact-checking [75, 78], are also worthwhile areas of study for future research.



Please try to directly implement this function according to its description:

```
def popitem(self):
    """
    Remove and return the `(key, value)` pair most recently used.
    """
```

Here's a simple implementation of the `popitem()` function:

```
def popitem(self):
    if not self.my_dict:
        return None
    key, value = list(self.my_dict.items())[-1]
    del self.my_dict[key]
    return key, value
```

```
# You need to implement a function based on relevant repository information
# Here are some relevant code fragments from other files of the repo:
......(ommited contexts)
#     def popitem(self):
#         """Remove and return the `(key, value)` pair most recently used."""
#         try:
#             key = next(iter(self.__order))
# ------------------------------
# Please implement the following function:
......(problem description)
```

Here is a Python implementation of the `popitem` function:

```
def popitem(self):
    try:
        key = next(iter(self.__order))
        self.__order.pop(key)
        return key, self[key]
    except StopIteration:
        raise KeyError("Cache is empty")
```

Fig. 13. An example of using RAG for hallucination mitigation with DeepSeek-Coder-1.3B.

## B. Implications

In this section, we discuss the implications of our findings for researchers and developers in software engineering and LLM fields.

*1) Implications for Researchers:* In our study, we have observed that code LLMs also experience and are frequently influenced by hallucinations. While most evaluation metrics and benchmarks in existing code generation research prioritize the functional correctness of the code, incorporating measures and benchmarks to detect hallucinations is crucial for providing a more comprehensive and nuanced assessment of the quality and reliability of the LLM-generated code. Considering the enormous workload and potential subjectivity associated with manual hallucination detection, it is necessary to develop automated methods for detection. One promising direction is to leverage advanced LLMs directly for detection. In this approach, the hallucination taxonomy, representative examples for each category, the generated code, and its contextual information can be provided to the LLM, which then determines whether a hallucination exists and identifies its type. To improve accuracy and reliability, more sophisticated mechanisms can be introduced, such as ensemble voting across multiple LLMs, a coarse-to-fine classification strategy, multi-agent debates to determine outcomes, or incorporating a critic model to evaluate the reasonableness of the predicted type. Additionally, based on our manually annotated dataset, the dataset can be further expanded and used to fine-tune an LLM, allowing it to learn code hallucination detection in a supervised manner.

Regarding hallucination mitigation in code generation, according to our findings in RQ2, most of the hallucinations are caused by model-related causes. Coupled with our hallucination mitigation exploration study in previous subsection, it is imperative to improve the reasoning capabilities of LLMs by advanced training/fine-tuning techniques. For example, employing reinforcement learning algorithms during the training can teach the model to think productively using its chain-of-thought, thereby improving its chain-of-thought reasoning ability during inference [56, 61, 73, 76]. Additionally, integrating RAG techniques and domain-specific knowledge [7, 44, 82, 87] warrants further investigation, as they hold promise for boosting the LLM's performance by leveraging external information to support and refine its reasoning process.

Furthermore, in this paper, we primary focus on studying hallucinations in the NL2Code generation task. However, hallucination occurrences and distribution may vary across different code-related tasks, such as code translation [62], unit test generation [81], program repair [77], code review [41, 47], *etc*. This presents an opportunity for further research to explore the characteristics and patterns of hallucinations in these tasks. By gaining a deeper understanding of the underlying mechanisms, researchers can develop task-specific strategies and techniques to mitigate hallucinations, improving the accuracy and reliability of code LLMs for specific code-related tasks.

*2) Implications for Developers:* The emergence of the LLM-as-a-service community has become a prominent trend, allowing developers to easily deploy and utilize various LLMs in their daily development tasks. To reduce the hallucinations, according to our findings in RQ1, the model with larger parameter size tend to hallucinate less, thus selecting an LLM with robust performance capabilities is crucial. Additionally, the quality of prompts plays a significant role. Developers should craft prompts that are clear, concise, and complete, ensuring they provide all necessary context to guide the model effectively. Besides, applying detailed constraints or rules also help guide the LLM to generate code stays within certain boundaries and reduces the likelihood of hallucinations.

