CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification

Yuchen Tian[1*], Weixiang Yan[2*†], Qian Yang[3,4], Xuandong Zhao[5]
Qian Chen[6], Wen Wang[6], Ziyang Luo[1], Lei Ma[7,8†], Dawn Song[5†]
[1]Hong Kong Baptist University
[2]University of California, Santa Barbara
[3]Mila - Québec AI Institute
[4]Université de Montréal
[5]University of California, Berkeley
[6]Alibaba Group
[7]The University of Tokyo
[8]University of Alberta
yctian@comp.hkbu.edu.hk, weixiangyan@ucsb.edu

Abstract

Large Language Models (LLMs) have made significant progress in code generation, offering developers groundbreaking automated programming support. However, LLMs often generate code that is syntactically correct and even semantically plausible, but may not execute as expected or fulfill specified requirements. This phenomenon of hallucinations in the code domain has not been systematically explored. To advance the community's understanding and research on this issue, we introduce the concept of **code hallucinations** and propose a classification method for code hallucination based on execution verification. We categorize code hallucinations into four main types: **mapping**, **naming**, **resource**, and **logic** hallucinations, with each category further divided into different subcategories to understand and address the unique challenges faced by LLMs in code generation with finer granularity. Additionally, we present a dynamic detection algorithm called **CodeHalu** designed to detect and quantify code hallucinations. We also introduce the **CodeHaluEval** benchmark, which includes 8,883 samples from 699 tasks, to systematically and quantitatively evaluate code hallucinations. By evaluating 17 popular LLMs using this benchmark, we reveal significant differences in their accuracy and reliability in code generation, offering detailed insights for further improving the code generation capabilities of LLMs. The CodeHalu benchmark and code are publicly available at https://github.com/yuchen814/CodeHalu.

Introduction

Deep neural networks often generate erroneous information that contradicts the original content, cannot be verified, or conflicts with real-world knowledge. This phenomenon, commonly known as model hallucination, attracts widespread attention in the fields of natural language processing and multimodal learning (Ji et al. 2023; Zhang et al. 2023; Liu et al. 2024), with the community actively exploring methods to mitigate hallucinations (Peng et al. 2023; Elaraby et al. 2023; Liu et al. 2023). However, the issue of model hallucination in the code generation domain remains unexplored.

Conducting a thorough and dedicated study on code hallucinations is crucial for improving the quality of code generated by LLMs. Firstly, the purpose of code is to solve problems, and its value is realized only when the code executes successfully and passes tests (Chen et al. 2021; Austin et al. 2021; Yan et al. 2023). This necessitates that the generated code not only maintain strict logic and precision but also undergoes execution verification to confirm its correctness. Therefore, the practical use and verification of code differ significantly from Natural Language(NL) texts, meaning we cannot directly apply the definitions and methods used for NL hallucinations to code. Secondly, code snippets containing hallucinations may trigger runtime errors, or exhibit functional defects, which hinder the reliable deployment of LLMs in automated software development scenarios. Lastly, by exploring and verifying code hallucinations in a targeted manner, we can effectively uncover their causes and contribute to improving the architecture and training methods of LLMs.

To fill this gap, we define the concept of *code hallucination* in LLMs, based on the unique purpose and function of the code. **Code hallucinations refer to the phenomenon where code generated by LLMs is syntactically correct or even semantically plausible but ultimately cannot execute as expected or fails to meet specified requirements.**[1] This phenomenon typically arises from various factors, such as errors or outdated information in the training data, an inadequate grasp of the syntax rules and programming paradigms of the programming languages, and limitations in the logical

[*]These authors contributed equally.
[†]The corresponding author.

[1]We test 16 LLMs using 105,958 code samples. The experimental results demonstrate that only 9 models occasionally exhibit syntactic errors in the generated code, with an exceptionally low average error rate of **0.0020**. These findings support our initial hypothesis that the code generated by LLMs is generally syntactically correct and even semantically plausible or appropriate.

processing capabilities of the models. In contrast to previous methods that *passively* explore hallucinations in NLP through a Q&A framework or by prompting LLMs to generate hallucinated answers (Lin, Hilton, and Evans 2021; Cheng et al. 2023), we employ an *active* strategy to detect hallucinations during the code generation process by LLMs. This approach is crucial as the ultimate goal of the generated code is to execute correctly and fulfill specific tasks.

To detect and quantify hallucinations in LLMs during code generation, we develop a dynamic detection algorithm named **CodeHalu**. This algorithm employs a statistical induction method based on execution validation to identify specific patterns that frequently occur in code generated by multiple LLMs, such as *error types*, *syntax interruptions*, or *unexpected execution results*. When a pattern consistently appears across multiple LLMs, it is recognized as a common code hallucination. Based on the CodeHalu algorithm 1, we employ an execution-based validation approach for hallucination detection, combined with a two-stage heuristic identification method. By conducting statistical quantification on 17 mainstream LLMs, we categorize code hallucinations into four major categories: **Mapping**, **Naming**, **Resource**, and **Logical Hallucinations**. These categories are further divided into eight subcategories, as illustrated in Figure 1. We analyze 17 LLMs for cross-task occurrence rates in eight categories of code hallucinations. The low average rate of 2.04% confirms the independence and validity of our classification.

