Refining Critical Thinking in LLM Code Generation: A Faulty Premises-Based Evaluation Framework

Anonymous ACL submission

Abstract

With the advancement of code generation capabilities in large language models (LLMs), their reliance on input premises has intensified. When users provide inputs containing faulty premises, the probability of code generation hallucinations rises significantly, exposing deficiencies in their self-scrutiny capabilities. This paper proposes Faulty Premises Bench (FPBench), the first code generation evaluation framework specifically targeting faulty premises. By systematically constructing three categories of faulty premises and integrating multi-dimensional evaluation metrics, we conduct in-depth assessments of 15 representative LLMs. Our key findings are: (1) Most models exhibit poor reasoning abilities and suboptimal code generation under faulty premises, heavily relying on explicit prompts for error detection; (2) Faulty premises trigger a point of diminishing returns in resource investment—blindly increasing response length fails to enhance quality; (3) The three types of faulty premises activate distinct defect patterns in models, revealing a triple dissociation in their cognitive mechanisms. This study highlights the need for LLMs to proactively verify premises in code generation, and through FPBench provides a theoretical foundation and practical pathway for developing reliable, human-centric code generation models.

1 Introduction

Large language models (LLMs) such as GPT-4 (Hurst et al., 2024) and the DeepSeek series (Liu et al., 2024) have significantly advanced code-related tasks, including generation, repair (Fan et al., 2023; Jiang et al., 2023; Xia et al., 2023), testing (Deng et al., 2023; Chen et al., 2024; Banescu et al., 2024), and test case generation (Steenhoek et al., 2025; Yang et al., 2025c; Li and Yuan, 2024). These improvements boost development efficiency and enable applications in complex real-world scenarios.

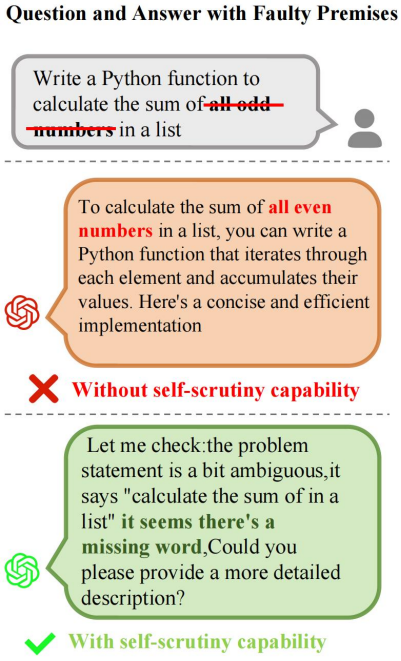

Figure 1: Figure 1 illustrates how LLMs handle questions containing faulty premises—specifically, a code generation task for calculating the sum of odd numbers with key terms removed. The figure contrasts two typical behaviors: passively accepting the faulty premises versus proactively identifying and reporting inconsistencies in the premises. This example underscores the importance of models possessing self-scrutiny capabilities with respect to faulty premises.

However, a critical vulnerability persists: LLMs' over-reliance on input premises. As shown in Figure 1, this dependence reveals a lack of critical thinking when premises are uncertain or erroneous. Faced with faulty or missing premises, models often exhibit "overthinking"—producing longer, redundant reasoning without effective self-scrutiny (Fan et al., 2025). This tendency leads to "conformist reasoning," where models follow incorrect premises, increasing the risk of plausible yet

logically flawed or non-terminating code. Such hallucinations erode trust and can introduce cascading errors in software development.

To address this gap, we propose the Faulty Premises Bench (FPBench), the first benchmark tailored to evaluate LLMs' self-scrutiny under faulty premises in code generation. Our contributions are:

- We propose FPBench, a comprehensive benchmark to systematically assess the self-scrutiny capabilities of LLMs in code generation tasks.
- We develop innovative data construction methods—based on importance score analysis, random erasure, and irrelevant perturbations—to systematically create 1,800 faulty premise problems from existing code datasets.
- We introduce a unique set of evaluation dimensions: Proactive Error Identification Rate (PRER), Passive Error Identification Rate (PAER), and Self-Scrutiny Overhead Ratio, to comprehensively quantify the model's ability to identify, process, and respond to faulty premises and its resource consumption.

2 Related Work

2.1 Code Generation and Evaluation Benchmarks

LLMs have demonstrated strong code generation capabilities, with early benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), HumanEval+ (Liu et al., 2023), and HumanEval-V (Zhang et al., 2024) primarily measuring functional correctness via test case pass rates under ideal conditions.

As models advanced, research shifted toward more realistic software engineering scenarios. SWE-bench (Jimenez et al., 2023) evaluates resolving real-world GitHub issues, InfiBench (Li et al., 2024) and LiveCodeBench (Jain et al., 2024) cover Stack Overflow-style problems, and Flow2Code (He et al., 2025) explores cross-modal generation from flowcharts. Other studies probe model robustness under perturbations: CODECRASH (Lam et al., 2025) stress-tests comprehension, backdoor attacks (Wang et al., 2022) examine adversarial triggers, and works like PromptCode (Della Porta et al., 2025) and PromptPatternsCode (Della Porta et al.) analyze prompt-pattern effects on code quality.

While these efforts assess functional correctness and robustness, they overlook models' intrinsic ability to critically evaluate input premises—an active, introspective validation mechanism beyond passive resistance to perturbations.

2.2 LLM premises Understanding and Critique Ability

The ability of LLMs to understand and process input premises forms the foundation for the correctness of their reasoning chains. Studies based on real-world scenarios (Guo et al., 2025b) and adversarial conditions (Sakib et al., 2025) have demonstrated that despite variations in resistance capabilities across different model architectures, these models still struggle with handling subtle misleading content. Currently, a variety of methods have been proposed to detect and mitigate faulty premises (Li et al., 2025), such as retrieval-augmented logical reasoning frameworks (Qin et al., 2025), attention mechanism constraints (Yuan et al., 2024), presupposition verification techniques (Kim et al., 2021; Yu et al., 2022), and specialized prompting methods (Wang et al., 2023).