Finally, rigorous result-checking and iterative prompt refining processes are also essential for validating and improving the generated results. Given that hallucinations are not always easy to identify and can lead to quality issues, result checking [8, 57] becomes even more critical. Besides, the developers should also carefully review and thoroughly test the generated code to ensure its logical correctness and consistency with the given requirements. If any issues are detected during this process, refining the prompts based on the feedback obtained from testing and review can be beneficial in identifying and mitigating hallucinations.

## C. Threats to Validity

**Threats to external validity** relate to the generalizability of our code hallucination taxonomy and findings. Our empirical study specifically targets Python and Java programming tasks with varying difficulty and domains from two widely-used code generation benchmarks, covering both standalone function and repository-level function generation scenarios. The LLMs evaluated in this study are advanced, representative, and widely used in software engineering research. They encompass both open-source and closed-source models, as well as general-purpose and code-specialized models, ensuring our conclusions generalize to other LLMs. During manual analysis, we annotated LLM-generated code sequentially by model. Notably, the final three (of four) LLMs introduced no new hallucination types, further validating both the completeness of our taxonomy and its applicability across diverse models. Additionally, the consistent distribution of hallucination proportions observed across all evaluated LLMs underscores the robustness of our findings. Nonetheless, it would be interesting to explore the hallucinations in other programming languages and code generation scenarios as well.

**Threats to internal validity** relate to the subjectivity of our manual analysis. The labeling of a code snippet's hallucination categories, causes, and impact is somewhat subjective, and different annotators may have varying determinations of the same code snippets. To address this, we initially created a codebook with precise guidelines for each hallucination categorization, cause, and impact based on an pilot analysis. Additionally, we organized discussions and meetings to resolve any conflicts or uncertainties that arose. This comprehensive process significantly enhances the reliability and quality of the taxonomy of hallucination categories, causes, and impact in LLM-generated code resulting from our manual analysis. However, despite these efforts, providing rigorous categorization criteria to completely eliminate subjectivity remains unrealistic. Whether in natural or programming languages, semantics are inherently continuous and lack clear-cut discrete divisions. Attempting to impose artificial boundaries would require numerous ad hoc exceptions, which is impractical and could undermine the generality and robustness of the taxonomy. A promising future direction is to leverage LLMs as annotators to automatically assign categories to given samples. Even when a taxonomy lacks strictly quantitative criteria, such an approach may yield more objective, consistent, and reproducible annotations. Nevertheless, due to the limited interpretability of LLMs themselves, the reliability and validity of this method remain important questions for further exploration.

**Threats to construct validity** relate to the variability in LLM-generated code in our manual analysis due to decoding strategies and different prompts. To address this issue, regarding the decoding strategy, we use greedy decoding to generate one code for each problem of the datasets, ensuring LLMs produce consistent outputs for the same input, thereby reducing variability. As for the prompt design, to ensure a fair evaluation across all models, we employ a consistent prompt structure for all LLMs and tasks within the benchmarks. This structure includes two core components: (1) a role description (*e.g.*, "I want you to play the role of a programmer, and please ...") to contextualize the LLM's objective, and (2) the programming task description provided in the dataset, which outlines specific programming requirements. According to our findings in Section VI-A, prompt engineering can have significant impact on the quality of the generated code, which is a promising research direction for code hallucination mitigation.

## VII. Conclusion

In this paper, we present an empirical study on code generation hallucinations produced by LLMs. Through a comprehensive manual analysis, we developed a taxonomy of hallucinations through open coding and iterative refinements. Further investigation reveals that code LLMs are frequently affected by hallucinations, particularly those that conflict with users' requirements. The code hallucinations are influenced by both the model's inherent capabilities and the quality of the prompts, leading to incorrect functionality, poor readability, poor maintainability, and low efficiency. We also explore training-free approaches for mitigating hallucinations through prompt enhancement techniques. Finally, we discuss the implications of our study and propose future research opportunities to improve the reliability and trustworthiness of LLMs in code generation. To foster further research, we have made the replication package publicly available at https://github.com/Lorien1128/code_hallucination.

