Measuring the Mistakes: Evaluating Code Hallucinations in LLMs

Ishan Kavathekar[1*†] **Sriharini Margapuri**[2*] **Kritin Maddireddy**[2] **Adithya Casichetty**[2]
Ponnurangam Kumaraguru[2] **Karthik Vaidhyanathan**[2]
[1] Adobe Research, India
[2] International Institute of Information Technology, Hyderabad

Abstract

Large Language Models (LLMs) have shown strong capabilities in generating code from natural language, leading to their widespread adoption in modern software development through tools such as GitHub Copilot, Cursor, Codex, etc. However, these models are prone to generating code that often appears plausible but deviates from the user's intended requirement, a phenomenon known as *code hallucination*. Previous works on code hallucination have primarily examined code generation tasks, leaving other coding tasks underexplored. In this work, we conduct a systematic study of code hallucinations across four coding tasks, including code generation, debugging, refactoring, and editing. Furthermore, to analyze model behavior across programming languages, we study hallucinations in three widely adopted languages used in real-world systems: Python, C++, and Java. Finally, we introduce the Code Reliability Index, a task- and language-agnostic metric designed to quantify hallucination severity and assist practitioners in assessing the reliability of LLMs. Our results reveal that logic hallucinations constitute the majority of failures across tasks, reaching rates as high as 74%, while other hallucination types occur at rates of at most 20%. These findings suggest that, despite the widespread integration of LLMs into software development workflows, they remain insufficiently reliable for critical coding tasks, highlighting the need for robust mitigation strategies.

1 Introduction

LLMs have rapidly advanced in capabilities across various domains, enabling strong performance across a wide range of natural language and reasoning tasks. Despite these advancements and their widespread adoption in various fields such as software development, code assistance, healthcare, and finance, LLMs often hallucinate, i.e., generate outputs that contradict world knowledge or known facts (Ji et al., 2023). As LLMs are now widely being used for software development and code assistance, these hallucinations naturally extend to coding tasks. This phenomenon, known as *code hallucination*, occurs when models generate code that appears syntactically correct but fails to execute as intended or produces fundamentally incorrect logic (Tian et al., 2025). Prior work argues that many common code errors, including syntactic and runtime errors, can be viewed as code hallucinations, as they represent instances where the generated code deviates from correct or intended behavior (Tian et al., 2025). Even models achieving high performance on standard coding benchmarks often exhibit such failures (Liu et al., 2023). Moreover, executives at major companies, including Microsoft and Google, have publicly stated that 20–30% of the code written internally is now AI-generated or AI-assisted (Jordan Novet, 2025), amplifying the potential impact of code hallucinations in real-world software systems.

Prior benchmarks such as CodeHaluEval (Tian et al., 2025), CodeMirage (Agarwal et al., 2025), and HALoGEN (Ravichander et al., 2025) primarily evaluate hallucinations in code generation and largely restrict analysis to Python. While these studies provide valuable insights, they capture a limited subspace of real-world developer interactions with LLMs. In practice, developers rely on LLMs not only to generate new code but also to debug erroneous code snippets, refactor complex code, and apply localized edits to existing codebases. These tasks necessitate fundamentally different reasoning capabilities. For example, debugging requires identifying causal errors, while refactoring must preserve core semantics under structural change. Hallucinations in these tasks may arise in differ-

[*] Equal contribution
[†] This work was done during Ishan's time at IIIT-Hyderabad.

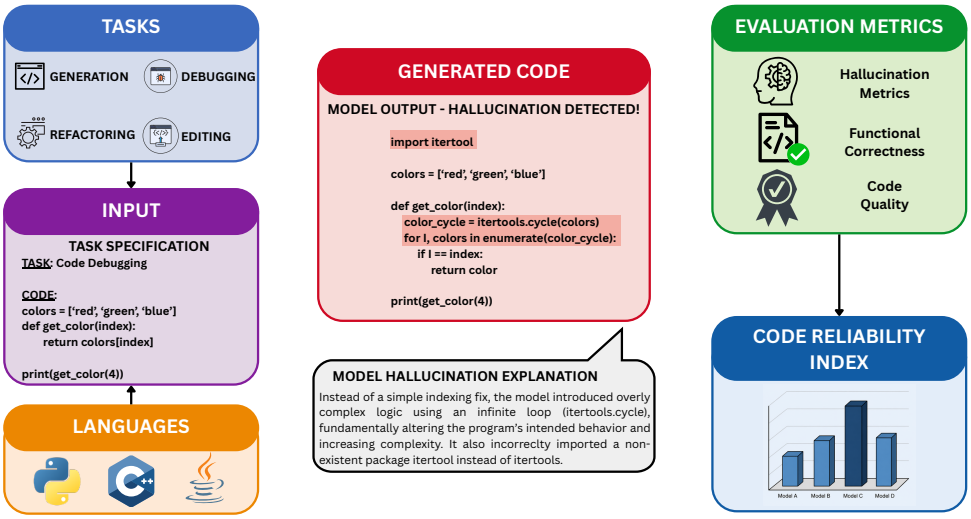


Figure 1: Overview of the evaluation pipeline. Given programming tasks, target languages, and input instructions, an LLM generates corresponding code solutions. The generated code is then assessed using multiple evaluation metrics, culminating in the Code Reliability Index leaderboard. The programming example provided in the figure is for illustrative purposes only; actual instances from our dataset involve varying degrees of complexity.

ent forms from those observed in previous benchmarks. Furthermore, real-world software systems are developed and maintained across diverse programming languages. Differences in syntax, type systems, and standard libraries may influence hallucination behavior, limiting the generalizability of Python-only results.

In this work, we conduct a systematic empirical study of code hallucinations across diverse coding tasks, including code generation, debugging, refactoring, and editing. We evaluate 10 LLMs on these tasks across three widely used programming languages: Python, Java, and C++, enabling a more comprehensive characterization of hallucination behavior beyond a single-language setting. This multi-task design across various programming languages allows us to uncover trends that provide new insights into why models struggle disproportionately in certain tasks and languages. Since modern coding workflows often involve iterative refinement, we additionally study how hallucinations evolve across multiple iterations, analyzing whether models are able to reduce hallucinations as they refine their outputs. Lastly, we introduce the **Code Reliability Index (CRI)**, a task- and language-agnostic metric designed to quantify the severity of hallucinations by jointly incorporating hallucination rate, code quality, and the functional correctness of the generated code. Figure 1 shows an overview of our evaluation pipeline.

Our contributions can be summarized as follows:

1. We conduct a comprehensive analysis of code hallucinations across four distinct coding tasks: Code Generation, Debugging, Refactoring, and Editing, analyzing task-dependent failure modes.

2. We evaluate code hallucinations across three widely used programming languages: Python, C++, and Java, to assess both their prevalence and cross-language consistency, highlighting language-specific capabilities.

3. We introduce the Code Reliability Index (CRI), a task- and language-agnostic metric that quantifies hallucination severity and enables systematic evaluation of LLM reliability across diverse coding scenarios.

2 Related Work

Hallucinations. Research on natural language generation errors has progressed from broad categorizations to finer-grained taxonomies (Ji et al., 2023). Rawte et al. (2023) introduced fine-grained categories of hallucinations based on degree, orientation, and type, defining two main forms of hallucinations: Factual Mirage and Silver Lining. To standardize evaluations, recent works have proposed benchmarks to measure hallucinations across diverse tasks and domains, including factual question answering, summarization, and domain-specific reasoning.

Code Hallucinations. While general text evaluation relies on semantic consistency, code hallucinations present unique challenges due to the strict semantic and syntactic constraints of programming languages (Ma et al., 2024). Several benchmarks quantify these hallucinations across specific tasks. For instance, the CodeHaluEval benchmark (Tian et al., 2025) defines code hallucination, introduces a taxonomy of four hallucination types: mapping, naming, resource, and logic hallucinations, and focuses on problems from the APPS dataset (Hendrycks et al., 2021). Similarly, the CodeMirage dataset (Agarwal et al., 2025) contains hallucinated test cases based on the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) datasets, to detect five types of hallucinations: dead-/unreachable code, syntactic incorrectness, logical error, robustness issue, and security vulnerabilities. Furthermore, HALoGEN (Ravichander et al., 2025) benchmarks hallucination across nine domains, with a particular focus on hallucinations related to code packages in programming.

Detection of Code Hallucinations. Prior work on code hallucination detection primarily falls into two categories: execution-based and static analysis approaches. Execution-based detection methods compile and execute the generated code against test cases, identifying hallucinations when the code fails to compile, produces runtime errors, or returns incorrect outputs. CodeHaluEval (Tian et al., 2025) adopts this approach by categorizing the hallucinations and measuring the hallucination rate across each type. Similar execution-based detection is used in Hallucode (Liu et al., 2026), while CoderEval (Yu et al., 2024) is a Python and Java benchmark with a self-contained execution framework to assess functional correctness of generated code. In contrast, static detection techniques rely on API usage checking, dependency analysis, and syntactic validation to identify naming- and dependency-related hallucinations. Zhuo et al. (2026) studied API misuse patterns in LLM-generated Python and Java code, identifying hallucinations that arise from incorrect method selection and parameter usage. Krishna et al. (2025) and Fan et al. (2026) analyzed package hallucinations and proposed heuristics to detect hallucinated dependencies that introduce security risks in the software supply chain.

3 Experimental Setup

3.1 Coding Tasks

We evaluate model performance across four distinct coding tasks: (i) Code Generation, (ii) Debugging, (iii) Refactoring, and (iv) Editing.

Code Generation: The input consists of a natural language problem statement along with a set of test case examples. The model is tasked with generating functional code that strictly adheres to these requirements. We employ a randomly selected subset of the APPS dataset (Hendrycks et al., 2021), which contains 500 problem statements (250 competition-style and 250 interview-style).

Debugging: The model is provided with an incorrect code snippet along with its corresponding error category. The model is tasked with identifying and correcting the error to produce a debugged version of the code. We conduct our evaluation using the CodeEditorBench dataset (Guo et al., 2025), which contains 268 instances.

Refactoring: The refactoring task requires improving code quality, such as understandability, maintainability, testability, etc., while preserving the overall functionality (Fowler, 2018). We employ two sources: (i) the CodeEditorBench dataset (Guo et al., 2025), which contains functional but unstructured, single-file code, and (ii) a set of established Code Refactoring Katas (Kata-Log, 2015), which features multi-file programs in which the task specifies structural improvements to the existing code under given constraints.

Editing: The input consists of a source code snippet paired with a natural language prompt that describes a required modification. The model is tasked with applying this specific change, such as adding features or altering logic, while preserving code integrity. We evaluate editing tasks using the EditEval benchmark (Li et al., 2024).

All the dataset preprocessing and filtering details along with sample examples are provided in Appendix A.

3.2 Pseudocode Evaluations

To further assess whether models understand fundamental programming concepts, we evaluate their ability to reason about algorithmic properties us-

ing pseudocode. Models are asked to infer time complexity and make binary judgments on logical reasoning, assessing reasoning over both computational resource constraints and algorithmic correctness. To construct the evaluation set, we use an existing dataset [1] containing code and annotated time complexities. We first convert the code into language-agnostic pseudocode and then generate incorrect variants by perturbing the pseudocode and use them to assess the model's ability to detect logical errors. To ensure the reliability of this evaluation set, all generated pseudocode instances are manually reviewed for semantic equivalence with the original code, and all perturbed variants are manually created and verified to contain a meaningful and detectable logical inconsistency. More details about the translation and perturbation process are mentioned in Appendix A.3

3.3 Evaluation Metrics

We evaluate models along three complementary axes to capture distinct aspects of the generated code: (i) hallucination rate, (ii) functional correctness measured via test case execution, and (iii) code quality assessed using standard software engineering metrics.

Hallucination Rate (HR): To quantify hallucination behaviour, we use Hallucination Rate (HR) (Tian et al., 2025), defined as the percentage of generated samples that exhibit code hallucination. For this, hallucinations are detected using an execution-based pipeline where generated code is compiled and executed. Execution traces, error outputs, and exit codes are analyzed using language-specific rules to classify hallucination types. A sample is assigned a value of 1 if any hallucination is detected and 0 otherwise. A lower HR value indicates fewer hallucinations. Full details of the hallucination detection process are provided in Appendix E. To provide granular insight, we utilize the CodeHalu taxonomy (Tian et al., 2025) to classify failures into four distinct categories:

- **Mapping Hallucinations:** Inconsistencies in the mapping of data types, values, or structures during data operations (e.g., accessing non-existent indices, type-mismatched operations).
- **Naming Hallucinations:** Inaccuracies in handling the naming, scope, and existence of variables, attributes, and modules (e.g., calling an undefined function, importing non-existent libraries).
- **Resource Hallucinations:** Failures arising from ignoring physical constraints or computational boundaries (e.g., stack overflow, out-of-memory errors).
- **Logic Hallucinations:** Discrepancies between the expected and actual execution outcomes, or incoherent control flow (e.g., missing base condition for recursive calls, algorithmically incorrect implementations).

Test Case Passing Rate (TCPR): TCPR computes the average fraction of test cases passed by each generated solution, allowing us to understand the model's underlying logic beyond a strict pass or fail outcome. A test case is considered passed if the generated code compiles, executes without runtime errors, and produces the expected output.

$$\text{TCPR} = \frac{1}{N} \sum_{i=1}^{N} \frac{P_i}{T_i} \tag{1}$$

where, N is the number of samples in the dataset, T_i is the number of test cases associated with sample i, and P_i is the number of test cases passed.

Code Quality Metrics: In real-world software engineering, a code snippet should be functionally correct as well as reusable, modular, and maintainable. Even if a solution passes all test cases, poor structure can hinder extensibility and debugging, especially in refactoring tasks, where improving internal code quality is the primary goal. Therefore, we evaluate generated code using metrics that go beyond correctness. Specifically, we calculate the Cyclomatic Complexity and Cognitive Complexity using AST parsing tools, specifically `radon` for Python and `tree-sitter`-based implementations for C++ and Java. More details about the implementation can be found in Appendix F.