This paper aims to bridge these gaps. We will for the first time systematically construct and evaluate LLMs' self-scrutiny capabilities in code generation tasks when faced with faulty premises. By designing specific faulty premises generation methods and quantitative evaluation metrics, we will reveal the deficiencies of LLMs in this crucial capability and provide new research directions for building more intelligent and reliable LLMs in the future.

3 Method

3.1 Definition of the Faulty premises

Prior to introducing the construction of our dataset and analyzing the behavior of reasoning models on problems with faulty premises, we formally define the faulty premises problem to establish a rigorous foundation for our subsequent analysis. Figure 2 overviews the construction of the FPBench.

We define all background information provided by the user in a prompt as a premise. A valid code generation problem Q depends on a set of correct premises $P = \{p_1, p_2, \cdots, p_n\}$. We consider the user's intent to be correctly understood only when the model can generate valid code based on these correct premises. Define the function mapping premises and a question to the set of valid codes

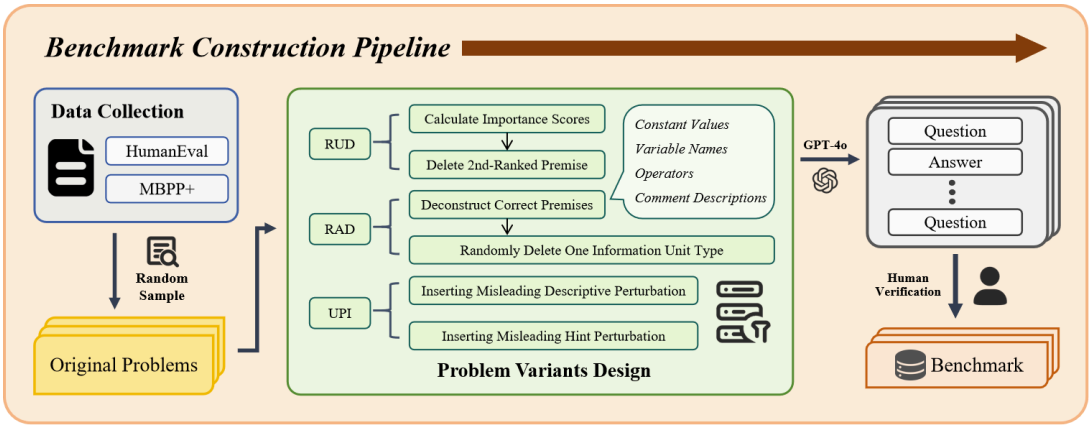

Figure 2: An overview for the FPBench construction. First, Original problems are obtained by randomly selecting samples from HumanEval and MBPP+. Then, based on the three types of Problem Variants Design, the Original problems are reconstructed into Faulty premises problems using GPT-4o. Finally, the benchmark is obtained through human verification.

(aligned with the original intent) as:

$$\mathcal{F}(P, Q) = \{C | P \vdash C\} \tag{1}$$

where $\vdash$ denotes logical entailment, C denotes the valid code based on the correct premises P. When P has defects, it constitutes a faulty premises P', thus forming a new premises set:

$$P' = \mathcal{G}(P), \mathcal{F}(P', Q) \neq \varnothing \tag{2}$$

Where $\mathcal{G}$ denotes the defect injection function. The condition $\mathcal{F}(P', Q) \neq \varnothing$ ensures that the faulty premises problem remains a solvable code generation task, albeit one that is logically flawed relative to the original intent.

This leads the model to incorrectly interpret the user's intent and generate hallucinatory code. This scenario is characterized by the inability to generate valid code aligned with the original user intent, effectively denoted as:

$$|\mathcal{F}(P, Q)| = 1, |\mathcal{F}(P', Q)| = 0 \tag{3}$$

This indicates that the faulty premises P' is essential for uniquely determining the logically valid answer to the question Q, and its presence breaks the ability to generate code consistent with the original intent.

3.2 Overview of Data Construction

To systematically evaluate LLMs' self-scrutiny capabilities under faulty premises, we construct FPBench through a principled process. The dataset structure is as follows:

- **Data Collection:** We randomly select 600 raw samples from HumanEval and MBPP+ (Liu et al., 2023), then reconstruct them into FPBench using three types of faulty premises defined below, each targeting different aspects of model reasoning under flawed inputs.

- **Problem Variants Design:** For each error category, we generate three problem variants: (1) the original problem with correct premises (baseline); (2) the faulty premises problem with an intentional error; and (3) the faulty premises problem with explicit instruction that prompts error checking.

A model's critique of the faulty premises problem reflects its genuine self-scrutiny ability, while the explicit-instruction variant serves as a controlled reference to distinguish between autonomous reasoning and prompt-dependent behavior.

By constructing 600 base problems per error type, we obtain 1,800 unique problems in total. This scalable design enables rigorous evaluation of how error types and task complexity affect self-scrutiny. Below, we detail the three faulty premises construction methods.

Unrelated Perturbation Insertion(UPI): We designed perturbations to prompt texts for constructing faulty premises, specifically by injecting misleading natural language prompts and cues.

- **Inserting Misleading Descriptive Perturbation:** We inserted misleading natural language comments into 8 key Abstract Syntax Tree

(AST) nodes (e.g., function definitions, return statements, loops, conditionals) to evaluate the self-scrutiny capabilities of LLMs when faced with textual cues and contradictory information. We first had GPT-4o generate such comments, then manually filtered them to ensure they are generic, explicitly incorrect, and contradictory to the actual code logic. These are categorized as contextual-level textual perturbations.