To effectively measure and compare code hallucinations across different LLMs, we introduce an evaluation benchmark named **CodeHaluEval**, which is based on the incidence rate of hallucinations. It follows a structured process of *Validation-Identification-Construction* as shown in Figure 4 to detect and evaluate code hallucinations in LLMs, closely tied to real-world programming scenarios, ensuring that the generated code correctly achieves the expected functionality. CodeHaluEval encompasses eight types of code hallucinations as illustrated in Figure 1, covering 699 distinct tasks and corresponding to 8,883 samples. Additionally, we systematically evaluate 17 mainstream LLMs to reveal the distribution and behavior patterns of their code hallucinations. We also analyze the potential causes of various code hallucinations, providing detailed insights for further improving the code generation capabilities of LLMs. Our contributions can be summarized as follows:

- **Code Hallucination**: We introduce the concept of *code hallucination* in LLMs and propose an execution-based verification method to define code hallucination, addressing a gap in the research on hallucination within the code generation domain.
- **CodeHalu Algorithm**: We develop a dynamic detection algorithm, CodeHalu, to identify and quantify the types of hallucinations that occur in LLMs during code generation. We categorize code hallucinations into four main categories based on a two-stage heuristic approach, discussing their theoretical implications and potential causes.
- **CodeHaluEval Benchmark**: We propose the CodeHaluEval benchmark to systematically evaluate 17 popular LLMs, revealing the distribution and patterns of code hallucinations across these models, and providing insights for developing more robust and reliable LLMs.

Related Work

Hallucination

In the field of NLP, hallucination is initially defined as the phenomenon where the text generated by a model is fluent and natural but either lacks substantive meaning or is inconsistent with the provided source content (Ji et al. 2023). Recently, Zhang et al. (2023) standardize the definition of NL hallucinations in LLMs into three categories: *input-conflicting hallucinations*, where the content generated by LLMs diverges from the user's input; *context-conflicting hallucinations*, in which the generated content contradicts previously generated content; and *fact-conflicting hallucinations*, where the generated content conflicts with established world knowledge. These hallucinations are attributed to various factors, such as poor-quality data samples in the training dataset or the use of sampling algorithms with high uncertainty.

In the multimodal domain, Zhai et al. (2023) classify types of hallucinations in image-to-text scenarios, such as image captioning and visual question answering. They define three main types of hallucinations: *object existence hallucinations*, *object attribute hallucinations*, and *object relationship hallucinations*. In text-to-image scenarios, such as image generation, hallucinations refer to the creation of factually incorrect details by the image generation model in response to the given text input. Huang et al. (2024) introduce VHTest, which evaluates hallucinations across eight dimensions in images, including the *existence*, *shape*, *color*, *orientation*, *OCR*, *size*, *position*, and *counting* of visual objects. In text-to-video scenarios, such as video generation, Chu et al. (2024) define three types of hallucinations: *prompt consistency hallucinations*, *static hallucinations*, and *dynamic hallucinations*. Although the issue of hallucinations receives extensive attention in NLP and multimodal domains, it remains unexplored in the code domain. Therefore, we propose CodeHalu to systematically define, identify, classify, and quantify code hallucinations in LLMs.

Existing Coding Benchmarks

In recent years, numerous studies focus on evaluating the capability of LLMs to handle various programming tasks. Among these, the HumanEval (Chen et al. 2021), includes 164 Python programming problems, each with an average of 6.7 unit tests. The APPS (Hendrycks et al. 2021) benchmark presents more challenging programming questions, with each problem averaging 293.2 words in length. CodeScope (Yan et al. 2023) covers 43 programming languages and 8 coding tasks to evaluate LLMs in code understanding and generation. MMCode (Li et al. 2024) is designed to evaluate the programming capability of code models in multimodal scenarios. SWE-bench (Jimenez et al. 2023) evaluates the ability of LLMs to modify code repositories and solve problems with a complexity level comparable to what human programmers encounter. Overall, existing code benchmarks focus on evaluating the performance of LLMs on various programming tasks. However, there is still a lack of effective methods to

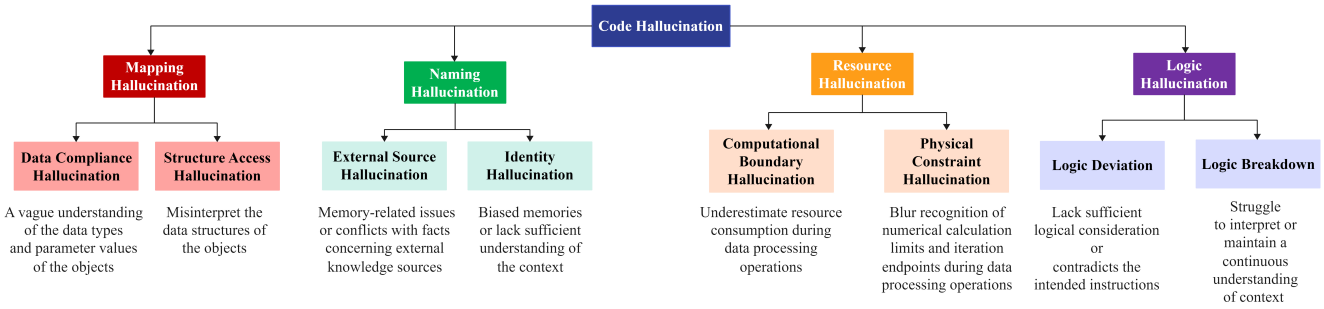

Figure 1: The definition and classification of code hallucinations, including 4 main categories and 8 subcategories.

detect and quantify potential hallucinations that may occur in code generation. Therefore, we propose CodeHaluEval to detect and quantify code hallucinations in LLMs.