- **Cyclomatic Complexity** measures the structural complexity of a program by calculating the number of linearly independent paths in its control-flow graph (McCabe, 1976). Higher values indicate more branching, which increases the number of execution paths

[1] https://huggingface.co/datasets/NiharMandahas/Time_Complexity

that must be tested, reducing maintainability (Denaro et al., 2002).

- **Cognitive Complexity** measures how difficult the code is for humans to understand (Campbell, 2018). It increases with code patterns such as deep nesting or long boolean expressions, which disrupt reading flow (Muñoz Barón et al., 2020). Consequently, code that is difficult to understand loses much of its practical value, even if it is functionally correct, as high complexity increases the likelihood of introducing defects during maintenance and debugging.

3.4 Models

We evaluate 10 LLMs, including both proprietary and open-source models spanning six model families. This list includes: (i) GPT-5 (OpenAI, 2025) (ii) GPT-5-mini (OpenAI, 2025) (iii) Gemini-2.5-Pro (Comanici et al., 2025) (iv) Gemini-2.5-Flash (Comanici et al., 2025) (v) Claude-Sonnet-4.5 (Anthropic, 2025) (vi) Llama-3.1-8B-Instruct (Grattafiori et al., 2024) (vii) DeepSeek-Coder-V2-Lite-Instruct (DeepSeek-AI et al., 2024) (viii) Phi-4 (Abdin et al., 2024) (ix) Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) and (x) Qwen2.5-Coder-14B-Instruct (Hui et al., 2024). Table 2 in Appendix B summarizes the specifications of all the models considered in our experiments.

4 Results

Overall Hallucination Trends

Table 1 presents the hallucination rates and test case passing rate across models and programming languages. Logic hallucinations are consistently the most prevalent across all tasks, indicating that models frequently struggle to correctly interpret problem constraints and generate semantically correct implementations. In contrast, mapping, naming, and resource hallucinations occur far less frequently, typically remaining below 20%, with resource hallucinations being particularly rare (at most 2%). These findings suggest that while current LLMs generally produce syntactically valid and efficient code, they often fail to accurately translate problem requirements into correct algorithmic solutions.

The prevalence of logic hallucinations highlights the gap between syntactic fluency and semantic understanding in current LLMs. Since models are trained on large corpora of syntactically valid code, they are proficient at reproducing common coding patterns, resulting in relatively few resource, naming, and mapping hallucinations. Logic hallucinations, however, require correctly translating natural language specifications into executable algorithms that satisfy all problem constraints. Unlike surface-level errors, such failures often remain hidden until execution, making them substantially harder to avoid.

Understanding Logical Reasoning through Pseudocode

The pseudocode evaluation provides additional insight into the source of logic hallucinations. As shown in Tables 3 and 4, models are generally capable of reasoning about the logical correctness of pseudocode and can reliably determine whether a solution is correct or incorrect. However, they perform substantially worse when reasoning about time complexity, with Phi-4 exhibiting the weakest performance among the evaluated models. These results suggest that models possess a basic understanding of algorithmic reasoning but often fail to fully interpret or satisfy the complete set of problem requirements during code generation.

Proprietary vs. Open-Source Models

We observe a clear distinction between proprietary and open-source models. Proprietary models consistently achieve lower hallucination rates across all coding tasks. GPT-5 exhibits the lowest hallucination rates, remaining below 22% across tasks and languages, whereas Llama-3.1-8B reaches rates between 40% and 74%. Interestingly, the performance gap is largest for code generation and considerably smaller for debugging and refactoring tasks. Since the latter tasks provide an existing implementation as a starting point, they reduce the need for unconstrained generation. This suggests that the primary advantage of proprietary models is in generating complete solutions from natural language specifications as compared to their open-source counterparts.

Cross-Language Hallucination Analysis

Across both proprietary and open-source models, C++ snippets consistently exhibit the highest hallucination rates, while Python and Java exhibit similar trends. This may be due to C++'s stricter semantic requirements, including explicit memory management, pointer arithmetic, and a more

		Code Generation					Code Debugging					Code Refactoring					Code Editing				
		Hallucination Rate (↓)				T (↑)	Hallucination Rate (↓)				T (↑)	Hallucination Rate (↓)				T (↑)	Hallucination Rate (↓)				T (↑)
Model	**Lang.**	**Map.**	**Name**	**Rsrc.**	**Logic**		**Map.**	**Name**	**Rsrc.**	**Logic**		**Map.**	**Name**	**Rsrc.**	**Logic**		**Map.**	**Name**	**Rsrc.**	**Logic**	
Proprietary Models																					
GPT-5	Python	1.40	0.00	0.00	21.20	85.12	0.37	0.00	0.00	11.19	**91.44**	0.37	0.00	0.38	1.71	**99.63**	3.80	1.90	0.00	12.66	88.76
	Java	0.80	0.20	0.00	20.20	85.22	0.37	1.49	0.00	13.06	87.57	0.00	0.37	0.00	3.04	98.36	0.00	5.70	0.00	18.35	83.78
	C++	0.00	0.20	0.00	20.60	**86.00**	0.00	1.12	0.00	22.01	84.52	0.00	0.00	0.00	16.51	92.06	0.63	2.53	0.00	10.13	84.02
GPT-5-mini	Python	1.20	0.20	0.00	28.00	81.13	0.00	**10.07**	0.00	14.93	79.41	0.75	0.00	0.00	2.13	99.13	2.53	1.27	0.00	17.09	85.48
	Java	1.40	0.20	0.20	23.80	82.81	0.37	1.49	0.00	14.93	87.11	0.75	0.00	0.00	3.84	98.21	0.63	7.59	0.00	20.25	77.26
	C++	0.00	0.20	0.00	26.80	81.80	0.00	1.12	0.00	29.85	80.20	0.00	0.00	0.00	19.04	91.22	0.63	5.06	0.00	6.96	78.30
Gemini-2.5-Pro	Python	1.80	0.40	0.00	22.20	82.92	0.75	0.00	0.00	13.43	89.61	0.00	0.00	0.00	9.15	95.87	3.16	2.53	0.00	12.03	**88.89**
	Java	1.60	0.00	0.40	23.60	83.79	0.75	1.49	0.37	12.69	88.16	0.38	0.38	0.00	7.04	95.83	0.64	6.37	0.00	17.20	80.45
	C++	0.00	1.00	0.00	21.20	83.46	0.00	1.49	0.00	21.27	84.59	0.00	0.00	0.00	22.45	88.59	0.63	4.43	0.00	10.13	82.40
Gemini-2.5-Flash	Python	2.00	0.00	0.00	30.20	73.13	0.37	0.00	0.00	20.15	83.60	1.89	0.00	0.00	11.77	93.46	3.16	1.90	0.00	17.09	85.75
	Java	0.80	0.60	0.20	21.66	67.15	0.37	1.87	0.00	18.28	82.37	0.38	0.00	0.00	12.35	91.85	0.63	8.23	0.00	20.89	78.08
	C++	0.00	3.20	0.00	24.40	70.70	0.00	1.12	0.00	28.36	78.79	0.00	0.91	0.00	25.65	87.70	0.63	1.90	0.00	11.39	81.04
Claude-Sonnet-4.5	Python	1.60	4.00	1.20	37.40	66.25	0.75	0.00	0.00	17.16	87.47	1.12	0.00	0.00	9.67	94.15	6.33	0.63	0.00	16.46	85.69
	Java	1.60	0.60	1.40	43.20	65.32	0.37	1.49	0.00	16.04	85.02	0.00	0.37	0.00	8.24	94.94	0.63	5.06	0.00	22.15	78.92
	C++	0.00	2.40	0.20	44.00	63.94	0.00	1.12	0.00	23.51	82.22	0.00	1.18	0.00	26.05	83.13	0.63	3.16	0.00	12.03	79.94
Open Source Models																					
Llama-3.1-8B-Instruct	Python	**18.60**	1.60	**1.60**	73.00	19.37	5.22	0.37	0.00	39.18	65.78	**13.81**	4.85	0.00	**64.71**	50.04	7.59	6.33	0.63	32.28	69.55
	Java	15.80	1.00	0.20	65.20	18.34	1.87	2.24	0.37	31.34	69.51	6.74	5.99	0.47	47.87	59.63	4.43	9.49	0.00	39.24	58.23
	C++	0.30	7.40	0.80	**74.00**	18.40	0.00	3.73	0.00	**41.42**	57.06	1.48	13.06	0.61	49.74	38.08	1.27	9.49	0.00	27.22	46.38
DeepSeek-Coder-V2-Lite-Instruct	Python	7.80	2.20	0.40	65.60	27.59	**8.58**	2.99	**0.75**	29.10	62.18	5.24	3.37	0.00	57.97	53.19	**11.39**	3.80	1.27	**41.77**	60.95
	Java	4.80	2.20	0.80	54.40	28.72	4.10	3.36	0.00	28.73	65.26	4.12	10.49	0.92	29.15	63.66	3.16	2.53	**1.90**	39.87	57.73
	C++	0.00	6.20	0.00	59.40	29.16	0.00	4.85	0.00	35.45	54.75	0.00	8.88	0.00	30.78	55.62	0.63	8.23	0.00	38.61	45.31
Phi-4	Python	7.40	1.00	0.60	60.00	43.84	2.61	0.37	0.37	25.00	75.40	2.99	1.12	0.00	18.13	88.25	6.33	2.53	0.00	25.32	79.73
	Java	9.60	1.40	0.40	53.60	46.55	0.75	2.24	0.00	20.52	78.04	1.12	0.37	0.47	15.81	90.24	0.63	8.23	0.00	27.85	72.21
	C++	0.00	6.40	0.00	57.50	44.24	0.00	0.75	0.00	37.69	67.11	0.30	2.66	0.59	19.95	77.35	1.27	10.13	0.00	20.25	59.98
Qwen2.5-Coder-7B-Instruct	Python	11.80	1.80	0.80	72.00	23.41	4.85	1.12	0.37	32.46	68.60	10.82	3.36	0.00	38.81	69.42	5.69	6.33	0.00	22.78	76.21
	Java	8.40	2.40	0.60	73.80	25.28	1.87	2.24	0.00	26.49	73.06	5.97	3.36	**1.98**	42.66	66.46	3.16	4.43	0.00	33.54	69.87
	C++	0.00	7.80	0.00	71.40	26.54	0.00	2.24	0.00	37.69	64.70	0.30	**15.98**	0.30	42.45	44.54	1.27	**10.76**	0.00	20.89	59.50
Qwen2.5-Coder-14B-Instruct	Python	10.80	2.00	0.40	70.60	33.54	4.10	0.37	0.37	27.24	74.27	2.29	1.91	0.00	19.87	86.52	5.06	3.80	0.00	22.15	80.95
	Java	4.20	0.60	0.80	68.00	33.04	1.87	1.49	0.37	21.64	76.60	1.12	0.75	0.00	12.31	90.58	2.53	6.96	0.00	25.32	73.40
	C++	0.00	**9.80**	0.00	65.40	29.49	0.00	0.75	0.00	36.94	66.91	0.30	3.55	0.00	23.99	77.20	1.27	6.33	0.00	22.15	69.20

Table 1: Performance of LLMs (in %) across various coding tasks, measured by hallucination rate (mapping, naming, resource, and logic hallucination categories (lower value is better) and test case passing rate (higher value is better). Map., Name, Rsrc., and Logic indicate the hallucination rates for Mapping, Naming, Resource, and Logic hallucinations, respectively. T represents TCPR for each model. The highest values for each column are in **bold**. The bootstrapped results are provided in C.3.

restrictive type system, which make correct code generation more challenging.

Hallucinations Across Coding Tasks

Next, we analyze how hallucination rates vary across coding tasks. As shown in Figure 2, code generation exhibits the highest hallucination rates compared to other tasks. This is likely because models have to generate a complete solution from scratch compared to only modifying the code in other tasks. Editing produces the highest average rate of naming hallucinations because edit instructions frequently introduce new identifiers, functions, or components that increase the likelihood of inconsistent naming. Debugging exhibits negligible mapping, naming, and resource hallucinations because the task primarily requires identifying and correcting code within an existing implementation, while logic hallucinations remain dominant. Refactoring and editing achieve the lowest rates of logic hallucinations since the input programs are already functionally correct and only require structural improvements. Nevertheless, both tasks still exhibit average logic hallucination rates of approximately 22–23%.

5 Do Hallucinations Self-Correct? An Iterative Analysis

LLMs have demonstrated the ability to reflect on and revise their own outputs through iterative interactions. Prior work has shown that models can generate feedback on their initial responses and subsequently refine them, often leading to improved reasoning and task performance (Madaan et al., 2023; Lu et al., 2023). These capabilities are particularly relevant in coding workflows, where developers frequently interact with models over multiple turns, providing error messages or execution feedback that the model can use to revise its code. To study how code hallucinations evolve under iterative interactions, we analyze hallucinations across multiple iterative feedback rounds. Specifically, we study this for the code generation task by re-prompting the model (using both standard prompting and Chain-of-Thought (CoT) prompting) up to five turns using the previous response and its execution traceback as context, while keeping the task instruction fixed. At each turn, we compute the hallucination rate to analyze how it evolves across iterations and whether hallucinations are self-correcting or persistent.

We observe a decrease in overall hallucination rates during the first one to two iterations for most models. However, as illustrated in Figure 4, the

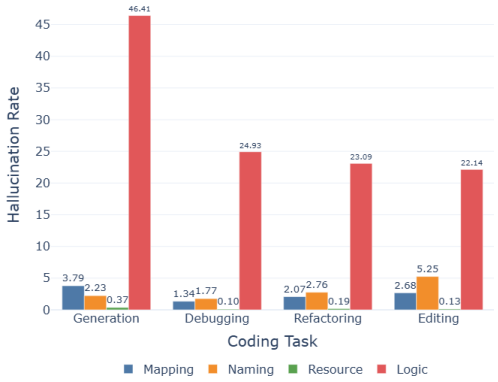


Figure 2: Breakdown of hallucination rates by category across coding tasks. Logic hallucinations occur most frequently while resource hallucinations are the least common.

rates tend to stabilize after that with only minor or no further improvements. In the case of the Qwen models, hallucination rates remain largely unchanged across iterations and stay relatively high. In contrast, Llama-3.1-8B-Instruct, Claude-Sonnet-4.5, and Phi-4 models show more consistent improvements across successive iterations. We discuss type-wise self-fixing behavior in Appendix C. Overall, these results suggest that simply providing execution traceback errors as feedback is insufficient for effectively mitigating hallucinations, highlighting the severity of hallucinations and the need for more robust mitigation strategies. Unlike standard self-fixing, which consistently reduces hallucination rates over five attempts, CoT self-fixing plateaus almost immediately and maintains roughly the same hallucination rate across iterations. This effect is strongest for Llama-3.1-8B-Instruct, DeepSeek-Coder-V2-Lite, and Phi-4, where standard self-fixing reduces hallucinations by 30–38% compared to 2–10% for CoT.