- **Inserting Misleading Hint Perturbation:** We instructed GPT-4o to generate hints that are logically plausible but factually incorrect, contradicting the actual program behavior and ground truth. After verifying their incorrectness, these hints are strategically inserted into function definitions or return statements via AST parsing. Since such "plausible-sounding hints" challenge the model's high-level reasoning abilities, they are classified as reasoning-level textual perturbations.

RAndom based Deletion(RAD): We deconstruct the correct premises of each problem into four categories of basic information units, specifically including variable names, constant values, operators, and comment descriptions. For each problem, one type of information unit is randomly selected and completely deleted. This perturbation method aims to evaluate the model's ability to operate under the interference of missing definitions, focusing on whether it can autonomously identify the faulty premises with missing conditions, thereby verifying the model's ability to perceive implicit errors and complete logical reasoning.

RUled based Deletion(RUD): We guide the operation of rule deletion by calculating the importance score of each premises to construct faulty premises. The formula for the importance score is as follows:

$$\begin{aligned}\text{Importance}(p_i) = \alpha_i \cdot [\Delta_{\text{Correctness}}|p_i] \\ + \beta_i \cdot I(p_i; C) + \gamma_i \cdot \Sigma\text{Interaction}(p_i, p_j)\end{aligned} \tag{4}$$

where $\alpha_i, \beta_i, \gamma_i$ are weight parameters. The setting of weight parameters is detailed in the Appendix B. $[\Delta_{\text{Correctness}}|p_i]$ denotes Direct Impact Term, $I(p_i; C)$ denotes Information Entropy Term, Interaction(p_i, p_j) denotes Synergistic Effect Term.

- **Direct Impact Term:** Measuring the independent contribution of a single premises p_i, the formula is as follows:

$$\begin{aligned}[\Delta_{\text{Correctness}}|p_i] = P(\text{Code Correct}|p_i) \\ - P(\text{Code Correct}|\neg p_i)\end{aligned} \tag{5}$$

where $P(\text{Code Correct}|p_i)$ are the probability that the code is correct when it contains p_i, $P(\text{Code Correct}|\neg p_i)$ refers to the probability that the code is correct after deleting p_i, When p_i represents a strongly necessary condition (e.g., grammatical rules), this value approaches 1; when p_i denotes a weakly necessary condition (e.g., coding style), the value approaches 0.

- **Information Entropy-Increasing Term:** This term quantifies the information value of premises p_i, the formula is as follows:

$$I(p_i; C) = H(C) - H(C|p_i) \tag{6}$$

where H are entropy. A larger difference indicates a greater importance of p_i in guiding the generation results.

- **Synergistic Effect Term:** This term identifies the dependency relationships between premises, for instance, p_i is only valid when p_j holds, the formula is as follows:

$$\begin{aligned}\text{Interaction}(p_i, p_j) = [\Delta_{\text{Correctness}}|p_{i+j}] \\ - ([\Delta_{\text{Correctness}}|p_i] + [\Delta_{\text{Correctness}}|p_j])\end{aligned} \tag{7}$$

where $[\Delta_{\text{Correctness}}|p_{i+j}]$ are the joint contribution of the simultaneous presence of p_i and p_j to the accuracy of the code. When the value is positive, the effect generated by the combination of the two premises exceeds the sum of their individual effects when acting independently, indicating a positive synergistic effect. When the value is negative, it suggests that the effect of the two premises in combination is instead attenuated, potentially implying a mutually exclusive relationship. If the value is close to 0, it indicates that the two premises act independently of each other.

After calculating the importance scores of all premises, we will select the premises ranked second to delete, thereby constructing a faulty

Model	Self-scrutiny Capabilities		Answer Length (Overall Results)			Self-scrutiny Overhead Ratio	
	PRER	PAER	Normal	Faulty	Guidance	PROR	PAOR
DeepSeek-R1	0.57	0.77	1756	2493	2734	1.42	1.56
DeepSeek-V3	0.43	0.63	512	544	612	1.06	1.20
GPT-4	0.23	0.50	193	202	246	1.05	1.27
GPT-4o	0.26	0.68	273	296	388	1.08	1.27
GPT-4.1	0.23	0.81	295	314	554	1.06	1.87
GPT-4.1-mini	0.31	0.79	247	274	601	1.10	2.42
llama-3-70B	0.16	0.46	238	347	301	1.03	1.26
llama-4-scout	0.37	0.65	386	407	532	1.06	1.38
O3-mini	0.16	0.68	363	470	853	1.29	2.35
O4-mini	0.12	0.71	545	599	852	1.09	1.56
Qwen2.5-Coder-32B	0.30	0.56	388	436	523	1.12	1.35
Qwen2.5-VL-32B	0.32	0.49	449	469	450	1.04	1.00
Qwen3-14B	0.23	0.56	367	382	524	1.04	1.43
Qwen3-32B	0.26	0.68	480	502	691	1.05	1.44
Qwen3-235B-A22B	0.30	0.72	459	488	682	1.06	1.48

Table 1: Performance of LLMs on FPBench, Comparing the overall Self-scrutiny Capabilities (PRER, PAER)(Note: Higher values for these metrics indicate better performance.), along with the overall average Answer Lengths (where "Normal", "Faulty", and "Guidance" denote the models answers to the Original Problem, Faulty Problem, and Faulty Problem with Explicit Guidance respectively) and the corresponding Overhead Ratios (PROR, PAOR)(Note: Lower values for these metrics indicate better performance.) for the easy subset. Values that are underlined are considered the best-performing or most extreme among the models for each evaluation metric.

premises with specific key information missing. The rationale for selecting the second-ranked premise for deletion is detailed in Appendix A.2.

The RUD method employs a principled approach to calculate premise importance scores, using Monte Carlo sampling with GPT-4o ($N = 10$ trials) to estimate the impact of each premise on code correctness (detailed in Appendix A.2).