## Acknowledgement

This work is supported by the National Natural Science Foundation of China Grants Nos. 62302021, 62577007, 62332001, 62502283, and the State Key Laboratory of Complex & Critical Software Environment (Grant Nos. CCSE-2025ZX-09 and CCSE-2024ZX-14).

## References

[1] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333*, 2024.

[2] Amazon. Amazon codewhisperer. https://aws.amazon.com/codewhisperer/, 2023.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

[5] Simin Chen, Pranav Pusarla, and Baishakhi Ray. Dycodeeval: Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. In *Forty-second International Conference on Machine Learning*.

[6] Simin Chen, Xiaoning Feng, Xiaohong Han, Cong Liu, and Wei Yang. Ppm: Automated generation of diverse

programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering*, 1(FSE):1194–1215, 2024.

[7] Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *ACL (1)*, pages 7957–7977. Association for Computational Linguistics, 2024.

[8] I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios. *arXiv preprint arXiv:2307.13528*, 2023.

[9] Daniela S. Cruzes and Tore Dybå. Recommended steps for thematic synthesis in software engineering. In *ESEM*, pages 275–284. IEEE Computer Society, 2011.

[10] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023.

[11] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. In *ACL (Findings)*, pages 3563–3578. Association for Computational Linguistics, 2024.

[12] Yangruibo Ding, Marcus J. Min, Gail E. Kaiser, and Baishakhi Ray. CYCLE: learning to self-refine the code generation. *Proc. ACM Program. Lang.*, 8(OOPSLA1): 392–418, 2024.

[13] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*, 2024.

[14] Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*, 2023.

[15] Euronews. Microsoft attracting users to its code-writing, generative ai software. https://www.euronews.com/next/2023/01/25/microsoft-results-ai, 2023.

[16] Jill Fitzgerald, Jeff Elmore, Heather Koons, Elfrieda H Hiebert, Kimberly Bowen, Eleanor E Sanford-Moore, and A Jackson Stenner. Important text characteristics for early-grades text complexity. *Journal of Educational Psychology*, 107(1):4, 2015.

[17] Jill Fitzgerald, Jeff Elmore, Elfrieda H Hiebert, Heather H Koons, Kimberly Bowen, Eleanor E Sanford-Moore, and A Jackson Stenner. Examining text complexity in the early grades. *Phi Delta Kappan*, 97(8):60–65, 2016.

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *ICLR*. OpenReview.net, 2023.

[19] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[20] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[21] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695, 2020.

[22] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[23] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *ICLR*. OpenReview.net, 2020.

[24] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232*, 2023.

[25] Minlie Huang, Xiaoyan Zhu, and Jianfeng Gao. Challenges in building intelligent open-domain dialog systems. *ACM Transactions on Information Systems (TOIS)*, 38(3):1–32, 2020.

[26] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[27] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. *arXiv preprint arXiv:2303.11455*, 2023.

[28] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12): 1–38, 2023.

[29] Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. Collu-bench: A benchmark for predicting language model hallucinations in code. *arXiv preprint arXiv:2410.09997*, 2024.

[30] Eirini Kalliamvakou. Research: quantifying github copilot's impact on developer productivity and happiness. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/, 2022.

[31] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577,

2020.

[32] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.

[33] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[34] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[36] Chenliang Li, Bin Bi, Ming Yan, Wei Wang, and Songfang Huang. Addressing semantic drift in generative question answering with auxiliary extraction. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 942–947, 2021.

[37] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *ACL (Findings)*, pages 3603–3614. Association for Computational Linguistics, 2024.

[38] Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36: 41451–41530, 2023.

[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[41] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *ESEC/SIGSOFT FSE*, pages 1035–1047. ACM, 2022.

[42] Zuchao Li, Shitou Zhang, Hai Zhao, Yifei Yang, and Dongjie Yang. Batgpt: A bidirectional autoregessive talker from generative pre-trained transformer. *arXiv preprint arXiv:2307.00360*, 2023.