6 Code Reliability Index (CRI)

Code hallucinations can manifest in various forms, such as the use of non-existent APIs, logically inconsistent implementations, or syntactically valid but semantically meaningless constructs, each affecting the code's usability to varying degrees. Importantly, not all hallucinations affect the use cases equally: some result in minor inefficiencies, while others render the generated code entirely non-functional or unsafe. Existing evaluations typically report hallucination rates or functional correctness independently, making it difficult to compare systems holistically. To address this limitation, we introduce the **Code Reliability Index (CRI)**, a unified metric that combines executability, functional correctness, structural quality, and resource efficiency into a single severity score.

Mapping and naming hallucinations correspond to references to incorrect or non-existent APIs, libraries, or identifiers, preventing the generated code from executing. Since no further evaluation is meaningful for non-executable code, we first define a binary blocking indicator over dataset D,

$$\begin{aligned}\text{blocking}_i = \mathbf{1}[\text{mapping hallucination}_i] \\ \vee\, \mathbf{1}[\text{naming hallucination}_i]\end{aligned} \tag{2}$$

The dataset-level blocking rate is

$$p_B = \frac{1}{|D|} \sum_{i \in D} \text{blocking}_i, \tag{3}$$

and the executable subset is $D_{\text{exec}} = \{i \in D : \text{blocking}_i = 0\}$.

For executable programs, we measure functional correctness using TCPR. Using TCPR provides an objective measure of correctness without requiring manual annotation of logic hallucinations. The failure rate for sample i is

$$f_i = 1 - \text{TCPR}_i \tag{4}$$

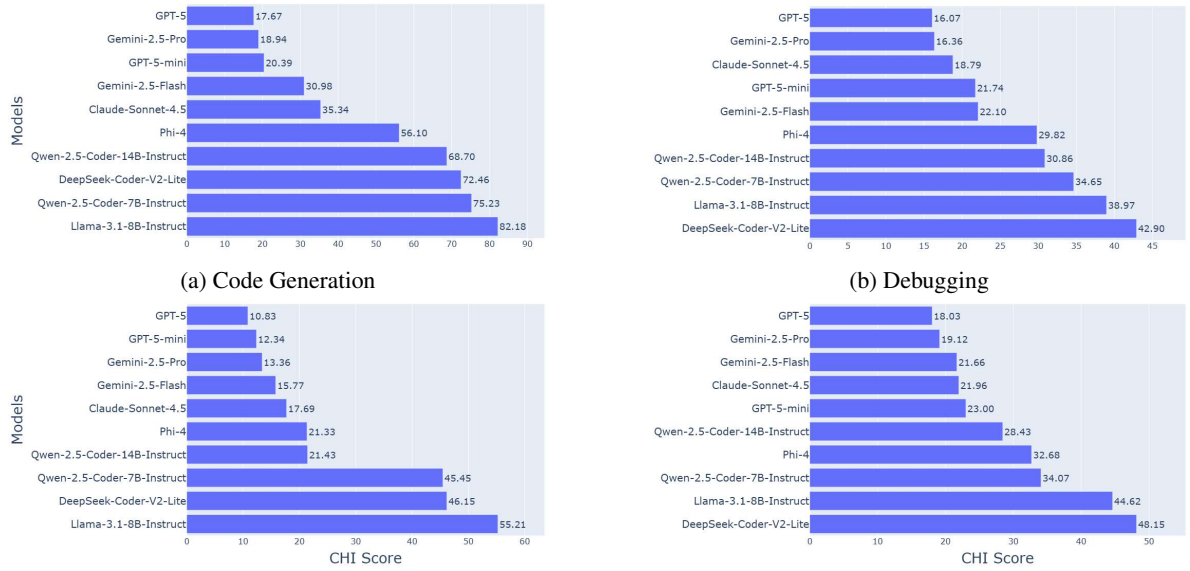
Beyond correctness, hallucinations may increase structural complexity or resource consumption, reducing maintainability and efficiency. We therefore evaluate both aspects for executable programs. Let c_i^{cyc} and c_i^{cog} denote the min–max normalized cyclomatic and cognitive complexity scores. Since these metrics capture complementary aspects of structural complexity and we wish to emphasize the dominant contributor while allowing partial compensation, we aggregate them using the generalized (power) mean with exponent $p = 4$:

$$\tilde{C}_i = \left(\frac{(c_i^{\text{cyc}})^4 + (c_i^{\text{cog}})^4}{2} \right)^{1/4} \tag{5}$$

Similarly, let t_i and m_i denote normalized runtime and memory usage. As these represent complementary resource costs with equal importance, we aggregate them using the arithmetic mean,

$$S(t_i, m_i) = \frac{t_i + m_i}{2} \tag{6}$$

Structural complexity and resource consumption jointly characterize implementation quality. To ensure that severe degradation in either aspect has



(a) Code Generation

(b) Debugging

(c) Refactoring

(d) Editing

Figure 3: Code Reliability Index (CRI) for individual coding tasks. Models are ranked separately for each task according to their CRI scores, with lower scores indicating better performance. GPT-5 consistently achieves the lowest CRI across all coding tasks, while model rankings remain largely consistent across tasks.

greater influence while still permitting reasonable trade-offs, we again employ the generalized mean with $p = 4$,

$$\tilde{U}_i = \left(\frac{\tilde{C}_i^4 + S(t_i, m_i)^4}{2} \right)^{1/4} \tag{7}$$

The generalized mean interpolates between the arithmetic mean ($p = 1$) and the maximum operator ($p \to \infty$). We set $p = 4$ to moderately prioritize the weaker of structural quality and resource efficiency. However, implementation quality is meaningful only after functional correctness is established. We therefore gate the quality penalty by correctness,

$$Q_i = f_i + (1 - f_i)\gamma \tilde{U}_i, \qquad \gamma \in (0, 1) \tag{8}$$

When a program fails all test cases ($f_i = 1$), its severity is maximal regardless of implementation quality. Conversely, quality contributes only to the portion of severity not already explained by incorrectness. We set $\gamma = 0.5$ throughout the paper and study its sensitivity in Appendix G.2.

Hence, the per-example score is

$$\text{CRI}_i = \begin{cases} 1, & \text{blocking}_i = 1, \\ Q_i, & \text{otherwise} \end{cases} \tag{9}$$

And the dataset-level CRI is

$$\text{CRI} = \frac{1}{|D|} \sum_{i \in D} \text{CRI}_i = p_B + (1 - p_B)\bar{Q}_{\text{exec}}, \tag{10}$$

where

$$\bar{Q}_{\text{exec}} = \frac{1}{|D_{\text{exec}}|} \sum_{i \in D_{\text{exec}}} Q_i \tag{11}$$

Finally, we report CRI on a 0–100 scale,

$$\text{CRI}_{100} = 100 \times \text{CRI}, \tag{12}$$

Across all four tasks, closed-source models achieve substantially lower CRI scores than open-source models, indicating more reliable code generation. GPT-5 performs the best overall, attaining the lowest CRI score in every task, with Gemini-2.5-Pro consistently close behind. Among open-source models, Phi-4 is the strongest performer across all tasks, while Qwen2.5-Coder-14B-Instruct also performs well, particularly in refactoring and editing. Llama-3.1-8B-Instruct and DeepSeek-Coder-V2-Lite are the weakest models overall, each ranking last in different tasks.

7 Conclusion

In this work, we present an empirical evaluation of code hallucinations extending beyond the code generation task. By evaluating 10 LLMs across four distinct software engineering tasks and the three most commonly used programming languages, we identified significant gaps in current performance. We introduce the Code Reliability Index, a unified measure of hallucination severity considering both functional correctness and code quality. Our results

revealed that hallucinations are both widespread and language-specific across tasks. These findings demonstrate that code hallucinations remain a fundamental limitation of current LLMs and underscore the need for targeted mitigation strategies to ensure reliable use of LLMs in real-world software development.

Limitations

In this work, we study the phenomenon of code hallucinations in various coding tasks across languages. However, there are some limitations to this work, which we discuss below:

- **Coding Tasks:** While our evaluation covers a subset of coding tasks, such as code generation, debugging, refactoring, and editing, it does not exhaustively cover all possible programming tasks. As a result, our findings may not fully capture hallucination behaviors that arise in other scenarios, such as software testing, security analysis, or repository-level reasoning. Future work can explore hallucinations in these tasks to better understand nuanced patterns across these tasks.

- **Language Diversity:** We consider only three widely used languages for our study and do not explore other paradigms such as functional, scripting, and logic programming languages. Our analysis may overlook hallucination patterns that are specific to other programming languages or type systems, which we leave out for future work.

References

Marah Abdin et al. 2024. Phi-4 technical report. *Preprint*, arXiv:2412.08905.

Vibhor Agarwal, Yulong Pei, Salwa Alamir and Xiaomo Liu. 2025. Codemirage: Hallucinations in code generated by large language models. *Preprint*, arXiv:2408.08333.

Anthropic. 2025. Claude sonnet 4.5 system card. Accessed: 2025-12-24.

Jacob Austin et al. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 57–58, New York, NY, USA. Association for Computing Machinery.

Mark Chen et al. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Gheorghe Comanici et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

DeepSeek-AI et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

Giovanni Denaro, Sandro Morasca and Mauro Pezzè. 2002. Deriving models of software fault-proneness. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, page 361–368, New York, NY, USA. Association for Computing Machinery.

Dongyang Fan, Sebastien Delsad, Nicolas Flammarion and Maksym Andriushchenko. 2026. Halluhard: A hard multi-turn hallucination benchmark. *arXiv preprint arXiv:2602.01031*.

Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Aaron Grattafiori et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Jiawei Guo et al. 2025. Codeeditorbench: Evaluating code editing capability of large language models. *Preprint*, arXiv:2404.03543.

Dan Hendrycks et al. 2021. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Binyuan Hui et al. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Ziwei Ji et al. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Jonathan Vanian Jordan Novet. 2025. Satya nadella says as much as 30% of microsoft code is written by ai. CNBC, Last Accessed: 2026-03-10.

Kata-Log. 2015. Refactoring katas. `https://kata-log.rocks/refactoring`. Accessed: 2025-10-12.

Arjun Krishna, Erick Galinkin, Leon Derczynski and Jeffrey Martin. 2025. Importing phantoms: Measuring llm package hallucination vulnerabilities. *Preprint*, arXiv:2501.19012.

Kaixin Li et al. 2024. InstructCoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 473–493, Bangkok, Thailand. Association for Computational Linguistics.

Fang Liu et al. 2026. Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code . *IEEE Transactions on Software Engineering*, 52(03):1037–1055.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Jianqiao Lu et al. 2023. Self: Self-evolution with language feedback. *arXiv preprint arXiv:2310.00533*.

Wei Ma et al. 2024. Lms: Understanding code syntax and semantics for code analysis. *Preprint*, arXiv:2305.12138.

Aman Madaan et al. 2023. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

T. J. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320.

Marvin Muñoz Barón, Marvin Wyrich and Stefan Wagner. 2020. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, page 1–12. ACM.

OpenAI. 2025. Gpt-5 system card. Accessed: 2025-12-16.

Abhilasha Ravichander, Shrusti Ghela, David Wadden and Yejin Choi. 2025. HALoGEN: Fantastic LLM hallucinations and where to find them. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1402–1425, Vienna, Austria. Association for Computational Linguistics.

Vipula Rawte et al. 2023. The troubling emergence of hallucination in large language models - an extensive definition, quantification, and prescriptive remediations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2541–2573, Singapore. Association for Computational Linguistics.

Yuchen Tian et al. 2025. Codehalu: investigating code hallucinations in llms via execution-based verification. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press.

Hao Yu et al. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Terry Yue Zhuo et al. 2026. Identifying and Mitigating API Misuse in Large Language Models . *IEEE Transactions on Software Engineering*, 52(03):855–873.

Appendix

A Dataset Preprocessing

To ensure the validity and cross-lingual comparability of our evaluation, we had to apply the following modifications to the source datasets:

A.1 Datapoint Filtering and Selection

We applied the following filtering criteria to the datasets mentioned below:

- For CodeEditorBench, we identified inconsistencies in the ground-truth solutions, including cases where reference implementations failed some of the provided test cases and instances where the reported errors did not correspond to the actual runtime errors. Because correcting these issues would require extensive manual verification, we removed such datapoints from our study. After filtering, the final dataset contains 268 datapoints per language for the code debugging task. In this task, we only retained datapoints that were consistently available across Python, Java, and C++ to maintain cross-language uniformity.

  For the code refactoring task, the same filtering procedure was applied to remove inconsistent datapoints; however, strict cross-language alignment was not required because the task only involves refactoring the provided code rather than reproducing a specific bug or error condition. Therefore, all valid datapoints were retained, resulting in 268 datapoints each for Python and Java, and 338 datapoints for C++.

- For the EditEval dataset, a small number of datapoints contained language-specific syntax that were not directly translatable across languages. To ensure consistency and comparability across languages, we excluded these datapoints from our analysis. The final dataset therefore contains 158 datapoints per language for the code editing task.

- For the Code Refactoring Katas dataset, some katas were excluded because they do not provide implementations in either Python and/or C++, which are required for our experiments. Specifically, the *Task List* and *Tell Don't Ask* katas were removed. For the *Tennis* kata, although multiple variants exist, only variants 1 to 4 were retained as they are stable and have working tests across all three languages. For the same reason, for the *Yahtzee* kata, only variant 1 was used. The rest of the Katas have been used as is. The refactoring instructions provided with these katas have been included in the prompt as part of the instructions to the LLM, and the final refactored code was tested on the provided test cases.