Code Hallucination

As a tool, code aims to achieve specific objectives through correct execution. This inherent characteristic motivates our use of an execution-based verification method to explore and identify code hallucinations. In this section, we define the concept of *code hallucination* and distinguish it from code errors, clarifying the relationship and differences between these two phenomena.

Definition 1 (Code Hallucinations)**.** Code hallucinations refer to the code generated by large language models that is syntactically correct or even semantically plausible, but ultimately cannot execute as expected or fails to meet specified requirements.

Definition 2 (Code Errors)**.** Code errors refer to issues in a program that cause it to stop executing.

Remark 3 (Code Hallucinations vs. Code Errors). In multiple domains, existing work (Ji et al. 2023; Zhang et al. 2023; Huang et al. 2024; Zhai et al. 2023; Chu et al. 2024) often equates errors with hallucinations, or considers errors as a specific subset of hallucinations. We follow this perspective and regard code errors as a specific subset of code hallucinations. In other words, errors manifest as a form of hallucination, but not all hallucinations can be adequately expressed through errors. Figure 2 illustrates the distinction between code errors and code hallucinations. The code on the left exhibits a typical code error due to the use of an undefined variable "N", resulting in a NameError. On the right, the code repeatedly calls the same function due to a logical collapse during generation, eventually exceeding the maximum token limit and leading to a SyntaxError. However, the underlying issue is a latent logical hallucination, rather than the observed syntactic error.

Overall, although there is some slightly overlap between code hallucinations and code errors, their meanings, research objects, and scopes differ significantly. Code hallucinations focus on why the model produces hallucinations, while code errors focus on what grammatical rules the code violates. Code errors form a proper subset of code hallucinations, while code hallucinations encompass a broader range of potential logical and functional issues, representing a finer-grained and more comprehensive evaluation of the overall quality and functionality of the code.

Algorithm 1: CodeHalu Algorithm

Input: Code Generation Dataset α, Language models π
Output: HaluTypes ξ

Let $\xi \leftarrow$ empty list
for α_i, where $i \in \{1, \ldots, k\}$ **do**
  for π_j, where $j \in \{1, \ldots, m\}$ **do**
    $\mathsf{GC}_j^{\alpha_i} \leftarrow \pi_j(\mathsf{GI}_j, \mathsf{Q})$
    if $\mathsf{GC}_j^{\alpha_i}$ is stuttering, infinite enumeration, or gibberish
      $\xi \leftarrow \xi \cup \mathrm{State}(\mathsf{GC}_j^{\alpha_i})$
    else
      for t_n, where $n \in \{1, \ldots, N\}$ **do**
        if Execute($\mathsf{GC}_j^{\alpha_i}(t_n)$)
          $\mathsf{ER}_j^{\alpha_i}(t_n) \leftarrow$Execute($\mathsf{GC}_j^{\alpha_i}(t_n)$)
          if $\mathsf{ER}_j^{\alpha_i}(t_n) \neq op_{t_n}$
            $\xi \leftarrow \xi \cup \mathrm{State}(\mathsf{GC}_j^{\alpha_i})$
Aggregate and count frequencies of unique $\mathrm{State}(\mathsf{GC}_j^{\alpha_i})$ in ξ

CodeHalu Algorithm

In this section, we introduce a dynamic detection algorithm called **CodeHalu**, which detects and quantifies hallucinations in LLMs in code generation. CodeHalu operates on the assumption ASS: if a specific pattern frequently appears in the code generated by multiple LLMs, it is considered a common code hallucination. These patterns include *error types*, *syntax interruptions*, *logical collapse*, or *unexpected execution results*.

Consider a dataset α contains k samples, where each sample α_i consists of a problem description Q and a series of test cases $t_1, t_2, \ldots, t_n$. Each test case t_n includes an input ip and the corresponding expected output op. Notably, following previous work (Li et al. 2023b; Yan et al. 2023), we integrate resource (time and memory) constraints into the code generation instructions. As shown in Algorithm 1, we use a π_j to generate a code solution $\mathsf{GC}_j^{\alpha_i}$ for each sample α_i based on the code generation instruction $\mathsf{GI_j}$ and the problem description Q. If $\mathsf{GC}_j^{\alpha_i}$ exhibits any of the states such as stuttering, infinite loops, or gibberish, we include it in ξ.

To test the potential hallucinations of $\mathsf{GC}_j^{\alpha_i}$ at a fine-grained level, we execute all test cases of sample α_i one

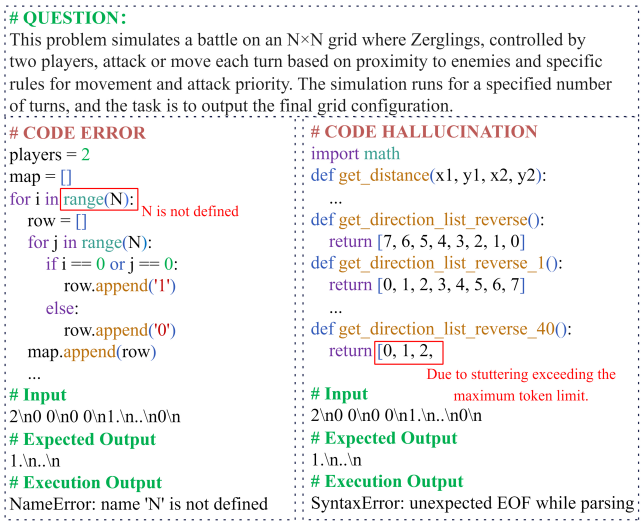

Figure 2: Examples that differentiate between code errors and code hallucinations.

by one to verify whether it successfully executes and meets the expected functionality. We record the actual execution result $\mathsf{ER}_j^{\alpha_i}(t_n)$ of the code under each test case t_n and extensively test each sample α_i across more than 15 π to obtain statistically-based inductive results. If the code execution fails or does not meet the expected results, we record it in ξ.