[43] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[44] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *CoRR*, abs/2406.07003, 2024.

[45] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *arXiv preprint arXiv:2307.12596*, 2023.

[46] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *arXiv preprint arXiv:2308.04838*, 2023.

[47] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *ISSRE*, pages 647–658. IEEE, 2023.

[48] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[49] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. On faithfulness and factuality in abstractive summarization. *arXiv preprint arXiv:2005.00661*, 2020.

[50] Ali Mesbah. Reducing hallucinations: Harnessing contextual analysis for ai code generation.

[51] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[52] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*. OpenReview.net, 2023.

[54] Yasumasa Onoe, Michael JQ Zhang, Eunsol Choi, and Greg Durrett. Entity cloze by date: What lms know about unseen entities. *arXiv preprint arXiv:2205.02832*, 2022.

[55] OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt, 2022.

[56] OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, 2024.

[57] Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

[58] Mirza Masfiqur Rahman and Ashish Kundu. Code hallucination. *arXiv preprint arXiv:2407.04831*, 2024.

[59] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 111–118, 2024.

[60] Clément Rebuffel, Marco Roberti, Laure Soulier, Geoffrey Scoutheeten, Rossella Cancelliere, and Patrick Gallinari. Controlling hallucinations at word level in data-to-text generation. *Data Mining and Knowledge Discovery*, pages 1–37, 2022.

[61] Paul Roit, Johan Ferret, Lior Shani, Roee Aharoni, Geoffrey Cideron, Robert Dadashi, Matthieu Geist, Sertan Girgin, Léonard Hussenot, Orgad Keller, et al. Factually consistent summarization via reinforcement learning with textual entailment feedback. *arXiv preprint arXiv:2306.00186*, 2023.

[62] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611, 2020.

[63] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[64] Tushar Sharma. Llms for code: The potential, prospects, and problems. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pages 373–374. IEEE, 2024.

[65] Jihoo Shim, Shin Dong Ho, and Jeongwon Kim. Cot harms performance of rather smaller language models. 2024.

[66] Joseph Spracklen, Raveen Wijewickrama, AHM Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. We have a package for you! a comprehensive analysis of package hallucinations by code generating llms. *arXiv preprint arXiv:2406.10279*, 2024.

[67] Benyamin Tabarsi, Heidi Reichert, Ally Limke, Sandeep Kuttal, and Tiffany Barnes. Llms' reshaping of people, processes, products, and society in software development: A comprehensive exploration with early adopters. *arXiv preprint arXiv:2503.05012*, 2025.

[68] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. Bugs in large language models generated code. *arXiv preprint arXiv:2403.08937*, 2024.

[69] Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. Codehalu: Code hallucinations in llms driven by execution-based verification. *arXiv e-prints*, pages arXiv–2405, 2024.

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[71] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022.

[72] Anthony J Viera, Joanne M Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.

[73] Yanlin Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. *CoRR*, abs/2407.19487, 2024.

[74] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[75] Shi Weijia, Min Sewon, Yasunaga Michihiro, Seo Minjoon, James Rich, Lewis Mike, and Yih Wen-tau. Replug: Retrieval-augmented black-box language models. *arXiv preprint ArXiv:2301.12652*, 2023.

[76] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. *Advances in Neural Information Processing Systems*, 36, 2024.

[77] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.

[78] Jian Xie, Kai Zhang, Jiangjie Chen, Renze Lou, and Yu Su. Adaptive chameleon or stubborn sloth: Revealing the behavior of large language models in knowledge conflicts. In *The Twelfth International Conference on Learning Representations*, 2023.

[79] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 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.

[80] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[81] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[82] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

[83] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*, 2023.

[84] Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A Smith. How language model hallucinations can snowball. In *International Conference on Machine Learning*, pages 59670–59684. PMLR, 2024.

[85] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. Siren's song in the ai ocean: A survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*, 2023.

[86] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *arXiv preprint arXiv:2409.20550*, 2024.

[87] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, and Bin Cui. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473*, 2024.

[88] Li Zhong and Zilong Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 21841–21849, 2024.

[89] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.