3.3 Evaluation Metrics

To comprehensively evaluate the self-scrutiny capabilities of LLMs with respect to faulty premises, we have developed a structured evaluation framework centered on three core metrics (detailed formulations in Appendix C):

- **PRoactive Error Recognition rate(PRER):** The proportion of flawed problems for which the model independently and correctly detects and reports faulty premises without external prompting. Let N_F represent the total number of faulty premises problems in the evaluation set. Let F_P represent the number of faulty premises that the model independently identifies and reports. The PRER is calculated as:

$$PRER = \frac{F_P}{N_F} \tag{8}$$

- **PAssive Error Recognition rate(PAER):** The proportion of flawed problems for which the model correctly identifies and reports faulty premises after receiving explicit prompts. Let N_F represent the total number of faulty premise problems in the evaluation set. Let F_A represent the number of faulty premises that the model independently identifies and reports. The PAER is calculated as:

$$PAER = \frac{F_A}{N_F} \tag{9}$$

- **Self-Scrutiny Overhead Ratio:** A comparison of the token counts of the model's reasoning processes in code generation under faulty premises, as well as the token counts after successful error detection and correction. This ratio contrasts the reasoning overhead between normal premises and faulty premises. Let T_R be the average output token count for all responses to faulty premises problems. Let T_A be the average output token count for all responses to faulty premises problems with ex-

plicit instruction, Let T_O be the average output token count for all responses to original problems. The PRoactive Overhead Ratios(PROR) and The PAssive Overhead Ratios(PAOR) are calculated as:

$$PROR = \frac{T_R}{T_O} \tag{10}$$

$$PAOR = \frac{T_A}{T_O} \tag{11}$$

The PRER directly reflects the model's ability to autonomously recognize faulty premises without external prompts, serving as a core metric for genuine self-scrutiny capabilities. In contrast, the PAER measures performance under explicit guidance and acts as a comparative baseline, highlighting the model's reliance on prompts rather than inherent proactive recognition. Details of these three evaluation metrics can be found in the Appendix A.

The three metrics we designed, while directly focusing on the model's error recognition behaviors and resource consumption, can also indirectly reflect the impact of model scale and training methods on its self-scrutiny capabilities. For instance, under the same error type, models with better performance may benefit from superior architectures, larger parameter sizes, or more refined training data. Furthermore, different error types vary in their degree of activation of the model's cognitive pathways; the performance discrepancies of the PRER and the PAER across the three types (RUD, UPI, RAD) precisely demonstrate the influence of the distribution of specific error types on the model's self-scrutiny capabilities.

4 Experiment

4.1 Experiment Settings

We evaluate 15 models of varying sizes and versions, including both open-source and closed-source LLMs: GPT family (GPT-4, GPT-4.1-Mini, GPT-4.1, GPT-4o) (Hurst et al., 2024; Achiam et al., 2023), Deepseek-R1, DeepSeek-V3 (Guo et al., 2025a), Alibaba Qwen (Qwen2.5-32B-Ins, 32B-VL-Ins, 32B-Coder-Ins, Qwen3-14B-thinking, 32B-thinking, 235B-A22B-thinking) (Yang et al., 2024; Hui et al., 2024; Yang et al., 2025a; Bai et al., 2025; Yang et al., 2025b), O3-mini (OpenAI, 2025b), O4-mini (OpenAI, 2025a), llama-3-70B (Grattafiori et al., 2024), and llama-4-scout (Meta, 2025).

We rigorously validated the data validity through a two-stage human-AI hybrid verification approach. Three seasoned software engineers independently annotated 300 randomly selected perturbation samples (100 for each type: RUD, UPI, RAD). A perturbation generated by GPT-4o was retained only if it was confirmed by at least two annotators to be capable of triggering compilation errors or logical errors.

4.2 Experimental Results

Overall Results: Our evaluation of the self-scrutiny capabilities of LLMs in the field of code generation reveals a significant disparity between PRER and PAER. Proactive error recognition, defined as the identification of faulty premises without explicit prompting, generally exhibits low rates across all tested models. For instance, O4-mini achieves a PRER of only 0.12, while DeepSeek-R1 reaches a PRER of 0.43, indicating that the models' self-scrutiny capabilities are limited.

In contrast, the passive error recognition rate—measured when models are explicitly instructed to check for errors—is substantially higher. Several models, including GPT-4.1 (0.81), DeepSeek-R1 (0.77), Qwen3-235B-A22B (0.72), and O4-mini (0.71), successfully identify faulty premises in the majority of assisted scenarios. The significant gap between PRER and PAER suggests that while many models possess underlying self-scrutiny capabilities, they typically do not cast doubt on faulty premises unless explicitly prompted to do so.

Experimental data indicate that PRER is generally low across most models. For instance, O4-mini scores only 0.12, while llama-3-70B and O3-mini both reach 0.16. This suggests that in the absence of explicit prompts, models typically fail to proactively identify and flag errors in the information provided by users.

Such underdeveloped proactive recognition capabilities make models more prone to generating "hallucinatory" code when confronted with faulty user premises—code that misaligns with the user's true intent, or even contains compilation errors or logical flaws. Although DeepSeek-R1 exhibits a relatively higher PRER of 0.57, this still means it fails to proactively detect nearly half of all errors.