Finally, we merge the identical states $\mathrm{State}(\mathsf{GC}_j^{\alpha_i})$ detected by CodeHalu and calculate their occurrence frequencies. According to assumption ASS, ξ can be represented as $[(\xi_1, P_1), (\xi_2, P_2), \ldots, (\xi_o, P_o)]$, where ξ_o denotes the o^{th} type of code hallucination, and P_o indicates its corresponding frequency. **Code hallucinations come from four perspectives: errors, syntax, logic, and execution results**. Additionally, CodeHalu is language-agnostic and can dynamically adjust to match various programming scenarios depending on the programming language.

Code Hallucinations Classification

In this section, we analyze the hallucination states detected by the CodeHalu algorithm, classify and define four main types of hallucinations, and discuss the rationale behind the classification method.

According to the TIOBE Index[2], a metric of programming language popularity, we primarily investigate code hallucinations within the Python. By applying the CodeHalu algorithm on the complex APPS dataset (Hendrycks et al. 2021) and 17 widely-used LLMs, we identify and validate 18 types of hallucination states that violate human expectations during the code generation, including *inconsistent code context*, *ambiguous logic and data flow*, *conflicting intentions*, among others. Using the two-stage heuristic classification method introduced in Remark 8, we categorize code hallucinations into four main types based on the nature and origin of these phenomena: mapping hallucinations, naming hallucinations, resource hallucinations, and logical hallucinations, as illustrated in Figure 1.

Definition 4 (Mapping Hallucinations)**.** Mapping Hallucinations refer to the ambiguity and confusion that occur in LLMs' perception and mapping of data types, values, and structures during data operations. This phenomenon is further divided into two sub-categories: *data compliance hallucinations* and *structure access hallucinations*.

Data compliance hallucinations occur when LLMs have a vague understanding of the data types and parameter values of the objects being manipulated, resulting in generated code that attempts to perform type-mismatched or rule-violating operations.

Structure access hallucinations occur when LLMs misinterpret the data structures of the objects being manipulated, leading to generated code that attempts to access non-existent array indices or dictionary keys.

Definition 5 (Naming Hallucinations)**.** Naming Hallucinations refer to the memory-related issues and factual inaccuracies exhibited by LLMs when handling the naming, scope, and existence of variables, attributes, and modules. This phenomenon is further divided into two subcategories: *identity hallucinations* and *external source hallucinations*.

Identity hallucinations occur when LLMs possess biased memories or lack sufficient understanding of the context, leading to generated code that references undefined variables, accesses non-existent object properties, or uses unassigned variables in local scopes.

External source hallucinations occur when LLMs exhibit significant memory-related issues or obvious conflicts with facts concerning external knowledge sources, resulting in generated code that attempts to import non-existent modules or fails to correctly load modules from other paths.

Definition 6 (Resource Hallucinations)**.** Resource Hallucinations occur when LLMs lack an adequate perception and prediction of resource consumption and control flow of the generated code during execution. This phenomenon is further divided into *physical constraint hallucinations* and *computational boundary hallucinations*.

Physical constraint hallucinations arise when LLMs underestimate resource consumption during data processing operations, causing code failure due to exceeding memory capacity, stack depth, or other physical constraints.

Computational boundary hallucinations occur when LLMs blur recognition of numerical calculation limits and iteration endpoints during data processing operations, causing code failure due to numerical overflow or improper iteration control.

Definition 7 (Logic Hallucinations)**.** Logic Hallucinations refer to the discrepancies between the expected results and the actual outcomes after executing the code generated by LLMs, or outputs with low semantic density or even complete chaos. This phenomenon is further divided into *logic deviation* and *logic breakdown*.

Logic deviation occurs when LLMs generate code that lacks sufficient logical consideration or contradicts the intended instructions. While this hallucination may not cause

[2]https://www.tiobe.com/tiobe-index/

errors during execution, logical deviations or confusion result in outcomes that fail to meet the expected results.

Logic breakdown occurs when LLMs struggle to interpret or maintain a continuous understanding of context during code generation. This indicates that the models may lose direction while generating code, making it difficult to maintain strict consistency of contextual information.

Remark 8 (Discussion of Rationality). To ensure the rationality and effectiveness of our code hallucination classification method, we conduct in-depth analyses.

Firstly, we extensively reference classification methods for hallucinations in the fields of NLP and multimodal research (Zhang et al. 2023; Ji et al. 2023; Huang et al. 2024; Zhai et al. 2023; Chu et al. 2024), as well as methods for classifying code errors and vulnerabilities in software engineering (Pan et al. 2023; Wang et al. 2024; Huang et al. 2023). We adopt a two-stage heuristic classification strategy. Initially, team members independently review failed code cases and develop preliminary classification frameworks; then, we reach a consensus through collaborative discussions. This widely used approach ensures the adaptability and accuracy of our framework, enabling a systematic understanding of code hallucinations in LLMs.

Secondly, we analyze the cross-task occurrence rates of each model across eight categories of code hallucinations. The results show that the average cross-task occurrence rate for these categories is only 2.04%, confirming the independence and rationality of our classification. For the Gemma-7B model, which exhibits the most severe hallucinations in Table 2, only 1.07% of task samples show cross-task hallucinations, as illustrated in Figure 3.