A.2 Cross-Lingual Dataset Translation

The EditEval dataset consists only of Python samples. All the datapoints were translated to the corresponding C++ and Java snippets using the ChatGPT interface and manually verified by the authors.

A.3 Pseudocode Evaluation Dataset

We assess whether models can (i) infer the time complexity of a given pseudocode and (ii) determine whether a pseudocode is logically correct. To construct the dataset for this evaluation, we begin with code samples that contain annotated time complexities. We first convert these code snippets into language-agnostic pseudocode using GPT-5.3 Codex with the prompt described in Appendix J. The generated pseudocode serves as the base set for the evaluation. We then manually perturb the pseudocode to introduce logical inconsistencies or incorrect algorithmic steps. These perturbed versions, along with the original pseudocode, are used to evaluate whether models can correctly identify logical errors.

B Models

We evaluate 10 models in our study, including both open-source and proprietary models. We specify all the details for the models in Table 2. All experiments are conducted with a temperature of zero, except for GPT models; at the time of evaluation, GPT-5 and GPT-5-mini did not expose temperature parameters via the API, so we used the default sampling configuration.

C Additional Results

C.1 Pseudocode Results

To assess whether models possess the underlying logical reasoning required for coding tasks, we evaluate them using pseudocode rather than full programming language implementations. This approach allows us to isolate the core reasoning process by removing syntactic complexity and language-specific constraints. By focusing solely

Model Family	Provider	Model Name	Parameters
Proprietary Models			
GPT	OpenAI	GPT-5	-
		GPT-5-mini	-
Gemini	Google	Gemini-2.5-Pro	-
		Gemini-2.5-Flash	-
Claude	Anthropic	Claude-Sonnet-4.5	-
Open-Source Models			
Qwen	Alibaba Cloud	Qwen2.5-Coder-14B-Instruct	14B
		Qwen2.5-Coder-7B-Instruct	7B
DeepSeek	DeepSeek	DeepSeek-Coder-V2-Lite-Instruct	16B
Phi	Microsoft	Phi-4	14B
Llama	Meta	Llama-3.1-8B-Instruct	8B

Table 2: Models evaluated in this study across different model families. Parameter sizes are reported where publicly available; for proprietary models, this information is not disclosed and therefore not included.

on pseudocode, we aim to determine whether the models can correctly infer the algorithmic logic needed to solve the task.

As illustrated in Table 3, most models perform reasonably well in this setting, achieving F1 scores ranging from 0.66 to 0.96, suggesting that many models are capable of capturing the fundamental reasoning required for the problems when syntax is not a limiting factor.

We further evaluate whether the models can correctly infer the time complexity of the algorithms. In this setting, performance is more modest; while several models demonstrate some ability to estimate complexity, the overall accuracy remains limited. As shown in Table 4, the highest accuracy observed is 0.69, achieved by GPT-5-mini, indicating that estimating algorithmic complexity remains a challenging task for current models. However, despite the relatively lower accuracy in predicting the exact complexity, we observe that resource hallucinations are very rare, suggesting that models maintain a reasonable understanding of the underlying resource requirements of the algorithm.

Model	Accuracy	Precision	Recall	F1
GPT-5	0.94	0.96	0.92	0.94
GPT-5-mini	0.96	0.96	0.96	0.96
Gemini-2.5-pro	0.94	0.96	0.92	0.94
Gemini-2.5-flash	0.89	0.88	0.9	0.89
Claude-sonnet-4.5	0.81	1.0	0.62	0.77
Llama-3.1-8b-instruct	0.56	0.54	0.86	0.66
Phi-4	0.8	0.77	0.86	0.81
Qwen2.5-coder-7b-instruct	0.69	0.62	0.98	0.76
Qwen2.5-coder-14b-instruct	0.5	0.5	1.0	0.67

Table 3: Results for pseudocode logical reasoning task.

C.2 Self-fixing Results

We discuss the overall hallucination rates across iterative prompting rounds along with the different types of hallucinations in a more granular manner in Figure 4 and 5. Figure 4 shows that although mapping and naming hallucinations decrease slightly across the five iterations, resource hallucinations remain highly unstable, fluctuating throughout rather than exhibiting a consistent downward trend. Logic hallucinations, on the other hand, display mixed behaviors: most models exhibit a gradual decrease, whereas both Qwen models show an increase in the rate of logic hallucination.

Model	Accuracy
GPT-5	0.68
GPT-5-mini	0.69
Gemini-2.5-pro	0.64
Gemini-2.5-flash	0.61
Claude-sonnet-4.5	0.47
Llama-3.1-8b-instruct	0.60
Phi-4	0.00
Qwen2.5-coder-7b-instruct	0.66
Qwen2.5-coder-14b-instruct	0.61

Table 4: Results for pseudocode time complexity task.

C.3 Bootstrapped Results

We report the bootstrapped results for models across tasks. Specifically, we generate 10,000 bootstrap samples by resampling the evaluation instances with replacement while preserving the original sample size. The hallucination rate for each hallucination type is recomputed on every bootstrap sample. Table 5 - 8 reports the mean, standard deviation, and 95% confidence intervals for each hallucination type.

D Hallucination Definition and Taxonomy Scope

Following Tian et al. (2025), we define code hallucination as any generated code that is syntactically plausible yet fails to execute correctly or satisfy

		Code Generation			
		Hallucination Rate (↓)			
Model	**Lang.**	**Map.**	**Name**	**Rsrc.**	**Logic**
Proprietary Models					
GPT-5	Python	1.41 ± 0.52 [0.40, 2.60]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	21.22 ± 1.83 [17.60, 24.80]
	Java	0.80 ± 0.40 [0.20, 1.60]	0.20 ± 0.20 [0.00, 0.60]	0.00 ± 0.00 [0.00, 0.00]	20.23 ± 1.82 [16.60, 24.00]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.20 ± 0.20 [0.00, 0.60]	0.00 ± 0.00 [0.00, 0.00]	20.59 ± 1.81 [17.20, 24.20]
GPT-5-mini	Python	1.20 ± 0.48 [0.40, 2.20]	0.20 ± 0.20 [0.00, 0.60]	0.00 ± 0.00 [0.00, 0.00]	27.97 ± 2.01 [24.00, 32.00]
	Java	1.40 ± 0.52 [0.40, 2.40]	0.20 ± 0.20 [0.00, 0.60]	0.20 ± 0.20 [0.00, 0.60]	23.79 ± 1.89 [20.00, 27.60]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.20 ± 0.20 [0.00, 0.60]	0.00 ± 0.00 [0.00, 0.00]	26.79 ± 1.99 [22.80, 30.60]
Gemini-2.5-Pro	Python	1.79 ± 0.59 [0.80, 3.00]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	22.19 ± 1.88 [18.60, 26.00]
	Java	1.60 ± 0.56 [0.60, 2.80]	0.00 ± 0.00 [0.00, 0.00]	0.40 ± 0.28 [0.00, 1.00]	23.43 ± 1.89 [19.80, 27.20]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.01 ± 0.44 [0.20, 2.00]	0.00 ± 0.00 [0.00, 0.00]	21.22 ± 1.82 [17.60, 25.00]
Gemini-2.5-Flash	Python	2.00 ± 0.63 [0.80, 3.40]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	30.20 ± 2.07 [26.20, 34.20]
	Java	0.80 ± 0.40 [0.20, 1.60]	0.60 ± 0.34 [0.00, 1.40]	0.40 ± 0.28 [0.00, 1.00]	21.65 ± 1.84 [18.00, 25.40]
	C++	0.00 ± 0.00 [0.00, 0.00]	3.20 ± 0.79 [1.80, 4.80]	0.00 ± 0.00 [0.00, 0.00]	24.63 ± 1.93 [21.00, 28.40]
Claude-Sonnet-4.5	Python	1.60 ± 0.57 [0.60, 2.80]	0.00 ± 0.00 [0.00, 0.00]	1.20 ± 0.49 [0.40, 2.20]	39.18 ± 2.17 [35.00, 43.40]
	Java	1.60 ± 0.57 [0.60, 2.80]	0.59 ± 0.34 [0.00, 1.40]	1.40 ± 0.52 [0.40, 2.60]	43.24 ± 2.23 [38.80, 47.60]
	C++	0.00 ± 0.00 [0.00, 0.00]	2.40 ± 0.68 [1.20, 3.80]	0.20 ± 0.20 [0.00, 0.60]	43.97 ± 2.25 [39.60, 48.40]
Open Source Models					
Llama-3.1-8B-Instruct	Python	18.59 ± 1.76 [15.20, 22.20]	1.40 ± 0.52 [0.40, 2.40]	1.60 ± 0.57 [0.60, 2.80]	72.98 ± 1.98 [69.20, 76.80]
	Java	15.78 ± 1.63 [12.60, 19.00]	1.00 ± 0.45 [0.20, 2.00]	2.01 ± 0.63 [0.80, 3.40]	65.20 ± 2.12 [61.00, 69.20]
	C++	0.20 ± 0.20 [0.00, 0.60]	7.39 ± 1.18 [5.20, 9.80]	0.80 ± 0.40 [0.20, 1.60]	73.60 ± 1.97 [69.80, 77.40]
DeepSeek-Coder-V2-Lite-Instruct	Python	7.60 ± 1.18 [5.40, 10.00]	2.18 ± 0.65 [1.00, 3.60]	0.40 ± 0.28 [0.00, 1.00]	64.38 ± 2.15 [60.20, 68.60]
	Java	4.80 ± 0.95 [3.00, 6.80]	2.19 ± 0.66 [1.00, 3.60]	0.59 ± 0.34 [0.00, 1.40]	54.61 ± 2.22 [50.20, 59.00]
	C++	0.00 ± 0.00 [0.00, 0.00]	6.00 ± 1.07 [4.00, 8.20]	0.00 ± 0.00 [0.00, 0.00]	59.41 ± 2.21 [55.20, 63.80]
Phi-4	Python	7.62 ± 1.19 [5.40, 10.00]	0.60 ± 0.35 [0.00, 1.40]	0.60 ± 0.35 [0.00, 1.40]	59.96 ± 2.20 [55.60, 64.20]
	Java	9.60 ± 1.33 [7.00, 12.20]	1.40 ± 0.53 [0.40, 2.60]	0.40 ± 0.28 [0.00, 1.00]	53.41 ± 2.22 [49.00, 57.80]
	C++	0.00 ± 0.00 [0.00, 0.00]	6.41 ± 1.09 [4.40, 8.60]	0.00 ± 0.00 [0.00, 0.00]	57.42 ± 2.22 [53.20, 61.80]
Qwen2.5-Coder-7B-Instruct	Python	12.23 ± 1.47 [9.40, 15.20]	1.60 ± 0.56 [0.60, 2.80]	0.80 ± 0.40 [0.20, 1.60]	76.42 ± 1.90 [72.60, 80.00]
	Java	8.39 ± 1.25 [6.00, 11.00]	2.41 ± 0.69 [1.20, 3.80]	0.61 ± 0.35 [0.00, 1.40]	74.62 ± 1.96 [70.60, 78.40]
	C++	0.00 ± 0.00 [0.00, 0.00]	7.39 ± 1.16 [5.20, 9.80]	0.00 ± 0.00 [0.00, 0.00]	71.40 ± 2.01 [67.40, 75.40]
Qwen2.5-Coder-14B-Instruct	Python	11.03 ± 1.40 [8.40, 13.80]	1.40 ± 0.53 [0.40, 2.60]	0.40 ± 0.28 [0.00, 1.00]	70.80 ± 2.02 [66.80, 74.80]
	Java	4.22 ± 0.89 [2.60, 6.00]	0.60 ± 0.35 [0.00, 1.40]	0.60 ± 0.35 [0.00, 1.40]	67.99 ± 2.08 [63.80, 72.00]
	C++	0.00 ± 0.00 [0.00, 0.00]	9.79 ± 1.33 [7.40, 12.40]	0.00 ± 0.00 [0.00, 0.00]	65.44 ± 2.14 [61.20, 69.60]

Table 5: Bootstrapped results for code generation task.