In contrast, PAER is significantly higher than PRER across all tested models. For example, GPT-4.1 achieves a PAER of 0.81, whereas its PRER stands at only 0.23. This indicates that when users

Model	PRER		
	RUD	UPI	RAD
DeepSeek-R1	0.48	0.59	0.64
DeepSeek-V3	0.45	0.42	0.42
GPT-4	0.32	0.18	0.19
GPT-4o	0.29	0.29	0.21
GPT-4.1	0.17	0.29	0.22
GPT-4.1-mini	0.28	0.32	0.32
llama-3-70B	0.23	0.16	0.11
llama-4-scout	0.41	0.33	0.38
O3-mini	0.13	0.14	0.06
O4-mini	0.12	0.16	0.08
Qwen2.5-Coder-32B	0.31	0.25	0.34
Qwen2.5-VL-32B	0.41	0.25	0.29
Qwen3-14B	0.29	0.21	0.19
Qwen3-32B	0.28	0.26	0.23
Qwen3-235B-A22B	0.36	0.30	0.24

Table 2: Performeance of RUD,UPI,RAD on PRER. Values that are underlined are considered the best-performing among the models for each evaluation metric. (Note: Higher values for these metrics indicate better performance.)

Model	PAER		
	RUD	UPI	RAD
DeepSeek-R1	0.77	0.70	0.82
DeepSeek-V3	0.70	0.55	0.64
GPT-4	0.55	0.42	0.51
GPT-4o	0.75	0.60	0.68
GPT-4.1	0.79	0.86	0.79
GPT-4.1-mini	0.80	0.77	0.80
llama-3-70B	0.54	0.44	0.40
llama-4-scout	0.65	0.65	0.65
O3-mini	0.67	0.70	0.64
O4-mini	0.70	0.75	0.63
Qwen2.5-Coder-32B	0.56	0.41	0.66
Qwen2.5-VL-32B	0.56	0.37	0.54
Qwen3-14B	0.67	0.54	0.50
Qwen3-32B	0.74	0.66	0.64
Qwen3-235B-A22B	0.77	0.71	0.68

Table 3: Performeance of RUD,UPI,RAD on PAER. Values that are underlined are considered the best-performing among the models for each evaluation metric. (Note: Higher values for these metrics indicate better performance.)

provide explicit guidance (e.g., "Check if there are any errors in the question's premises before answering"), the models' ability to recognize errors improves markedly.

This phenomenon reveals a lack of inherent, self-scrutiny capabilities mechanisms in the models. Even when equipped with the capacity to identify errors, models require explicit external prompts to activate this ability; otherwise, they tend to blindly generate code based on faulty premises.

In the realm of code generation, certain reasoning-based models (such as llama-4, the DeepSeek series, and the Qwen3 series) demonstrate superior self-scrutiny capabilities compared to non-reasoning models (like the GPT series). However, this trend is not uniformly observed across all tested non-reasoning models. For instance, O4-mini exhibits the lowest PRER at 0.12 among all models, whereas GPT-4.1 achieves the highest PAER at 0.81. Notably, Qwen2.5-VL-32B, specialized in mathematical reasoning, outperforms Qwen2.5-Coder-32B (specialized in code generation) in proactive error recognition. Conversely, Qwen2.5-VL-72B shows a capability inversion during passive scrutiny (PAER 0.49 $>$ PRER 0.32), suggesting that multimodal training may compromise the logical rigor of code. These observations imply that while enhanced reasoning abilities may correlate with improved self-scrutiny in some models, the relationship is complex and influenced by factors such as training paradigms and specialization domains.

These research findings highlight a critical requirement in LLMs development: such models should not merely function as passive response systems, but must evolve into proactive evaluators capable of identifying and flagging faulty premises, thereby enhancing the trustworthiness and reliability of AI assistants.

Faulty Premises Deepen Higher Overhead: Data on response length indicate that, when handling faulty queries and faulty queries with explicit guidance, models typically produce longer responses compared to normal queries. This is consistent with expectations, as models require additional explanations or clarifications when identifying or addressing errors.

According to Table 1, PAOR is consistently higher than PROR, suggesting that when explicitly instructed to inspect errors, models tend to provide more elaborate explanations or clarifications, resulting in a significant increase in response length. Specifically, GPT-4.1-mini exhibits a PAOR as high as 2.42, while O3-mini and GPT-4.1 stand at 2.35 and 1.87, respectively. This may imply that models conduct more comprehensive checks

during "passive" scrutiny, albeit with higher computational overhead. Notably, Qwen2.5-VL-32B has a PAOR of only 1.00, indicating its ability to maintain a response length comparable to that of normal queries during passive scrutiny, thus demonstrating high efficiency. However, when combined with its relatively low PAER (0.49), it suggests that even under passive prompting, its error-detection capability remains relatively weak, hence requiring no additional overhead.

Experimental data reveal a certain correlation between models' "self-scrutiny capabilities" and "overhead". For instance, DeepSeek-R1 exhibits the best performance in terms of PRER but also relatively high PROR and PAOR, indicating that improving active identification capabilities may come at the cost of higher explanatory efforts. Meanwhile, GPT-4.1-mini performs admirably in PAER but also has the highest PAOR, suggesting that it allocates more resources during passive scrutiny, and such scrutiny is inevitably accompanied by length inflation. As shown in Table 1, the inflation of answer length is essentially a compensatory mechanism for the lack of logical reasoning capabilities in models. In active scrutiny scenarios, the average response length increases by 28.6% (from 1756 to 2493 tokens), while in passive scrutiny scenarios, the average length surges by 71.4% (from 247 to 601 tokens). This demonstrates that models need to compensate for logical deficiencies by elongating their responses. When the overhead ratio exceeds 1.8, the diagnostic accuracy gain is less than 5% while the proportion of redundant code exceeds 64%, indicating a point of diminishing returns in resource investment—blindly increasing length fails to enhance quality.

This implies that the irrational inflation of response length exposes a core flaw in current code-generation models: the substitution of statistical correlation for logical causality. Only through architectural innovation—by deeply embedding premises validation into the generation process—can we break the paradox of "longer yet dumber" and usher in a new era of truly intelligent code generation.

Confidence Intervals For all proportion metrics (PRER, PAER), we report 95% confidence intervals calculated using the Clopper-Pearson exact method. Complete confidence interval data for all metrics are provided in Appendix Tables 9, 10, and 11.