Lastly, we conduct an empirical investigation of our classification results and design a questionnaire to evaluate the rationality of our method . The survey receives 23 responses, and after excluding seven respondents with less than three years of experience, we analyzed 16 valid responses. The survey results indicate a rationality rating of 91.08% for our classification method, further supporting its validity.

Cause Analysis of Code Hallucinations

In this section, we explore the potential causes of various hallucinations generated by LLMs, aiming to provide valuable insights for optimizing training data, training methods, model architecture, and alignment strategies.

Mapping hallucinations primarily stem from the model's misunderstanding of data types and structures. This phenomenon arises due to several factors: (1) The model generates code based on tokens, lacking insight into higher-level structures such as statements and functions (Yang, Liu, and Yin 2021); (2) When dealing with long-distance data dependencies, especially within complex code blocks, the model fails to continuously track the structure and state of variables, overly relying on local information while neglecting the importance of the overall context (Zhang et al. 2024); (3) The model does not explicitly perform type checking and structure matching during code generation, lacking static checking and error correction mechanisms.

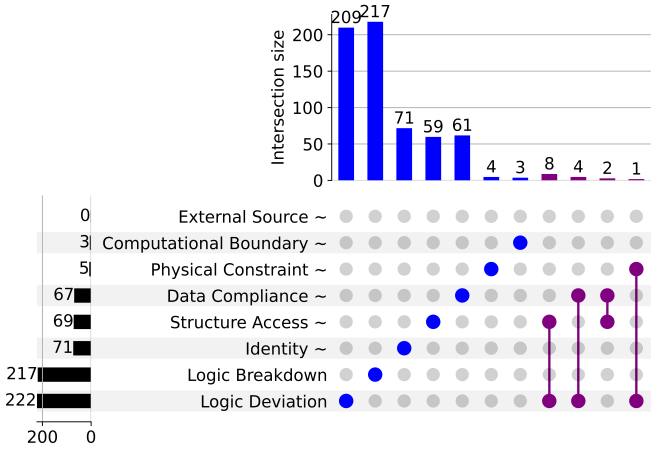

Figure 3: The diagram illustrates the intersection of various hallucinations in Gemma-7B during the CodeHaluEval. The bar chart at the top shows the frequency of each intersection, while the bar chart on the left indicates the frequency of each type of hallucination. The connecting lines represent the co-occurrence patterns between different hallucinations.

Naming hallucinations reflect the limitations of models in tracking information and utilizing external knowledge. This issue arises from several factors: (1) Token-based feature representation makes it difficult to accurately model long-distance dependencies, leading to model misjudgments regarding variable scope, lifecycle, and visibility (Xu et al. 2020); (2) The code generation process lacks consistency checks for identifiers and does not perform global tracking of variable definitions and usage; (3) Knowledge of external libraries is not effectively and timely integrated into the model's knowledge system, making it difficult for the model to accurately understand the names, functions, and invocation methods of libraries (Jesse et al. 2023).

Resource hallucinations highlight the model's lack of deep understanding of code execution mechanisms and physical constraints. These issues arise from several factors: (1) The training data lacks information related to resource consumption and performance optimization, making it difficult for the model to learn about complexity analysis and resource estimation; (2) As the model generates code based on probabilities, it lacks a module for calculating and estimating the resource consumption of the generated code, making it unable to simulate the real-world operating environment and resource limits; (3) During the model training process, the focus is usually on the correctness of the code's functionality, often overlooking its complexity and resource constraints in actual execution environments.

Logic hallucinations reveal the model's deficiencies in semantic understanding and reasoning about code. This issue arises due to several factors: (1) The model mainly relies on pattern matching and statistical rules to generate code, lacking a fundamental understanding of symbolic systems and rigorous verification of program logic; (2) The training data is often not rigorously verified for accuracy and may contain code with very similar functions. Since models some-

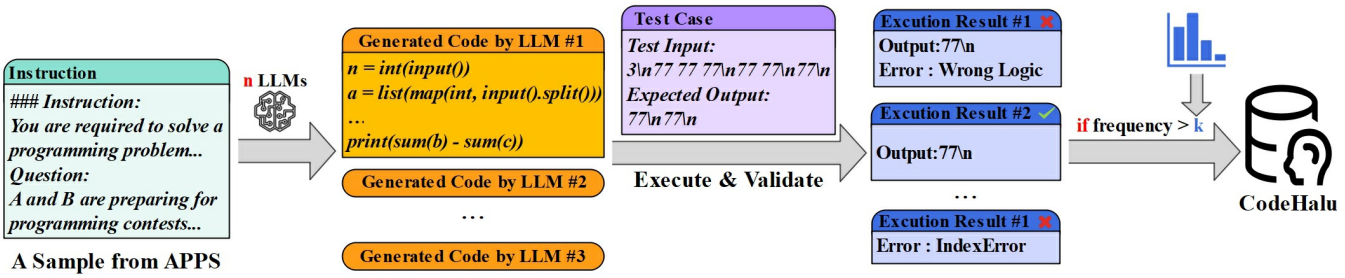

Figure 4: Collection of CodeHaluEval benchmark based on a **verification-identification-construction** process.