		Code Debugging			
		Hallucination Rate (↓)			
Model	**Lang.**	**Map.**	**Name**	**Rsrc.**	**Logic**
Proprietary Models					
GPT-5	Python	0.38 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	11.18 ± 1.94 [7.46, 14.93]
	Java	0.38 ± 0.38 [0.00, 1.12]	1.50 ± 0.74 [0.37, 2.99]	0.00 ± 0.00 [0.00, 0.00]	13.04 ± 2.08 [9.32, 17.16]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.11 ± 0.64 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	22.00 ± 2.52 [17.16, 26.87]
GPT-5-mini	Python	0.00 ± 0.00 [0.00, 0.00]	10.07 ± 1.82 [6.72, 13.81]	0.00 ± 0.00 [0.00, 0.00]	14.93 ± 2.19 [10.82, 19.40]
	Java	0.37 ± 0.37 [0.00, 1.12]	1.48 ± 0.74 [0.37, 2.99]	0.00 ± 0.00 [0.00, 0.00]	14.95 ± 2.19 [10.82, 19.40]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.11 ± 0.65 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	29.86 ± 2.79 [24.25, 35.45]
Gemini-2.5-Pro	Python	0.75 ± 0.52 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	13.40 ± 2.08 [9.70, 17.54]
	Java	0.75 ± 0.52 [0.00, 1.87]	1.49 ± 0.74 [0.37, 2.99]	0.37 ± 0.37 [0.00, 1.12]	12.68 ± 2.05 [8.58, 16.79]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.48 ± 0.74 [0.37, 2.99]	0.00 ± 0.00 [0.00, 0.00]	21.26 ± 2.50 [16.42, 26.12]
Gemini-2.5-Flash	Python	0.37 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	20.14 ± 2.47 [15.30, 25.00]
	Java	0.38 ± 0.37 [0.00, 1.12]	1.85 ± 0.82 [0.37, 3.73]	0.00 ± 0.00 [0.00, 0.00]	18.28 ± 2.35 [13.81, 22.76]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.12 ± 0.64 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	28.36 ± 2.79 [22.76, 33.96]
Claude-Sonnet-4.5	Python	0.74 ± 0.52 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	17.18 ± 2.31 [12.69, 21.64]
	Java	0.38 ± 0.37 [0.00, 1.12]	1.49 ± 0.73 [0.37, 2.99]	0.00 ± 0.00 [0.00, 0.00]	16.03 ± 2.24 [11.94, 20.52]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.13 ± 0.65 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	23.12 ± 2.61 [18.28, 28.36]
Open Source Models					
Llama-3.1-8B-Instruct	Python	5.20 ± 1.34 [2.61, 7.84]	0.37 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	39.18 ± 2.97 [33.21, 45.15]
	Java	1.87 ± 0.83 [0.37, 3.73]	2.23 ± 0.90 [0.75, 4.10]	0.38 ± 0.37 [0.00, 1.12]	31.30 ± 2.81 [25.75, 36.94]
	C++	0.00 ± 0.00 [0.00, 0.00]	3.72 ± 1.16 [1.49, 5.97]	0.00 ± 0.00 [0.00, 0.00]	41.45 ± 3.00 [35.45, 47.39]
DeepSeek-Coder-V2-Lite-Instruct	Python	8.57 ± 1.72 [5.22, 11.94]	2.99 ± 1.04 [1.12, 5.22]	0.74 ± 0.53 [0.00, 1.87]	29.10 ± 2.76 [23.88, 34.70]
	Java	4.11 ± 1.20 [1.87, 6.72]	3.35 ± 1.10 [1.49, 5.60]	0.00 ± 0.00 [0.00, 0.00]	28.74 ± 2.73 [23.51, 33.96]
	C++	0.00 ± 0.00 [0.00, 0.00]	4.48 ± 1.26 [2.24, 7.09]	0.00 ± 0.00 [0.00, 0.00]	35.09 ± 2.92 [29.48, 41.04]
Phi-4	Python	2.60 ± 0.97 [0.75, 4.48]	0.37 ± 0.37 [0.00, 1.12]	0.37 ± 0.37 [0.00, 1.12]	25.00 ± 2.64 [19.78, 30.22]
	Java	0.74 ± 0.52 [0.00, 1.87]	2.24 ± 0.90 [0.75, 4.10]	0.00 ± 0.00 [0.00, 0.00]	20.50 ± 2.44 [15.67, 25.37]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.74 ± 0.52 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	38.04 ± 2.96 [32.46, 44.03]
Qwen2.5-Coder-7B-Instruct	Python	4.85 ± 1.31 [2.61, 7.46]	1.11 ± 0.64 [0.00, 2.61]	0.38 ± 0.37 [0.00, 1.12]	32.40 ± 2.87 [26.87, 38.06]
	Java	1.87 ± 0.83 [0.37, 3.73]	2.25 ± 0.91 [0.75, 4.10]	0.00 ± 0.00 [0.00, 0.00]	26.50 ± 2.69 [21.27, 32.09]
	C++	0.00 ± 0.00 [0.00, 0.00]	2.24 ± 0.90 [0.75, 4.10]	0.00 ± 0.00 [0.00, 0.00]	37.68 ± 2.97 [31.72, 43.66]
Qwen2.5-Coder-14B-Instruct	Python	4.11 ± 1.21 [1.87, 6.72]	0.38 ± 0.37 [0.00, 1.12]	0.37 ± 0.37 [0.00, 1.12]	27.20 ± 2.75 [22.01, 32.84]
	Java	1.87 ± 0.82 [0.37, 3.73]	1.50 ± 0.75 [0.37, 2.99]	0.37 ± 0.37 [0.00, 1.12]	21.65 ± 2.52 [16.79, 26.87]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.74 ± 0.52 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	36.56 ± 2.95 [30.97, 42.54]

Table 6: Bootstrapped results for code debugging task.

		Code Refactoring			
		Hallucination Rate (↓)			
Model	**Lang.**	**Map.**	**Name**	**Rsrc.**	**Logic**
Proprietary Models					
GPT-5	Python	0.37 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	0.37 ± 0.37 [0.00, 1.12]	1.49 ± 0.75 [0.37, 2.99]
	Java	0.00 ± 0.00 [0.00, 0.00]	0.37 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	2.62 ± 0.99 [0.75, 4.85]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	14.52 ± 1.93 [10.65, 18.34]
GPT-5-mini	Python	0.75 ± 0.53 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	1.86 ± 0.82 [0.37, 3.73]
	Java	0.74 ± 0.52 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	3.36 ± 1.10 [1.49, 5.60]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	16.27 ± 1.99 [12.43, 20.41]
Gemini-2.5-Pro	Python	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	7.94 ± 1.65 [4.92, 11.36]
	Java	0.38 ± 0.38 [0.00, 1.14]	0.38 ± 0.38 [0.00, 1.14]	0.00 ± 0.00 [0.00, 0.00]	6.06 ± 1.46 [3.41, 9.09]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	19.26 ± 2.15 [15.06, 23.49]
Gemini-2.5-Flash	Python	1.89 ± 0.83 [0.38, 3.77]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	10.21 ± 1.86 [6.79, 13.96]
	Java	0.38 ± 0.37 [0.00, 1.13]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	10.52 ± 1.91 [6.79, 14.34]
	C++	0.00 ± 0.00 [0.00, 0.00]	0.91 ± 0.52 [0.00, 2.11]	0.00 ± 0.00 [0.00, 0.00]	21.48 ± 2.23 [17.22, 25.98]
Claude-Sonnet-4.5	Python	1.12 ± 0.64 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	0.00 ± 0.00 [0.00, 0.00]	8.22 ± 1.69 [5.22, 11.57]
	Java	0.00 ± 0.00 [0.00, 0.00]	0.37 ± 0.37 [0.00, 1.12]	0.00 ± 0.00 [0.00, 0.00]	7.11 ± 1.54 [4.10, 10.07]
	C++	0.00 ± 0.00 [0.00, 0.00]	1.18 ± 0.59 [0.30, 2.37]	0.00 ± 0.00 [0.00, 0.00]	21.59 ± 2.26 [17.16, 26.04]
Open Source Models					
Llama-3.1-8B-Instruct	Python	13.81 ± 2.10 [9.70, 17.91]	4.84 ± 1.30 [2.61, 7.46]	0.00 ± 0.00 [0.00, 0.00]	47.76 ± 3.09 [41.79, 53.73]
	Java	6.74 ± 1.53 [3.75, 10.11]	5.95 ± 1.45 [3.37, 8.99]	0.38 ± 0.38 [0.00, 1.12]	35.20 ± 2.94 [29.59, 41.20]
	C++	1.48 ± 0.66 [0.30, 2.97]	13.06 ± 1.81 [9.50, 16.62]	0.60 ± 0.42 [0.00, 1.48]	34.13 ± 2.59 [29.08, 39.17]
DeepSeek-Coder-V2-Lite-Instruct	Python	5.25 ± 1.37 [2.62, 7.87]	3.35 ± 1.11 [1.50, 5.62]	0.00 ± 0.00 [0.00, 0.00]	43.80 ± 3.02 [37.83, 49.81]
	Java	4.13 ± 1.20 [1.87, 6.74]	10.47 ± 1.86 [7.12, 14.23]	0.74 ± 0.53 [0.00, 1.87]	22.47 ± 2.58 [17.60, 27.72]
	C++	0.00 ± 0.00 [0.00, 0.00]	8.86 ± 1.53 [5.92, 12.13]	0.00 ± 0.00 [0.00, 0.00]	23.35 ± 2.33 [18.93, 28.11]
Phi-4	Python	3.00 ± 1.04 [1.12, 5.22]	1.12 ± 0.64 [0.00, 2.61]	0.00 ± 0.00 [0.00, 0.00]	15.30 ± 2.17 [11.19, 19.78]
	Java	1.12 ± 0.64 [0.00, 2.61]	0.37 ± 0.37 [0.00, 1.12]	0.37 ± 0.37 [0.00, 1.12]	13.45 ± 2.07 [9.70, 17.54]
	C++	0.30 ± 0.30 [0.00, 0.89]	2.67 ± 0.87 [1.18, 4.44]	0.59 ± 0.42 [0.00, 1.48]	16.31 ± 1.99 [12.43, 20.41]
Qwen2.5-Coder-7B-Instruct	Python	10.80 ± 1.89 [7.09, 14.55]	3.35 ± 1.10 [1.49, 5.60]	0.00 ± 0.00 [0.00, 0.00]	30.61 ± 2.84 [25.00, 36.19]
	Java	5.98 ± 1.44 [3.36, 8.96]	3.36 ± 1.11 [1.49, 5.60]	1.13 ± 0.65 [0.00, 2.61]	32.44 ± 2.86 [26.87, 38.06]
	C++	0.29 ± 0.29 [0.00, 0.89]	15.96 ± 2.00 [12.13, 20.12]	0.30 ± 0.29 [0.00, 0.89]	30.19 ± 2.48 [25.44, 35.21]
Qwen2.5-Coder-14B-Instruct	Python	2.31 ± 0.93 [0.76, 4.20]	1.91 ± 0.84 [0.38, 3.82]	0.00 ± 0.00 [0.00, 0.00]	16.80 ± 2.30 [12.60, 21.37]
	Java	1.12 ± 0.64 [0.00, 2.61]	0.76 ± 0.54 [0.00, 1.87]	0.00 ± 0.00 [0.00, 0.00]	10.45 ± 1.87 [7.09, 14.18]
	C++	0.30 ± 0.30 [0.00, 0.89]	3.55 ± 1.00 [1.78, 5.62]	0.00 ± 0.00 [0.00, 0.00]	19.22 ± 2.15 [15.09, 23.67]

Table 7: Bootstrapped results for code refactoring task.

		Code Editing			
		Hallucination Rate (↓)			
Model	**Lang.**	**Map.**	**Name**	**Rsrc.**	**Logic**
Proprietary Models					
GPT-5	Python	3.80 ± 1.52 [1.27, 6.96]	1.89 ± 1.08 [0.00, 4.43]	0.00 ± 0.00 [0.00, 0.00]	12.64 ± 2.63 [7.59, 18.35]
	Java	0.00 ± 0.00 [0.00, 0.00]	5.69 ± 1.81 [2.53, 9.49]	0.00 ± 0.00 [0.00, 0.00]	18.35 ± 3.13 [12.66, 24.68]
	C++	0.63 ± 0.62 [0.00, 1.90]	2.53 ± 1.25 [0.63, 5.06]	0.00 ± 0.00 [0.00, 0.00]	10.14 ± 2.39 [5.70, 15.19]
GPT-5-mini	Python	2.54 ± 1.25 [0.63, 5.06]	1.27 ± 0.89 [0.00, 3.16]	0.00 ± 0.00 [0.00, 0.00]	17.14 ± 3.00 [11.39, 23.42]
	Java	0.63 ± 0.63 [0.00, 1.90]	7.60 ± 2.12 [3.80, 12.03]	0.00 ± 0.00 [0.00, 0.00]	20.20 ± 3.16 [13.92, 26.58]
	C++	0.63 ± 0.62 [0.00, 1.90]	5.06 ± 1.75 [1.90, 8.86]	0.00 ± 0.00 [0.00, 0.00]	6.95 ± 2.02 [3.16, 11.39]
Gemini-2.5-Pro	Python	3.17 ± 1.40 [0.63, 6.33]	2.53 ± 1.25 [0.63, 5.06]	0.00 ± 0.00 [0.00, 0.00]	12.05 ± 2.61 [6.96, 17.09]
	Java	0.63 ± 0.63 [0.00, 1.90]	6.38 ± 1.95 [2.53, 10.13]	0.00 ± 0.00 [0.00, 0.00]	17.12 ± 2.97 [11.39, 23.42]
	C++	0.63 ± 0.63 [0.00, 1.90]	4.43 ± 1.64 [1.27, 7.59]	0.00 ± 0.00 [0.00, 0.00]	10.13 ± 2.41 [5.70, 15.19]
Gemini-2.5-Flash	Python	3.18 ± 1.40 [0.63, 6.33]	1.89 ± 1.09 [0.00, 4.43]	0.00 ± 0.00 [0.00, 0.00]	17.11 ± 2.95 [11.39, 23.42]
	Java	0.63 ± 0.63 [0.00, 1.90]	8.26 ± 2.19 [4.43, 12.66]	0.00 ± 0.00 [0.00, 0.00]	20.88 ± 3.24 [14.56, 27.22]
	C++	0.63 ± 0.63 [0.00, 1.90]	1.89 ± 1.07 [0.00, 4.43]	0.00 ± 0.00 [0.00, 0.00]	11.42 ± 2.52 [6.96, 16.46]
Claude-Sonnet-4.5	Python	6.31 ± 1.94 [3.16, 10.13]	0.63 ± 0.63 [0.00, 1.90]	0.00 ± 0.00 [0.00, 0.00]	16.45 ± 2.94 [10.76, 22.15]
	Java	0.64 ± 0.65 [0.00, 1.90]	5.05 ± 1.77 [1.90, 8.86]	0.00 ± 0.00 [0.00, 0.00]	22.13 ± 3.30 [15.82, 29.11]
	C++	0.62 ± 0.62 [0.00, 1.90]	3.14 ± 1.40 [0.63, 6.33]	0.00 ± 0.00 [0.00, 0.00]	12.00 ± 2.56 [6.96, 17.09]
Open Source Models					
Llama-3.1-8B-Instruct	Python	7.59 ± 2.11 [3.80, 12.03]	6.35 ± 1.93 [3.16, 10.13]	0.64 ± 0.63 [0.00, 1.90]	32.30 ± 3.75 [25.32, 39.87]
	Java	4.45 ± 1.64 [1.27, 7.59]	9.50 ± 2.35 [5.06, 14.56]	0.00 ± 0.00 [0.00, 0.00]	39.31 ± 3.92 [31.65, 46.84]
	C++	1.24 ± 0.89 [0.00, 3.16]	9.50 ± 2.34 [5.06, 13.92]	0.00 ± 0.00 [0.00, 0.00]	27.19 ± 3.55 [20.25, 34.18]
DeepSeek-Coder-V2-Lite-Instruct	Python	11.38 ± 2.54 [6.96, 16.46]	3.82 ± 1.52 [1.27, 6.96]	1.25 ± 0.89 [0.00, 3.16]	41.70 ± 3.97 [34.18, 49.38]
	Java	3.17 ± 1.40 [0.63, 6.33]	2.54 ± 1.25 [0.63, 5.06]	1.88 ± 1.09 [0.00, 4.43]	39.89 ± 3.92 [32.28, 47.47]
	C++	0.63 ± 0.63 [0.00, 1.90]	8.20 ± 2.16 [4.43, 12.66]	0.00 ± 0.00 [0.00, 0.00]	38.60 ± 3.86 [31.01, 46.20]
Phi-4	Python	6.32 ± 1.96 [2.53, 10.13]	2.54 ± 1.25 [0.63, 5.06]	0.00 ± 0.00 [0.00, 0.00]	25.35 ± 3.49 [18.99, 32.28]
	Java	0.63 ± 0.63 [0.00, 1.90]	8.24 ± 2.19 [4.43, 12.66]	0.00 ± 0.00 [0.00, 0.00]	27.83 ± 3.58 [20.89, 34.81]
	C++	1.26 ± 0.89 [0.00, 3.16]	10.10 ± 2.39 [5.70, 15.19]	0.00 ± 0.00 [0.00, 0.00]	20.24 ± 3.20 [13.92, 26.58]
Qwen2.5-Coder-7B-Instruct	Python	5.66 ± 1.84 [2.53, 9.49]	6.34 ± 1.96 [2.53, 10.13]	0.00 ± 0.00 [0.00, 0.00]	22.81 ± 3.28 [16.46, 29.11]
	Java	3.15 ± 1.38 [0.63, 6.33]	4.42 ± 1.64 [1.27, 7.59]	0.00 ± 0.00 [0.00, 0.00]	33.57 ± 3.72 [26.58, 41.14]
	C++	1.28 ± 0.89 [0.00, 3.16]	10.70 ± 2.46 [6.33, 15.82]	0.00 ± 0.00 [0.00, 0.00]	20.85 ± 3.21 [14.56, 27.22]
Qwen2.5-Coder-14B-Instruct	Python	5.07 ± 1.73 [1.90, 8.86]	3.78 ± 1.51 [1.27, 6.96]	0.00 ± 0.00 [0.00, 0.00]	22.15 ± 3.31 [15.82, 28.50]
	Java	2.53 ± 1.23 [0.63, 5.06]	6.92 ± 2.01 [3.16, 10.76]	0.00 ± 0.00 [0.00, 0.00]	25.31 ± 3.46 [18.99, 32.28]
	C++	1.26 ± 0.89 [0.00, 3.16]	6.34 ± 1.95 [2.53, 10.13]	0.00 ± 0.00 [0.00, 0.00]	22.09 ± 3.31 [15.82, 29.11]