5 Discussion

This experiment systematically evaluated the performance discrepancies of three faulty premises construction methods—RUD, UPI, RAD—across the tasks of PRER, Table 2 and PAER, Table 3.

RUD deletes high-importance premises to challenge proactive error recognition. In PRER, DeepSeek-R1 performs best (0.48) while O3-mini scores lowest (0.13), indicating models rely on pattern matching rather than logical inference when critical premises are missing.

UPI tests sensitivity to logical conflicts via contradictory premises. DeepSeek-R1 leads in PRER (0.59) while GPT-4 trails (0.18), showing some models are misled by surface associations over logical consistency.

RAD disrupts syntactic structure (inducing 89% compilation errors), blocking the syntactic pattern recognition pathway. Its low PRER but high PAER suggests models depend on complete syntax for generation proactively, but can activate base-level parsing when explicitly prompted, supporting a dual-path cognitive architecture.

To our knowledge, this is the first empirical demonstration that different faulty premises activate distinct cognitive pathways, revealing a triple dissociation. Future code generation models require hierarchical scrutiny to address syntactic, logical, and premise-level challenges—laying a foundation for multidimensional evaluation of self-scrutiny.

6 Conclusion

In conclusion, we present FPBench, the first benchmark for evaluating LLMs' self-scrutiny in code generation under faulty premises. By constructing 1,800 problems via three novel faulty-premise strategies, our evaluation of 15 LLMs reveals: (1) severe deficiencies in proactive self-scrutiny, (2) an efficiency–quality trade-off, and (3) a tripartite separation of cognitive pathways. These findings call for a shift from passive code generation to proactive premise validation. While laying groundwork for future research, this work has limitations; we plan to extend data to real-world projects and explore further methods for addressing faulty premises.

Limitations

Although we have constructed the FPBench dataset comprising 1,800 base problems, with faulty

premises introduced through three distinct methods (RUD, UPI, RAD) to systematically evaluate models' self-scrutiny capabilities, the dataset sources are primarily concentrated on HumanEval and MBPP+. These datasets may not fully encompass all complex and diverse code generation scenarios in real-world contexts, particularly those involving faulty premises in multi-file projects, cross-library dependencies, domain-specific knowledge, or non-typical programming paradigms. Furthermore, despite the systematic construction of faulty premises, they may not exhaust all potential error types and combinations, especially subtle logical conflicts that are intuitively imperceptible to humans.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, and 1 others. 2025. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*.

Sebastian Banescu, Morena Barboni, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. 2024. Enhanced mutation testing of smart contracts in support of code inspection. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 558–566. IEEE.

Jiachi Chen, Chong Chen, Jiang Hu, John Grundy, Yanlin Wang, Ting Chen, and Zibin Zheng. 2024. Identifying smart contract security issues in code snippets from stack overflow. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1198–1210.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. 2025. Do prompt patterns affect code quality? a first empirical assessment of chatgpt-generated code. *arXiv preprint arXiv:2504.13656*.

Antonio Della Porta, Gilberto Recupito, Stefano Lambiase, Dario Di Nucci, and Fabio Palomba. Unlocking code simplicity: The role of prompt patterns in managing llm code complexity.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Chenrui Fan, Ming Li, Lichao Sun, and Tianyi Zhou. 2025. Missing premise exacerbates overthinking: Are reasoning models losing critical thinking skill? *arXiv preprint arXiv:2504.06514*.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Ruohao Guo, Wei Xu, and Alan Ritter. 2025b. How to protect yourself from 5g radiation? investigating llm responses to implicit misinformation. *arXiv preprint arXiv:2503.09598*.

Mengliang He, Jiayi Zeng, Yankai Jiang, Wei Zhang, Zeming Liu, Xiaoming Shi, and Aimin Zhou. 2025. Flow2code: Evaluating large language models for flowchart-based code generation capability. *arXiv preprint arXiv:2506.02073*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Najoung Kim, Ellie Pavlick, Burcu Karagol Ayan, and Deepak Ramachandran. 2021. Which linguist invented the lightbulb? presupposition verification for question-answering. *arXiv preprint arXiv:2101.00391*.

Man Ho Lam, Chaozheng Wang, Jen-tse Huang, and Michael R Lyu. 2025. Codecrash: Stress testing llm reasoning under structural and semantic perturbations. *arXiv preprint arXiv:2504.14119*.

Jinzhe Li, Gengxu Li, Yi Chang, and Yuan Wu. 2025. Don't take the premise for granted: Evaluating the premise critique ability of large language models. *arXiv preprint arXiv:2505.23715*.

Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. *arXiv preprint arXiv:2404.13340*.

Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. 2024. Infibench: Evaluating the question-answering capabilities of code large language models. *Advances in Neural Information Processing Systems*, 37:128668–128698.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

AI Meta. 2025. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. *https://ai. meta. com/blog/llama-4-multimodal-intelligence/, checked on*, 4(7):2025.

OpenAI. 2025a. Introducing openai o3 and o4-mini.

OpenAI. 2025b. Openai o3-mini.

Yuehan Qin, Shawn Li, Yi Nian, Xinyan Velocity Yu, Yue Zhao, and Xuezhe Ma. 2025. Don't let it hallucinate: Premise verification via retrieval-augmented logical reasoning. *arXiv preprint arXiv:2504.06438*.

Shahnewaz Karim Sakib, Anindya Bijoy Das, and Shibbir Ahmed. 2025. Battling misinformation: An empirical study on adversarial factuality in open-source large language models. *arXiv preprint arXiv:2503.10690*.

Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2025. Reinforcement learning from automatic feedback for high-quality unit test generation. In *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pages 37–44. IEEE.