Category	#Tasks	#Samples	Sub-Category	#Tasks	#Samples
Mapping	262	2,288	**Data Compliance**	110	941
			Structure Access	152	1347
Naming	157	1,853	**Identity**	115	1323
			External Source	42	530
Resource	107	1,130	**Physical Constraint**	47	491
			Computational Boundary	60	639
Logic	173	3,612	**Logic Deviation**	119	2,443
			Logic Breakdown	54	1,169

Table 1: Detailed statistics of categories, and quantities in **CodeHaluEval** benchmark.

times imitate and memorize previous examples (Yan and Li 2022), this can result in the model directly replicating similar logic in the code or even learning incorrect logic from the outset; (3) When the model generates code, repetition at the line level has a self-reinforcing effect, causing the model to become increasingly confident in the code it generates, which may lead to a stuttering phenomenon (Xu et al. 2022).

The CodeHaluEval Benchmark

We construct the **CodeHaluEval** benchmark, a unified evaluation method for comparing various types and frequencies of hallucinations in code generation across different LLMs. We develop the CodeHaluEval benchmark based on the APPS testing set, following a structured process of **Validation-Identification-Construction**, as shown in Figure 4.

In the validation phase, we use the CodeHalu algorithm to identify multiple types of hallucinations HaluTypes ξ, represented as $[(\xi_1, P_1), \ldots, (\xi_i, P_i)]$. In the identification phase, we annotate the k^2 most common hallucinations and their frequencies in each sample α_i, represented as $[(\xi_1, P_1), \ldots, (\xi_{k^2}, P_{k^2})]$. In the construction phase, we sort all samples in descending order based on the frequency P_i of each hallucination type ξ_i. If the hallucination frequency in a sample α_i exceeds the threshold k, we include this sample in the corresponding hallucination type set in the CodeHaluEval benchmark. When selecting the threshold k, we consider both the minimum number of samples required to detect code hallucination effects in the CodeHaluEval benchmark and the inference costs associated with evaluating various LLMs. Through this method, we establish the CodeHaluEval benchmark, with detailed statistics shown in Table 1.

Experiments

Models. To comprehensively analyze the different hallucinations of various competitive LLMs in CodeHaluEval, we evaluate **12 general LLMs**, including GPT-4 (OpenAI 2023), GPT-3.5 (OpenAI 2023), Gemini-Pro-1.0 (Gemini 2023), Claude-3-haiku (Anthropic 2024), LLaMA-2 & 3 (Touvron et al. 2023), Vicuna (Chiang et al. 2023), Qwen-turbo (Bai et al. 2023), ChatGLM3-6B (Du et al. 2021), Ernie-3.5 (Baidu 2023), Mistral-7B (Jiang et al. 2023), Gemma (Team et al. 2024). We also evaluate **5 coding LLMs**, including Code LLaMA (Roziere et al. 2023), DeepSeek Coder (Guo et al. 2024), CodeGeeX-2 (Zheng et al. 2023), StarCoder-2 (Li et al. 2023a), MagicCoder-7B (Wei et al. 2023), WizardCoder-7B (Luo et al. 2023). The experimental evaluation is conducted using API calls or 8 NVIDIA A6000 GPUs.

Metrics. Given the limited exploration of code hallucinations, no dedicated metrics currently exist for evaluating them in LLMs. To address this gap, we propose an evaluation metric called **Hallucination Rate (HR)**. Specifically, HR is defined as the percentage of hallucination samples detected in the test set among all samples, with the formula: $\text{HR} = \frac{1}{N}\sum_{i=1}^{N} S(i, K)$, where $S(i, K)$ is an indicator function. If the i^{th} sample satisfies the hallucination condition, then $S(i, K) = 1$; otherwise, $S(i, K) = 0$. Ideally, a lower HR indicates a lower likelihood of hallucinations during code generation by the LLM, thus demonstrating greater robustness and reliability. To our knowledge, HR is the first metric that accurately reflects the hallucination phenomenon in LLMs during code generation tasks through actual execution tests.

Result & Analysis

The experimental results are presented in Table 2

Mapping hallucination: GPT-4 and GPT-3.5 consistently identify and follow rules related to data types, values, and structures, demonstrating strong context sensitivity.

Naming hallucination: Claude-3 reliably remembers and references entity names from the context and external knowledge bases. In contrast, LLaMA-2 exhibits significant memory bias when processing external knowledge and occasionally fabricates information.

Resource hallucination: GPT-4, Qwen, and LLaMA-2 effectively account for actual resource constraints when generating code, showing an understanding of computational