Table 8: Bootstrapped results for code editing task.

problem constraints, a broader operational notion than factual fabrication (e.g., hallucinated APIs) that explicitly includes logic errors as a failure mode. The CodeHalu taxonomy is grounded in fundamental properties of code execution rather than task- or language-specific constructs, making it naturally extensible to debugging, refactoring, and editing tasks across Python, Java, and C++. This broader usage is consistent with related benchmarks such as CodeMirage (Agarwal et al., 2025), which similarly treat logical errors as a hallucination category.

E Calculation of Hallucination Rates

We use an execution-based verification pipeline to detect and label hallucinations and compute the Hallucination Rate (HR) based on the proportion of generated outputs that exhibit at least one hallucination according to the taxonomy described in Section 3.3.

E.1 Hallucination Detection and Labeling Process

The following steps are taken to detect and categorize hallucinations:

- Every generated code snippet is compiled and executed against a set of provided test cases. The execution is calculated using the `psutil` library. We log the compilation output, runtime tracebacks, standard outputs, and peak memory usage.
- If the code fails to compile or execute correctly, the resulting traceback, standard error (`stderr`), and exit codes are parsed using predefined heuristics. Regular expressions are used to map language-specific error patterns directly to specific hallucination types:
  - **Python:** `TypeError` and `IndexError` are mapped to Mapping Hallucinations. `NameError` and `ModuleNotFoundError` are mapped to Naming Hallucinations. `MemoryError` and `RecursionError` are mapped to Resource Hallucinations.
  - **C++:** Compilation errors like "Undeclared identifier" or "No such file" are mapped to Naming Hallucinations. Runtime crashes like "Segmentation fault" are mapped to Mapping Hallucinations.
  - **Java:** Compilation errors such as "cannot find symbol" or "package does not exist" are mapped to Naming Hallucinations. Exceptions like `NullPointerException` and `ArrayIndexOutOfBoundsException` are mapped to Mapping Hallucinations.
- If the code compiles and executes successfully without system-level errors or resource exhaustion, but produces an actual output that does not match the expected test case output, it is automatically classified as a Logic Hallucination. This indicates a fundamental flaw in the algorithmic reasoning or problem constraint adherence.

E.2 Computation of Hallucination Rate (HR)

Given a dataset D containing N generated samples, the overall hallucination rate is computed as the average of these labels:

$$HR = \frac{1}{N} \sum_{i=1}^{N} h_i \tag{13}$$

In addition to the overall rate, we compute category-specific hallucination rates for each hallucination type in the taxonomy: mapping, naming, resource, and logic. For a hallucination category $c \in \{M, N, R, L\}$, the corresponding hallucination rate is defined as:

$$p_c = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[\text{hallucination}_c(i)] \tag{14}$$

F Code Quality Metrics and Complexity Calculation

Depending on the target language, we utilize the following tools to calculate the code quality metrics:

- **Python:** We utilize the `radon` library to compute Cyclomatic complexity and the `cognitive-complexity` package (which adheres to SonarQube guidelines) for Cognitive complexity.
- **C++ and Java:** Due to the unavailability of standardized, readily available packages for measuring these specific complexities in C++ and Java, we implement custom AST traversal using `tree-sitter` alongside its respective language bindings (`tree-sitter-cpp` and `tree-sitter-java`). This ensures that our analysis remains consistent and directly comparable across languages.

F.1 Cyclomatic Complexity Calculation

Cyclomatic complexity measures the number of linearly independent paths through a program's source code. For C++ and Java, our `tree-sitter` implementation calculates this by assigning a base score of 1 to every function and incrementing by 1 for each of the following control-flow nodes:

- Decision statements: `if`, `while`, `for`, `do-while`, and `catch` blocks.
- Branching operators: Ternary operators (`? :`) and switch cases/labels.
- Boolean logic: Logical AND (`&&`) and logical OR (`||`) operators within binary expressions.

The resulting score has a theoretical lower bound of 1, representing a program with a single execution path and no branching. To aid interpretation, we follow the standard risk tiers proposed by McCabe (McCabe, 1976): scores of 1–10 indicate low structural risk, 11–20 indicate moderate complexity, 21–50 indicate high complexity, and values above 50 indicate code that is extremely difficult to test and maintain.

F.2 Cognitive Complexity Calculation

Cognitive complexity evaluates the understandability of the code by heavily penalizing structural nesting, closely following the SonarQube specification. Our AST traversal algorithm calculates this score (lower bound of 0) based on three primary increments:

- **Basic Control Flow:** The score increments by 1 for statements that break linear execution flow, including `if`, `while`, `for`, `catch`, and ternary operators. Sequences of boolean operators (e.g., alternating `&&` and `||`) also increment the score.
- **Nesting Penalty:** Unlike Cyclomatic complexity, Cognitive complexity increments the score further for every level of structural nesting. For example, a `for` loop nested inside an `if` statement receives a higher penalty than the same structures placed sequentially. (Note: `else if` chains are treated as linear branching and do not incur additional nesting penalties).
- **Jumps and Recursion:** The score increments for recursive function calls and labeled jumps (e.g., `goto` statements in C++).

To aid interpretation, we adopt the SonarQube thresholds as a reference; scores below 15 are considered acceptable, 15–30 indicate elevated cognitive load, and scores above 30 suggest significantly difficult code to understand.

G CRI

G.1 CRI Algorithm

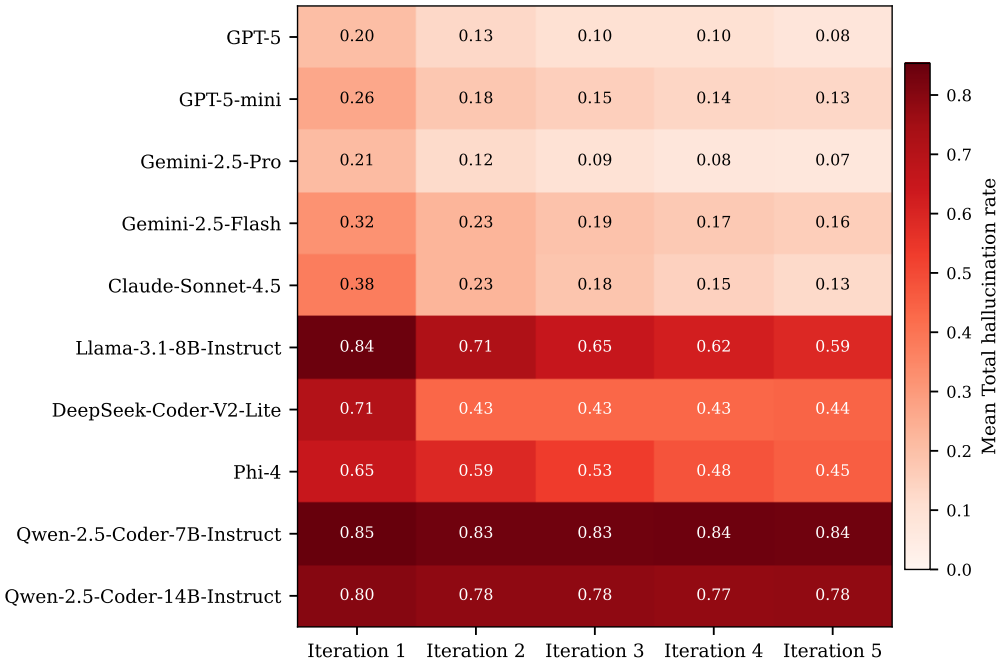
Algorithm 1 Computation of the Code Reliability Index (CRI)

Require: Dataset D, test cases, γ
Ensure: Dataset-level CRI_{100}

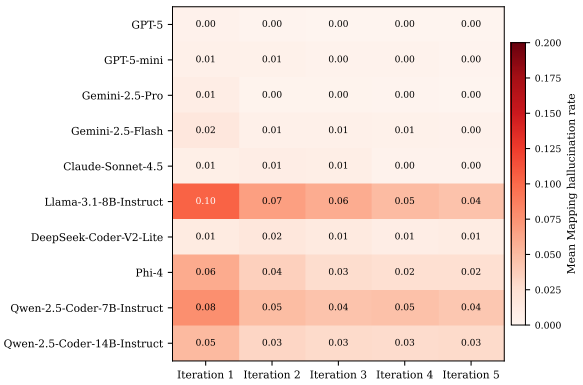
$B \leftarrow 0, \quad Q_{\text{sum}} \leftarrow 0, \quad N_{\text{exec}} \leftarrow 0$
for all sample $i \in D$ **do**
  $\text{blocking}_i \leftarrow \mathbf{1}[\text{mapping hallucination}] \vee \mathbf{1}[\text{naming hallucination}]$
  if $\text{blocking}_i = 1$ **then**
    $B \leftarrow B + 1$
    continue
  end if
  Execute the generated program
  Compute TCPR_i
  $f_i \leftarrow 1 - \mathrm{TCPR}_i$
  Compute normalized cyclomatic complexity c_i^{cyc}
  Compute normalized cognitive complexity c_i^{cog}
  $\tilde{C}_i \leftarrow \left(\frac{(c_i^{\text{cyc}})^4 + (c_i^{\text{cog}})^4}{2} \right)^{1/4}$
  Compute normalized runtime t_i and memory usage m_i
  $S_i \leftarrow \frac{t_i + m_i}{2}$
  $\tilde{U}_i \leftarrow \left(\frac{\tilde{C}_i^4 + S_i^4}{2} \right)^{1/4}$
  $Q_i \leftarrow f_i + (1 - f_i)\gamma \tilde{U}_i$
  $Q_{\text{sum}} \leftarrow Q_{\text{sum}} + Q_i$
  $N_{\text{exec}} \leftarrow N_{\text{exec}} + 1$
end for
$p_B \leftarrow \frac{B}{|D|}$
if $N_{\text{exec}} > 0$ **then**
  $\bar{Q}_{\text{exec}} \leftarrow \frac{Q_{\text{sum}}}{N_{\text{exec}}}$
else
  $\bar{Q}_{\text{exec}} \leftarrow 0$
end if
$\mathrm{CRI} \leftarrow p_B + (1 - p_B)\bar{Q}_{\text{exec}}$
return $\mathrm{CRI}_{100} = 100 \times \mathrm{CRI}$

G.2 Robustness to the Choice of γ

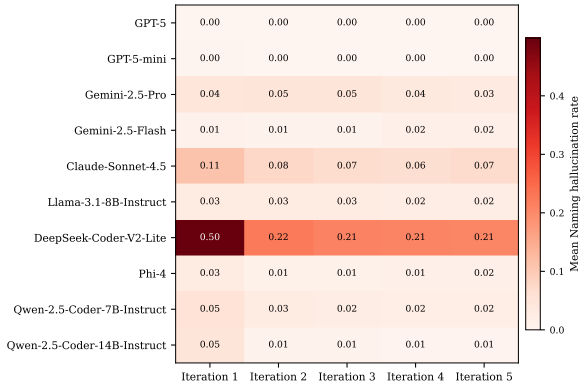
Since γ determines the weight assigned to implementation quality in CRI, we evaluate the metric under multiple values of $\gamma \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. As shown in Figure 6, the resulting model rankings remain highly consistent across all settings, indicating that CRI provides a stable assessment of model reliability and is not overly sensitive to the choice of γ.