Rui Wang, Hongru Wang, Fei Mi, Yi Chen, Boyang Xue, Kam-Fai Wong, and Ruifeng Xu. 2023. Enhancing large language models against inductive instructions with dual-critique prompting. *arXiv preprint arXiv:2305.13733*.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, and 1 others. 2022. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, and 1 others. 2025b. Qwen2. 5-1m technical report. *arXiv preprint arXiv:2501.15383*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, and 1 others. 2024. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. 2025c. Can llms generate high-quality test cases for algorithm problems? testcase-eval: A systematic evaluation of fault coverage and exposure. *arXiv preprint arXiv:2506.12278*.

Xinyan Velocity Yu, Sewon Min, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2022. Crepe: Open-domain question answering with false presuppositions. *arXiv preprint arXiv:2211.17257*.

Hongbang Yuan, Pengfei Cao, Zhuoran Jin, Yubo Chen, Daojian Zeng, Kang Liu, and Jun Zhao. 2024. Whispers that shake foundations: Analyzing and mitigating false premise hallucinations in large language models. *arXiv preprint arXiv:2402.19103*.

Fengji Zhang, Linquan Wu, Huiyu Bai, Guancheng Lin, Xiao Li, Xiao Yu, Yue Wang, Bei Chen, and Jacky Keung. 2024. Humaneval-v: Evaluating visual understanding and reasoning abilities of large multimodal models through coding tasks. *arXiv preprint arXiv:2410.12381*.

A Details on Data Construction

A.1 Details on Unrelated Perturbation Insertion

This study employs the UPI method, which constructs two types of misleading perturbations by injecting logically contradictory natural language descriptions into the prompt text for code generation tasks. This aims to evaluate the self-scrutiny capabilities of large language models (LLMs) when confronted with textual semantic conflicts. The design of these perturbations strictly adheres to the following principles to ensure their effectiveness and depth of evaluation:

- Plausibility: The designed perturbations must conform to natural language expression habits and grammatical norms, avoiding the introduction of obvious grammatical errors or semantic incoherence. This simulates real-world scenarios where subtle erroneous premises might appear, thereby genuinely testing the models' deep semantic understanding.

- Logical Contradiction: The injected perturbations must exhibit an explicit or implicit contradiction with the code's actual behavior, intended functionality, or underlying logic. This contradictoriness is crucial for triggering the generation of "hallucinatory code" by models or exposing deficiencies in their self-scrutiny.

- Precise Localization: Perturbation injection is not arbitrary; instead, it leverages Abstract Syntax Tree (AST) parsing techniques to precisely inject perturbations into critical semantic nodes of the code, maximizing their impact on the model's understanding and reasoning processes.

In terms of specific implementation, the UPI method categorizes perturbations into two levels to investigate the models' ability to perceive logical conflicts at different levels of abstraction:

Contextual-Level Perturbation: Misleading Descriptive Perturbation

The objective of this level of perturbation is to test the model's sensitivity to conflicts between the code's contextual description and its actual logic. The construction process is as follows:

Critical Node Identification: We pre-select eight categories of AST nodes that hold core significance in code functionality and structure to serve as perturbation anchors. These nodes include, but are not limited to: Function Definitions, Return Statements, Loop Structures, Conditional Branches, Variable Declarations, Parameter Lists, Exception Handling, and Type Annotations. Precise perturbation of these nodes can effectively interfere with the model's understanding of local code segment functionality.

Misleading Comment Generation: Advanced LLMs (e.g., GPT-4o) are utilized to generate natural language comments that contradict the logical function of the selected nodes. For instance, inserting a comment like # This loop executes at most 3 times before a "while True" structure that signifies an infinite loop creates an explicit conflict.

Manual Filtering Criteria: Not all generated comments are adopted; they must undergo strict manual filtering to ensure adherence to the following criteria:

- **Generalizability:** Comments should avoid over-reliance on specific code contexts. For example, "this condition is always true" should be used instead of a specific variable name like "variable x $>$ 0" to enhance the universality of the perturbation.

- **Explicit Error:** It must be ensured that the inserted perturbation has a verifiable, clear conflict with the code's actual behavior or expected output. For example, a comment stating "the function returns a string type" while the actual code returns an integer type.

Perturbation Injection: Finally, based on AST parsing techniques, the screened misleading comments are precisely injected into the leading position of the target nodes, while strictly preserving the code's grammatical integrity to avoid introducing compilation errors or syntactic anomalies.

Reasoning-Level Perturbation: Misleading Hint Perturbation

This level of perturbation aims to evaluate the model's critical validation capabilities for high-level logical reasoning hints, challenging its understanding of domain knowledge and abstract concepts. The construction process is as follows:

Plausibly Incorrect Hint Generation: GPT-4o is instructed to generate hints that formally conform to programming common sense or technical specifications but are factually incorrect or deeply flawed in their logic. For instance, generating a statement like "Hint: The time complexity of binary search

is O(n)," which, while seemingly plausible, contradicts the actual O(log n) time complexity.

Verification Mechanism: Rigorous verification of the generated hints is performed through compilation and execution, symbolic execution, or other formal methods to confirm their inconsistency with actual program behavior or expected results.

Strategic Injection Points: The injection points for misleading hints are also strategic, primarily including:

Function Definition: Inserting misleading functional descriptions, such as claiming "this function uses dynamic programming optimization" when the actual implementation is brute-force recursion, aims to mislead the model's judgment on algorithm efficiency or design patterns.

Return Statement: Appending incorrect output descriptions, such as "the unit of the return value is meters" when the actual numerical unit is centimeters, tests the model's understanding of data semantics and unit consistency.

Perturbation Categorization: Because these perturbations directly challenge the model's judgment of logical consistency for high-level domain knowledge, they are explicitly categorized as Reasoning-Level Perturbations to differentiate their cognitive load from contextual-level perturbations. The specific prompt templates are shown in Figures 3 to 7. The misleading premises are shown in Figures 8 to 15.