Model	Mapping (↓)			Naming (↓)			Resource (↓)			Logic (↓)			Average (↓)
	DC	SA	Avg.	ID	ES	Avg.	PC	CB	Avg.	LD	LB	Avg.	
GPT-4	32.31	**10.02**	**19.19**	27.74	0.57	19.97	**0.20**	3.76	**2.21**	85.76	0.51	58.17	**33.04**
LLaMA-3-8B	46.87	23.46	33.09	12.09	**0.00**	8.63	15.48	12.99	14.07	**78.39**	**0.00**	**53.02**	33.67
DeepSeek Coder-6.7B	24.23	25.61	25.04	15.80	**0.00**	11.28	17.52	17.21	17.35	99.06	0.17	67.05	38.28
GPT-3.5	**20.19**	22.05	21.28	30.54	**0.00**	21.80	18.53	6.42	11.68	99.88	**0.00**	67.55	38.98
Claude-3-haiku	38.68	29.25	33.13	**9.07**	0.75	**6.69**	37.07	20.81	27.88	100.00	0.17	67.69	41.00
ChatGLM-3-6B	36.13	44.91	41.30	50.87	**0.00**	36.32	24.85	2.35	12.12	88.99	**0.00**	60.19	44.23
Ernie-3.5	48.14	36.90	41.52	30.31	0.38	21.75	18.13	11.89	14.60	98.98	**0.00**	66.94	44.31
Qwen-turbo	49.63	48.33	48.86	29.48	2.08	21.64	7.94	2.82	5.04	98.08	0.17	66.39	44.74
MagicCoder-7B	50.27	26.58	36.32	17.69	**0.00**	12.63	21.18	28.33	25.22	100.00	16.25	72.90	44.84
Code LLaMA-7B	65.04	42.17	51.57	31.07	**0.00**	22.18	18.53	6.26	11.59	94.76	9.41	67.14	46.68
StarCoder-16B	48.14	38.83	42.66	60.70	9.25	45.98	28.92	11.11	18.85	95.09	0.77	64.56	49.23
LLaMA-2-7B	51.22	32.29	40.08	78.46	71.13	76.36	14.87	**0.00**	6.46	81.05	0.34	54.93	49.41
Gemini-1.0	34.11	53.53	45.54	45.88	**0.00**	32.76	24.44	16.12	19.73	98.65	10.35	70.07	49.57
Mistral-7B	45.48	36.53	40.21	59.18	15.85	46.79	27.49	10.80	18.05	99.35	0.17	67.25	49.76
WizardCoder-7B	26.57	31.40	29.41	31.29	**0.00**	22.34	33.20	9.39	19.73	93.90	72.37	86.93	50.10
CodeGeeX-2-6B	47.61	27.99	36.06	45.05	**0.00**	32.16	36.66	23.47	29.20	89.60	99.66	92.86	57.47
Gemma-7B	55.26	41.05	46.90	51.85	**0.00**	37.02	14.46	14.55	14.51	97.18	100.00	98.09	61.53

Table 2: Evaluation results of 17 models on CodeHalu. **DC** denotes Data Compliance hallucination. **SA** denotes Structure Access hallucination. **ID** denotes identity hallucination. **ES** denotes External Source hallucination. **PC** denotes Physical Constraint hallucination. **CB** denotes computational Boundary hallucination. **LD** denotes Logic Deviation. **LB** denotes Logic Breakdown.

boundaries and limitations, which leads them to produce code with lower complexity.

Logical hallucination: Although all models face challenges in maintaining logical coherence, LLaMA-3 and GPT-4 perform relatively well in reducing repetition. Most models rarely generate code with stuttering or infinite loops, but such issues are more common in Gemma, CodeGeeX-2, and WizardCoder, indicating a tendency to lose semantic and logical consistency during code generation.

Overall, GPT-4 and LLaMA-3 perform well across all hallucination categories, displaying stability and robustness in various scenarios. Logical hallucinations remain the most prevalent issue across all models, while naming and resource hallucinations are relatively less common. The performance of different models varies significantly across hallucination types, likely due to differences in their training data, methods, and architectures. The average hallucination rate ranges from approximately 20% to 60%.

We view mitigating code hallucination as future work. Based on a detailed analysis of experimental results and generated cases, we provide insights into strategies for mitigating code hallucinations in LLMs. In terms of training data, improving the quality and increasing the diversity of data sources enhances the model's generalization ability. In terms of training methods, employing alignment strategies based on compilation and execution verification, as well as setting multiple objectives during training, enables the model to better understand the data flow and control flow of code. In terms of model architecture, introducing a static code verification module provides real-time feedback on verification results, thereby enhancing the model's robustness. Additionally, incorporating a code graph module allows the model to construct and utilize graph structure information when generating code, deepening its understanding of patterns and logical relationships in the generated code.

Conclusion

We introduce the concept of code hallucination and propose an execution-based verification method to classify code hallucinations. We develop the dynamic detection algorithm, CodeHalu, and categorize code hallucinations into four main types, providing a comprehensive understanding of the various challenges faced by LLMs in code generation. Additionally, we establish the CodeHaluEval benchmark and evaluate 17 widely-used LLMs, revealing significant differences in their hallucination patterns during code generation, and providing detailed insights for further improving the code generation capabilities of LLMs. Overall, we lay the theoretical foundation for understanding the hallucination phenomenon of LLMs in code generation, and provide a complete set of tools for detecting and evaluating code hallucinations.

Limitations

Python is our focus for exploring code hallucination, as it is the most widely used programming language according to the TIOBE Index. Furthermore, many existing studies, such as HumanEval and MBPP benchmarks, concentrate on Python. Thus, we do not extend our investigation to other languages.

CodeHalu focuses on ensuring the correctness of generated code to meet the needs of developers and users. In contrast, identifying and preventing security risks is a higher-level concern, effectively addressed through sandbox environments. However, we recognize the importance of potential security risks and will consider them in future research.

We focus on code hallucination specifically within the code generation task, excluding other programming tasks such as code translation, and code repair. This is because code generation is currently the most widely studied task in the community. It is important to emphasize that but our hallucination detection and evaluation methods can be easily adapted to other tasks , which we consider as future work.

References

Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf.

Austin, J.; Odena, A.; Nye, M. I.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C. J.; Terry, M.; Le, Q. V.; and Sutton, C. 2021. Program Synthesis with Large Language Models. *CoRR*, abs/2108.07732.

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Baidu. 2023. Introducing ERNIE 3.5: Baidu's Knowledge-Enhanced Foundation Model Takes a Giant Leap Forward.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Cheng, Q.; Sun, T.; Zhang, W.; Wang, S.; Liu, X.; Zhang, M.; He, J.; Huang, M.; Yin, Z.; Chen, K.; et al. 2023. Evaluating hallucinations in chinese large language models. *arXiv preprint arXiv:2310.03368*.