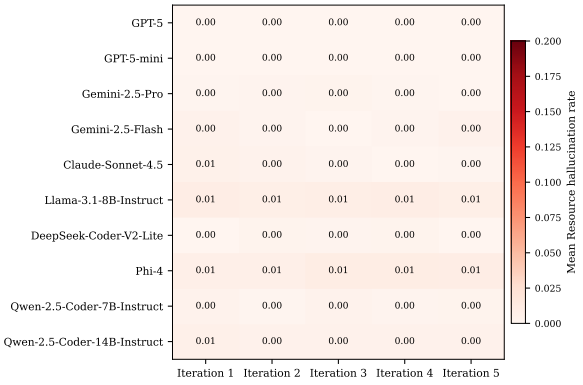
(a) Total Hallucination rates



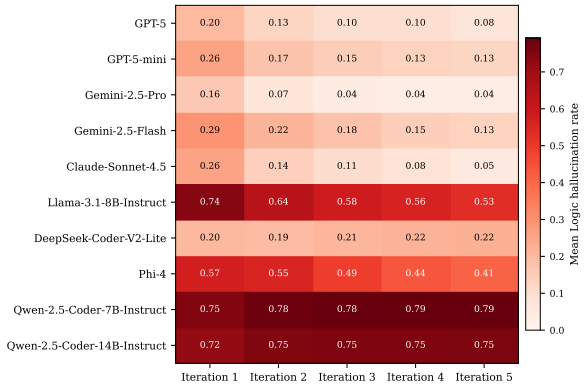
(b) Mapping Hallucination



(c) Naming Hallucination

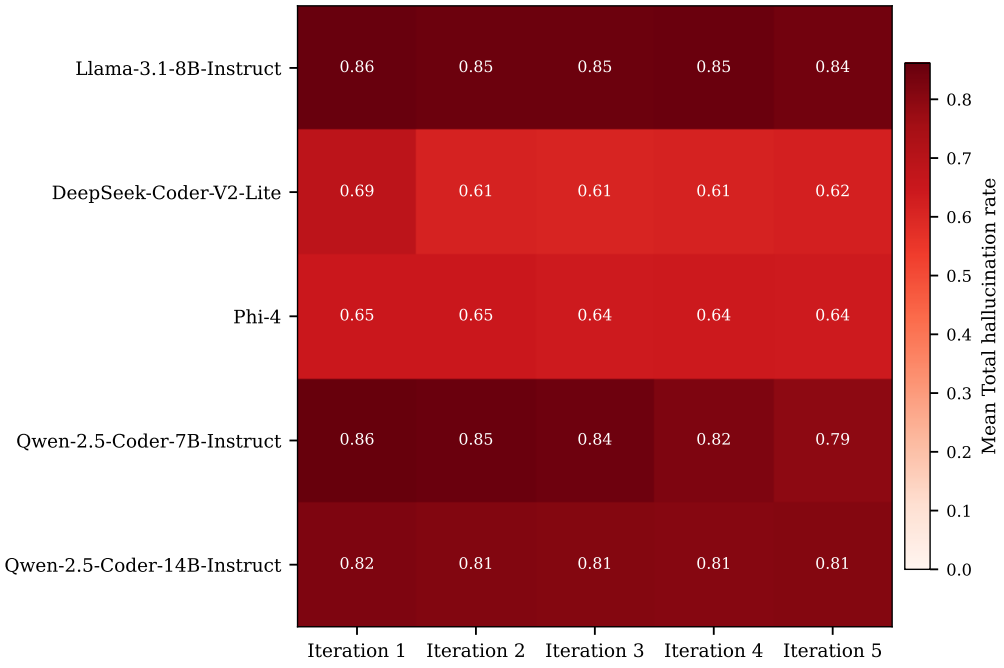


(d) Resource Hallucination

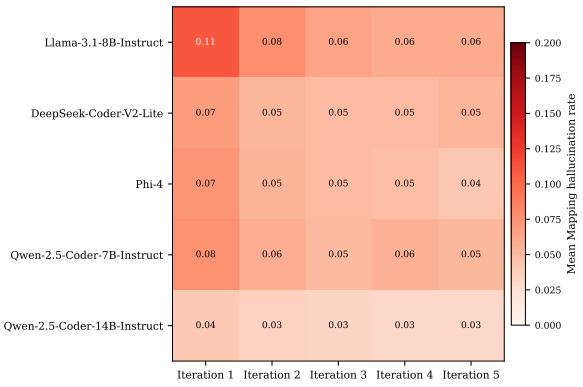


(e) Logic Hallucination

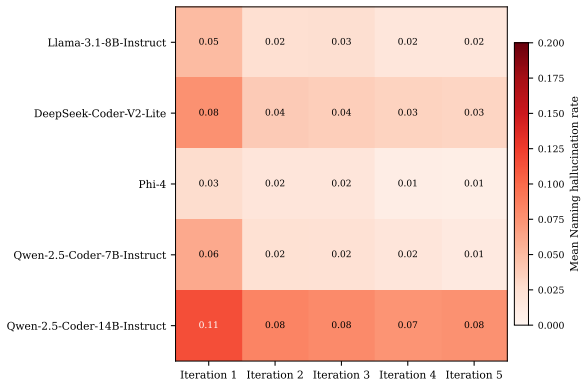
Figure 4: Hallucination rates across five self-improvement iterations. At each iteration, the model is provided with the code generated in the previous iteration, the original requirements, and any traceback errors. (a) shows the percentage of datapoints containing at least one hallucination, while (b) - (e) report the rates of individual hallucination types across iterations.



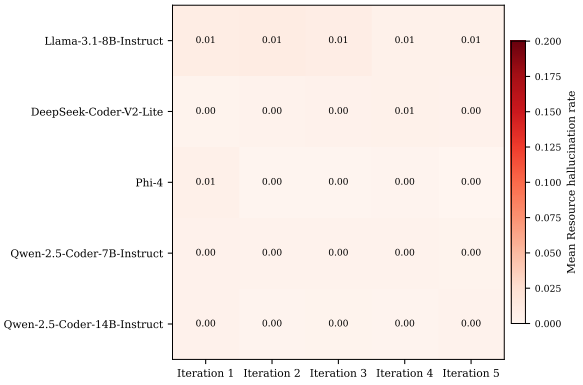
(a) Total Hallucination rates



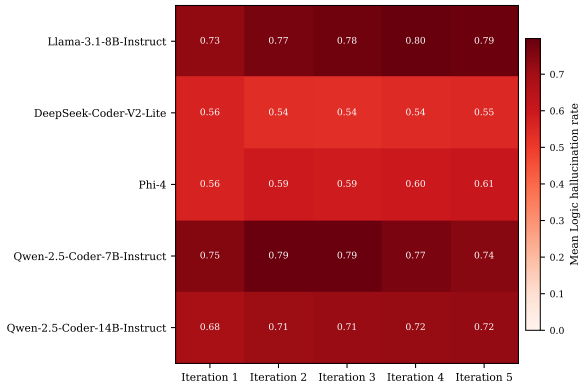
(b) Mapping Hallucination



(c) Naming Hallucination

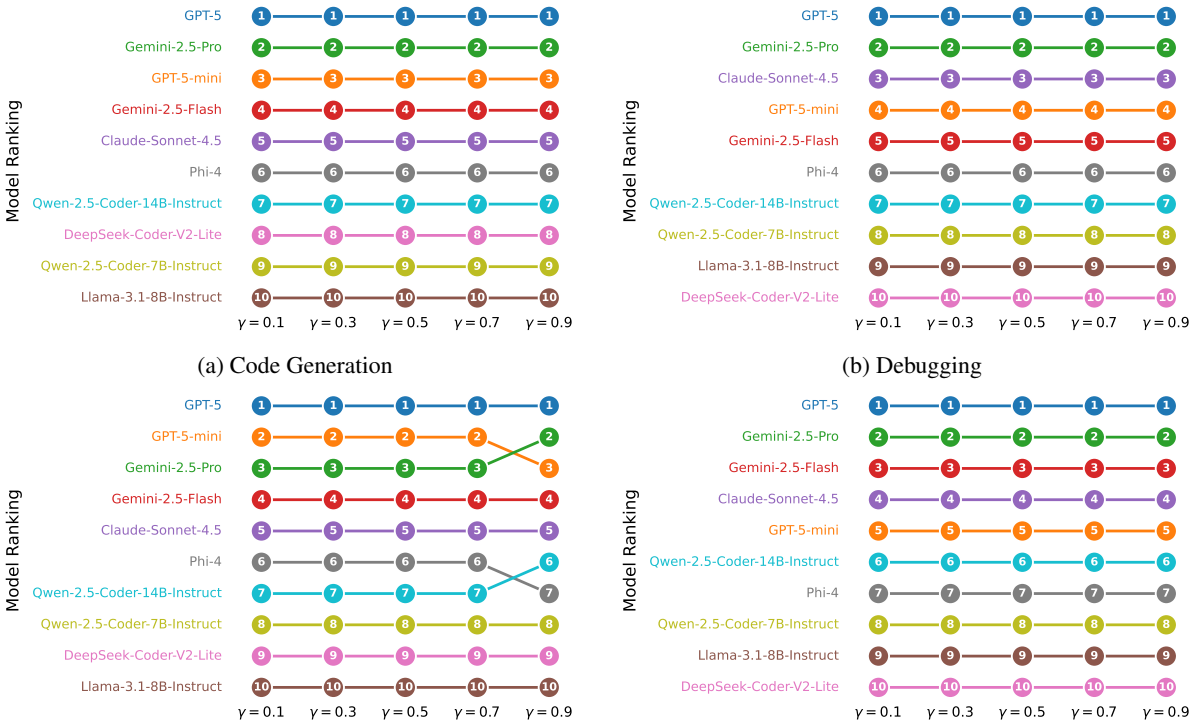


(d) Resource Hallucination



(e) Logic Hallucination

Figure 5: Hallucination rates across five self-improvement iterations using Chain-of-Thought. At each iteration, the model is provided with the code generated in the previous iteration, the original requirements, and any traceback errors. (a) shows the percentage of datapoints containing at least one hallucination, while (b) - (e) report the rates of individual hallucination types across iterations.



(a) Code Generation

(b) Debugging

(c) Refactoring

(d) Editing

Figure 6: Model rankings based on the Code Reliability Index (CRI) for different values of γ. The rankings remain largely unchanged across all settings, demonstrating that CRI is robust to the choice of γ.

H Examples of Coding Tasks

This section provides representative examples of coding task data points from the dataset, illustrating the different task types considered in our evaluation.

Example for Code Generation

Task: Code Generation
Input: The problem statement given in natural language.
Output: Generated code that compiles and passes all test cases.
Example:
Problem Statement: The online retailer Amagoogsoftbook currently offers N different so-called "home assistants", which it wants to recommend to its customers. For this recommendation, they wish to rank all the assistants. The quality of this ranking is not very important – multiple assistants may even be assigned the same rank – but they wish to maximize the number of distinct ranks assigned, to lessen the decision fatigue of their customers.

To ensure that the ranking is not completely arbitrary, they have collected for each assistant i two measurements a_i and b_i – the quality of the jokes the assistant can tell and how nice are the compliments the assistant is able to give (clearly these are the two most important aspects). These measurements are of course somewhat subjective, so we wish to ignore small differences in them. However, if for two given assistants i and j we have that $a_i + K < a_j$ or $b_i + K < b_j$, the ranking of assistant j must be the same or higher than the ranking of assistant i. This rule may force two products to be given the same ranking, for example if an assistant i gives much better puns than assistant j, while assistant j gives the superior self-esteem boosts.

What is the maximum number of distinct ranks, taken over all possible rankings?

——Input——
The first line contains the integers $1 \leq N \leq 100000$ and $0 \leq K \leq 10^9$ – the number of assistants and the measurement difference limit as described in the statement. The next line contains the N integers $a_1, a_2, \ldots, a_N$. The next line contains the N integers $b_1, b_2, \ldots, b_N$.

All measurements are between 0 and 10^9.

——Output——
Output a single integer: the maximum number of distinct ranks.

——Examples——
Sample Input 1:
2 10
1 12
1 13
Sample Output 1:
2

Sample Input 2:
2 10
1 5
1 12
Sample Output 2:
2

Generated Code:

```
import sys

def main():
    data = list(map(int, sys.stdin.buffer.read().split()))
    it = iter(data)
    N = next(it)
    K = next(it)
    a = [next(it) for _ in range(N)]
    b = [next(it) for _ in range(N)]

    arrA = sorted(((a[i], i) for i in range(N)))
    arrB = sorted(((b[i], i) for i in range(N)))
    valA = [x[0] for x in arrA]
    idxA = [x[1] for x in arrA]
    valB = [x[0] for x in arrB]
    idxB = [x[1] for x in arrB]

    # First pass: DFS on original graph to compute finishing order
    visited = [False] * N
    order = []
    ra = N - 1
    rb = N - 1
        .
        .
        .
if __name__ == "__main__":
    main()
```

Example for Debugging

Task: Code Debugging
Input: The faulty program and the specific bug/error.
Output: Corrected code that compiles and passes all test cases.
Example:

Incorrect Code:

```
ans = 0

t = []
x = input()
y = int(input())
for i in range(y):
    z = input()
    t.append(z)

pt = -1
ln = len(x)
for i in t:
    a = i[0]
```

```
    in = i[1]
    pt = 0
    for j in range(ln):
        .
        .
        .
print(ans)
```

Error: reference error: illegal keywords
Debugged Code:

```
ans = 0

t = []
x = input().strip()
y = int(input().strip())
for i in range(y):
    z = input().strip()
    t.append(z)

pt = -1
ln = len(x)
for i in t:
    a = i[0]
    b = i[1]
    pt = 0
    for j in range(ln):
        .
        .
        .
print(ans)
```

Example for Refactoring

Task: Code Refactoring
Input: An existing source code needed to be refactored.
Output: Refactored code that compiles and passes all test cases.
Example:

Original Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  int n, f, sum = 0;
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> f;
    sum += f;
  }
  int count = 0, demo = sum;
```

```
  for (int j = 1; j <= 5; j++) {
    demo++;
    if (demo % (n + 1) != 1)
      count++;
  }
  cout << count;
  return 0;
}
```

Refactored Code:

```
#include <iostream>
#include <vector>

int calculateCount(int totalSum, int numElements) {
    int count = 0;
    int demo = totalSum;
    for (int j = 1; j <= 5; ++j) {
        demo++;
        if (demo % (numElements + 1)!= 1) {
            count++;
        }
    }
    return count;
}

int main() {
    int numElements;
    std::cin >> numElements;
    int totalSum = 0;
    for (int i = 0; i < numElements; ++i) {
        int element;
        std::cin >> element;
        totalSum += element;
    }
    int result = calculateCount(totalSum, numElements);
    std::cout << result;
    return 0;
}
```

Example for Editing

Task: Code Editing
Input: The source code and the instructions for the edits to be made.
Output: Edited code that compiles and passes all test cases.
Example:

Original Code:

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
```

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Instructions: Change the function so that it checks if all numbers are within the threshold distance with a tolerance of 1e-5 from the single number before them.
Edited Code:

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    for idx in range(1, len(numbers)):
        distance = abs(numbers[idx] - numbers[idx - 1])
        if distance < threshold + 1e-5:
            return True

    return False
```

Example of Generated Pseudocode

```
BEGIN FUNCTION are_anagrams WITH INPUT str1, str2
    SET clean_str1 TO lowercase of str1 with all whitespace removed
    SET clean_str2 TO lowercase of str2 with all whitespace removed
    IF length of clean_str1 NOT EQUAL TO length of clean_str2
        RETURN FALSE
    END IF
    INITIALIZE char_count AS empty mapping
    FOR EACH char IN clean_str1 DO
        IF char exists in char_count
            SET char_count AT char TO char_count AT char PLUS 1
        ELSE
            SET char_count AT char TO 1
        END IF
    END FOR
    FOR EACH char IN clean_str2 DO
        IF char does not exist in char_count OR char_count AT char EQUALS 0
            RETURN FALSE
        END IF
        SET char_count AT char TO char_count AT char MINUS 1
    END FOR
    RETURN TRUE
END FUNCTION
```

I Examples of Hallucinated Outputs

This section provides representative examples of hallucinated outputs for each of the four hallucination types.

Example for Mapping Hallucination

Hallucination Type: Mapping Hallucination
Hallucinated Output:

```
def calculate_max_score(frequency_list):
    max_score = 0
    current_max = 0

    for i, frequency in enumerate(frequency_list, start=1):
        current_max = max(i * frequency + current_max, current_max)
        max_score += current_max

    return max_score


def process_input(input_list):
    frequency_list = [0] * 100000

    for item in input_list:
        frequency_list[int(item) - 1] += 1

    result = []
    current_list = []

    for frequency in frequency_list:
        if frequency != 0:
            current_list.append(frequency)
        elif current_list:
            result.append(current_list)
            current_list = []

    if current_list:
        result.append(current_list)

    return result


def main():
    n = int(input())
    input_list = input().split()

    frequency_list = process_input(input_list)
    max_score = calculate_max_score(frequency_list)

    print(max_score)


if __name__ == "__main__":
    main()
```

Hallucination Description: This generated program produces a `TypeError` because the function `calculate_max_score` expects a list of integer frequencies, but it is given the output of `process_input`, which is a list of lists. When iterating over `frequency_list`, the variable `frequency` therefore becomes a list instead of an integer. The expression `i * frequency + current_max` consequently attempts to add an integer to a list, causing the runtime error `can only concatenate list (not "int") to list`.

Example for Naming Hallucination

Hallucination Type: Naming Hallucination
Hallucinated Output:

```
import java.util.*;

public class Solution {
    public static List<Integer> xor(List<Integer> l1, List<Integer> l2) {
        Set<Integer> ret = new HashSet<>();

        for (Integer e1 : l1) {
            boolean found = false;
            for (Integer e2 : l2) {
                if (e1.equals(e2)) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                ret.add(e1);
            }
        }

        for (Integer e2 : l2) {
            boolean found = false;
            for (Integer e1 : l1) {
                if (e2.equals(e1)) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                ret.add(e2);
            }
        }

        List<Integer> result = new ArrayList<>(ret);
        Collections.sort(result);
        return result;
    }
}
```

Hallucination Description: The LLM defined a method `xor(List<Integer>, List<Integer>)` that computes the symmetric difference of two lists. However, the original program expects a

method named `common(List<Integer>, List<Integer>)` and invokes it in multiple assertions. In this code editing task, the LLM is expected to modify only the faulty parts of the implementation, and for this datapoint, the function name is not among the elements requiring modification. By changing the method name to `xor`, the generated code no longer provides the expected `common` method, causing the compiler to raise a `cannot find symbol` error.

Example for Resource Hallucination

Hallucination Type: Resource Hallucination
Hallucinated Output:

```
def print_column(x1, y1):
    global last
    print(x1, y1)
    a = y1 - 1
    k = 0
    while a > 0:
        print(x1, a)
        a -= 1
        k = a + 1
    a = y1 + 1
    while a <= y1:
        print(x1, a)
        a += 1
        k = a - 1
    last = k
    return last


def print_marixup(x, y):
    if 1 <= x <= n:
        j = print_column(x, y)
        print_marixup(x, j)


def print_matrixdwn(x, y):
    if 1 <= x <= n:
        j = print_column(x, y)
        print_matrixdwn(x + 1, j)


n, m, x, y = map(int, input().split())
sums = 0
last = x

print_marixup(x, y)
print_matrixdwn(x + 1, last)
```

Hallucination Description: The buggy code has two flaws: the second `while` loop uses `a <= y1` (always `False` since `a = y1 + 1`), making it dead code, and `print_marixup` recurses with the same `x` forever. The correct version uses `a <= m` as the loop bound and decrements `x - 1` to properly traverse upward.

Example for Logic Hallucination

Hallucination Type: Logic Hallucination
Hallucinated Output:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

struct Gem {
    long long x, y;
};

bool canReach(const Gem& from, const Gem& to, long long r) {
    long long dx = abs(to.x - from.x);
    long long dy = to.y - from.y;
    if (dy <= 0) return false;
    return dx * r <= dy;
}

int main() {
    int n, r;
    long long w, h;
    cin >> n >> r >> w >> h;

    vector<Gem> gems(n);
    for (int i = 0; i < n; i++) {
        cin >> gems[i].x >> gems[i].y;
    }

    sort(gems.begin(), gems.end(), [](const Gem& a, const Gem& b) {
        return a.y < b.y;
    });

    vector<int> dp(n, 0);
    int maxGems = 0;

    for (int i = 0; i < n; i++) {
        long long dx = min(gems[i].x, w - gems[i].x);
        long long dy = gems[i].y;
        if (dx * r <= dy) {
            dp[i] = 1;
        }

        for (int j = 0; j < i; j++) {
            if (canReach(gems[j], gems[i], r)) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
```

```
        maxGems = max(maxGems, dp[i]);
    }

    cout << maxGems << endl;

    return 0;
}
```

Hallucination Description: The model initialized `dp(n, 0)` instead of `dp(n, 1)`. Since every gem is always reachable from the start (the player can choose any starting x position freely), each gem should begin with a minimum chain length of 1 (itself), but the model hallucinated a reachability constraint, defaulting unreachable gems to 0 and incorrectly excluding them from valid chains.

J Prompts

Prompt for Code Generation

You are an expert {language} programmer. Write a solution for the following problem. Output only the executable code. Do not include any explanations, markdown formatting, or comments.

{QUESTION:}

Write a {language} program for the above problem. Output only the code, with no explanations or comments.

Prompt for Code Debugging

You are an expert at debugging {language} code.
The following code is incorrect and contains the following type(s) of error(s): {bug_type}. Analyze it, fix all the bugs, and provide the corrected version.

Incorrect Code:
“‘{language}
{incorrect_solution}
“‘

Provide only the corrected and complete Python code inside a markdown block. Do not provide any explanation.

Prompt for Code Refactoring

You are an expert code refactoring assistant. Your task is to refactor the following {language} code to improve its quality, extensibility, maintainability, and readability.

{code_section}

Please refactor the code following these principles:
1. Improve code quality, extensibility, maintainability, and readability
2. Remove all code smells
3. Ensure that the refactored code maintains the same functionality

4. Add meaningful variable/function names if needed
5. Remove code duplication
6. Do NOT modify the input/output format - the code must read input and produce output exactly as the original
7. Do NOT create new files - put all refactored code in the SAME files they originally came from
8. Do NOT rename files - use the EXACT same filenames as the input

Output ONLY the refactored code. Do NOT include any explanations, summaries, descriptions of changes, or commentary before or after the code.

Format your response EXACTLY as:
FILENAME: <filename>
“‘{language}
<refactored code>
“‘

Make sure to include all files, even if some don’t require changes.

Prompt for Code Editing

Your task is to edit the {language} code based on the provided instruction.
Do not add any explanations, comments, or extra text. Only output the edited {language} code without any explanations.

Instruction:
{instruction}

Input Code:
“‘{language_md_prefix}
{input_code}
“‘

Your {language} Code:

Prompt for Self-fixing

Initial Attempt

```
You are an expert programmer. You will be given a coding problem.
Output only the code, with no explanations or comments.

QUESTION:
{question_text}

Write a {language} program for the above problem.
```

Feedback When Test Cases Fail

```
The code passed {passed}/{total} test cases.
```

```
Errors encountered:

Test Cases {case_numbers}:
{error_message}

Please fix these errors and output the corrected code.
```

Feedback When All Test Cases Pass

```
No errors. All test cases passed. Please output the code.
```

Prompt for Self-fixing (Chain-of-Thought)

Initial Attempt

```
You are an expert programmer. You will be given a coding problem.
Output only the code, with no explanations or comments.

QUESTION:
{question_text}

Write a {language} program for the above problem.
Think step-by-step before providing the code.
First analyze the requirements and potential edge cases step-by-step.
After your reasoning, output the final solution in a markdown code block.
Do not include any text outside the code block for the final solution.
```

Feedback When Test Cases Fail

```
The code passed {passed}/{total} test cases.

Errors encountered:

Test Cases {case_numbers}:
{error_message}

Please fix these errors and output the corrected code.
Think step-by-step to correct the logic and provide a new, fixed code block.
```

Feedback When All Test Cases Pass

```
No errors. All test cases passed. Please output the code.
```

Prompt for Pseudocode Translation

You are given a code snippet. Convert it into pseudocode using the EXACT formatting rules below.

REQUIREMENTS:
1. The pseudocode must be completely language-agnostic (no Python/Java/C++ syntax).
2. Do NOT use language-specific keywords (e.g., def, public, printf, ->, etc.).
3. Focus on the algorithmic logic and control flow.
4. Use clear indentation.

5. Use descriptive step names where helpful
6. Represent loops, conditionals, and functions in generic pseudocode style.
7. Do not include explanations — output only pseudocode.
8. Preserve the logical structure and order of operations exactly as in the code.

STRICT FORMAT RULES (MANDATORY):

1. Every function MUST start with:
BEGIN FUNCTION <FunctionName> WITH INPUT <comma separated inputs>
and end with:
END FUNCTION

2. Use ONLY the following control keywords in ALL CAPS:
BEGIN FUNCTION
END FUNCTION
CALL FUNCTION
SET
INITIALIZE
IF
ELSE
END IF
FOR EACH
END FOR
WHILE
END WHILE
RETURN
PRINT

3. All variable assignments must use:
SET variable TO expression

4. All initializations must use:
INITIALIZE variable AS description

5. Boolean comparisons must use:
IS TRUE
IS FALSE
EQUALS
NOT EQUAL TO
GREATER THAN
LESS THAN
GREATER THAN OR EQUAL TO
LESS THAN OR EQUAL TO

6. Logical composition must use:
AND
OR
NOT
XOR

7. Element access must be written as:
element AT index

8. Literal values must use:
TRUE
FALSE
EMPTY
NULL

9. Mathematical operations must be written in words:
PLUS
MINUS
MULTIPLIED BY
DIVIDED BY
MODULO
MINIMUM OF
MAXIMUM OF
ABSOLUTE VALUE OF

10. Loops must follow exactly:
WHILE condition DO
...
END WHILE

FOR EACH element IN collection DO
...
END FOR

11. Indentation must be exactly 4 spaces per level.

12. Function calls MUST follow exactly one of the formats below:

- Using a return value:
SET resultVariable TO CALL FUNCTION <FunctionName> WITH INPUT <comma separated inputs>

- Ignoring the return value:
CALL FUNCTION <FunctionName> WITH INPUT <comma separated inputs>

13. Do NOT use:
- Language-specific syntax
- Symbols like =, ==, <=, >=, &&, ||, ++, –, ->, etc.
- Explanations
- Comments
- Extra blank lines inside blocks

Here is the code:
{code}

Prompt for Pseudocode Task – Time Complexity

You are given an algorithm written in pseudocode along with a description of its input.

Your task is to determine the asymptotic time complexity of the algorithm with respect to the input size.

Instructions:
1. Carefully analyze the pseudocode and identify the dominant operations.
2. Consider loops, nested loops, recursion, and any operations whose cost depends on the input size.
3. Ignore constant factors and lower-order terms.
4. Express the final answer using Big-O notation.

Return your answer in the following format:

Time Complexity: <Big-O expression>
Explanation: <Brief explanation of why this is the correct complexity>

Input Description:
{input_description}

Pseudocode:
{pseudocode}

Prompt for Pseudocode Task – Logical Reasoning

You are given an algorithm written in pseudocode.

Your task is to determine whether the pseudocode is logically correct and well-defined.

An algorithm is considered incorrect only if it contains:
* logical errors that produce wrong results for valid inputs,
* incorrect loop conditions that cause infinite loops or incorrect termination,
* undefined variables or operations that make the algorithm unexecutable,
* missing steps that prevent the algorithm from functioning correctly.

Do NOT mark an algorithm incorrect for:
* unstated but standard preconditions (e.g., sorted input for binary/interpolation search),
* handling only a limited input domain (e.g., single-digit operands, non-negative integers),
* implicit conventions common in pseudocode (e.g., shared object state in OOP-style pseudocode, 0-based indexing),
* minor ambiguities that do not affect correctness for the intended input.

Output Format:
Answer: <Yes or No>
Explanation: <Brief explanation of the reasoning>

Pseudocode:
{pseudocode}