Chiang, W.-L.; Li, Z.; Lin, Z.; Sheng, Y.; Wu, Z.; Zhang, H.; Zheng, L.; Zhuang, S.; Zhuang, Y.; Gonzalez, J. E.; et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*.

Chu, Z.; Zhang, L.; Sun, Y.; Xue, S.; Wang, Z.; Qin, Z.; and Ren, K. 2024. Sora Detector: A Unified Hallucination Detection for Large Text-to-Video Models. *CoRR*, abs/2405.04180.

Du, Z.; Qian, Y.; Liu, X.; Ding, M.; Qiu, J.; Yang, Z.; and Tang, J. 2021. Glm: General language model pretraining with autoregressive blank infilling. *arXiv preprint arXiv:2103.10360*.

Elaraby, M.; Lu, M.; Dunn, J.; Zhang, X.; Wang, Y.; and Liu, S. 2023. Halo: Estimation and Reduction of Hallucinations in Open-Source Weak Large Language Models. *CoRR*, abs/2308.11764.

Gemini. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Huang, K.; Meng, X.; Zhang, J.; Liu, Y.; Wang, W.; Li, S.; and Zhang, Y. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1162–1174.

Huang, W.; Liu, H.; Guo, M.; and Gong, N. Z. 2024. Visual Hallucinations of Multi-modal Large Language Models. *CoRR*, abs/2402.14683.

Jesse, K.; Ahmed, T.; Devanbu, P. T.; and Morgan, E. 2023. Large Language Models and Simple, Stupid Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 563–575. Los Alamitos, CA, USA: IEEE Computer Society.

Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y. J.; Madotto, A.; and Fung, P. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12): 1–38.

Jiang, A. Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D. S.; Casas, D. d. l.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825*.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Li, K.; Tian, Y.; Hu, Q.; Luo, Z.; and Ma, J. 2024. MMCode: Evaluating Multi-Modal Code Large Language Models with Visually Rich Programming Problems. arXiv:2404.09486.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, R.; Fu, J.; Zhang, B.-W.; Huang, T.; Sun, Z.; Lyu, C.; Liu, G.; Jin, Z.; and Li, G. 2023b. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Lin, S.; Hilton, J.; and Evans, O. 2021. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*.

Liu, F.; Lin, K.; Li, L.; Wang, J.; Yacoob, Y.; and Wang, L. 2023. Aligning large multi-modal model with robust instruction tuning. *arXiv preprint arXiv:2306.14565*.

Liu, H.; Xue, W.; Chen, Y.; Chen, D.; Zhao, X.; Wang, K.; Hou, L.; Li, R.; and Peng, W. 2024. A survey on hallucination in large vision-language models. *arXiv preprint arXiv:2402.00253*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

OpenAI. 2023. GPT-4 Technical Report.

Pan, R.; Ibrahimzada, A. R.; Krishna, R.; Sankar, D.; Wassi, L. P.; Merler, M.; Sobolev, B.; Pavuluri, R.; Sinha, S.; and Jabbarvand, R. 2023. Understanding the effectiveness of

large language models in code translation. *arXiv preprint arXiv:2308.03109*.

Peng, B.; Galley, M.; He, P.; Cheng, H.; Xie, Y.; Hu, Y.; Huang, Q.; Liden, L.; Yu, Z.; Chen, W.; et al. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Team, G.; Mesnard, T.; Hardin, C.; Dadashi, R.; Bhupatiraju, S.; Pathak, S.; Sifre, L.; Rivière, M.; Kale, M. S.; Love, J.; et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wang, Z.; Zhou, Z.; Song, D.; Huang, Y.; Chen, S.; Ma, L.; and Zhang, T. 2024. Where Do Large Language Models Fail When Generating Code? *arXiv preprint arXiv:2406.08731*.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and Zhang, L. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Xu, H.; van Genabith, J.; Xiong, D.; Liu, Q.; and Zhang, J. 2020. Learning Source Phrase Representations for Neural Machine Translation. In Jurafsky, D.; Chai, J.; Schluter, N.; and Tetreault, J., eds., *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 386–396. Online: Association for Computational Linguistics.

Xu, J.; Liu, X.; Yan, J.; Cai, D.; Li, H.; and Li, J. 2022. Learning to break the loop: Analyzing and mitigating repetitions for neural text generation. *Advances in Neural Information Processing Systems*, 35: 3082–3095.

Yan, W.; and Li, Y. 2022. WhyGen: explaining ML-powered code generation by referring to training examples. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 237–241.

Yan, W.; Liu, H.; Wang, Y.; Li, Y.; Chen, Q.; Wang, W.; Lin, T.; Zhao, W.; Zhu, L.; Deng, S.; et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*.

Yang, C.; Liu, Y.; and Yin, C. 2021. Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies. *Entropy*, 23(9).

Zhai, B.; Yang, S.; Zhao, X.; Xu, C.; Shen, S.; Zhao, D.; Keutzer, K.; Li, M.; Yan, T.; and Fan, X. 2023. Halle-switch: Rethinking and controlling object existence hallucinations in large vision language models for detailed caption. *arXiv preprint arXiv:2310.01779*.

Zhang, K.; Li, G.; Zhang, H.; and Jin, Z. 2024. HiRoPE: Length Extrapolation for Code Models. *arXiv preprint arXiv:2403.19115*.

Zhang, Y.; Li, Y.; Cui, L.; Cai, D.; Liu, L.; Fu, T.; Huang, X.; Zhao, E.; Zhang, Y.; Chen, Y.; et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.