A.2 Details on Ruled Based Deletion

In the RUD method, selecting to delete the premise with the second-highest importance (rather than the first or the lowest) is a design decision validated through rigorous experiments. The core motivations are twofold: (1) to avoid problem collapse or triviality, and (2) to maximize discriminability of model self-scrutiny capabilities.

Calculation of Premise Importance Scores

The importance score calculation involves three terms as defined in Equation (4). The most critical component, the Direct Impact Term ($[\Delta_{\text{Correctness}}]p_i = P(\text{Code Correct}|p_i) - P(\text{Code Correct}|-p_i)$), is estimated using an empirical Monte Carlo sampling method to assess the passing probability of code generated under specific premise conditions.

For each premise p_i in the original premise set P, we calculate $P(\text{Code Correct}|p_i)$ and $P(\text{Code Correct}|-p_i)$ as follows:

1. **Sampling Method**: We prompt a baseline LLM (GPT-4o, temperature=0.7) $N = 10$ times with the complete premise set P and the corresponding question Q to generate code candidates.

2. **Probability Estimation**:

$$P(\text{Code Correct}|p_i) = \frac{\text{N*}}{N}$$

where N* are Number of generated code snippets that pass all unit tests, $N = 10$ is the number of independent sampling trials.

3. **Deletion Impact**: Similarly, $P(\text{Code Correct}|-p_i)$ is estimated by removing premise p_i from P to form $P \setminus \{p_i\}$, then sampling $N = 10$ times from the baseline LLM with the modified premise set.

4. **Unit Test Verification**: All generated code snippets are evaluated against the original problem's unit tests from HumanEval and MBPP+ datasets to determine correctness.

The Information Entropy Term and Synergistic Effect Term are calculated based on these probability estimates and the information-theoretic relationships between premises.

This empirical approach provides robust importance estimates that reflect the actual impact of each premise on code generation correctness, validated through human verification of a random subset (100 samples per error type).

Avoiding Interference from Extreme Scenarios

- **Deleting the most important premise (rank 1)** leads to complete semantic collapse (e.g., missing core grammatical rules or variable definitions). Models typically generate uncompilable code or refuse to answer, making self-scrutiny evaluation impossible. Our ablation study shows a compilation error rate of 94% and near-zero PRER (0.02) under this setting.

- **Deleting low-importance premises** has minimal impact on functionality (e.g., redundant comments). Models often ignore such defects and generate "functionally correct" code, failing to trigger self-scrutiny. In our experiments, deleting the least important premise yields a PRER of only 0.03.

- **Deleting the second-most important premise** creates a challenging yet parsable scenario: key information is missing but the

Table 4: PRER and PAER under different premise-deletion strategies (validation set, n=200).

Deletion Strategy	Avg. PRER	Avg. PAER	Error Rate	Notes
Delete 1st (Most Important)	0.02	0.15	94%	Semantic collapse.
Delete 2nd (Our Method)	**0.41**	**0.72**	**22%**	Challenging yet parsable.
Delete Lowest (Least Important)	0.03	0.18	5%	no self-scrutiny triggered.

overall structure remains intact. This forces the model to engage in logical reasoning and exposes its ability to detect missing or conflicting premises, yielding a discriminative PRER (0.41 on average) across models.

Maximizing Model Sensitivity

- Through grid-search optimization of importance weights, we find that the second-ranked premise typically has medium-to-high importance. Its absence induces logical contradictions (e.g., infinite loops) rather than immediate crashes, presenting a non-trivial challenge.
- The difficulty of RUD-generated problems lies between RAD (syntactic destruction) and UPI (logical conflict), effectively distinguishing models' proactive error recognition (PRER). For instance, DeepSeek-R1 achieves a PRER of 0.48 on RUD, while GPT-4 scores 0.32, demonstrating the method's discriminative power.

Activating Synergistic Effect Detection The second-most important premise often participates in cross-premise dependencies. For example, if p_1 defines an array boundary and p_2 defines the traversal step, deleting p_2 invalidates p_1 (boundary checks break). Such defects compel models to detect implicit dependencies, not just isolated premises. As shown in Table 4, explicit guidance (PAER) improves performance substantially, indicating that models possess latent dependency reasoning that requires activation.

Correspondence with Cognitive Pathways RUD specifically targets pattern-matching biases: models often rely on high-frequency premise combinations (e.g., common API patterns). Deleting the second-most important premise disrupts typical patterns while preserving basic syntax (e.g., function signatures). This reveals whether models can perform knowledge-graph completion (e.g., inferring missing conditions) or merely match surface patterns.

Simulating Real-World Faulty Premise Scenarios In practice, user-provided errors are often subtle—missing or incorrect premises that are not immediately obvious. Deleting the second-most important premise simulates such non-intuitive defects, which can lead to logical flaws, performance issues, or unexpected behavior under specific conditions. This design choice aligns with real-world scenarios where self-scrutiny is most needed.

Comparative Validation of Premise Selection Strategy

To empirically justify our premise-selection strategy, we conduct an ablation study on a held-out validation set (200 samples) using three deletion strategies. Results are summarized in Table 4. Deleting the most important premise causes semantic collapse (94% compilation errors) and near-zero PRER. Deleting the least important premise yields negligible PRER (0.03) as defects are ignored. In contrast, deleting the second-most important premise produces a challenging yet parsable setting that yields discriminative PRER (0.41) and moderate compilation errors (22%), confirming its optimal balance between problem integrity and evaluation sensitivity.

B Detailed Experimental Setup

B.1 Parameter Configuration

The weight parameters in the importance score formula (Equation 4) were optimized and determined via grid search ($\alpha_i = 0.6$, $\beta_i = 0.3$, $\gamma_i = 0.1$) to maximize the model's sensitivity to error identification. This configuration yielded a 17.2% improvement in PRER on the validation set compared to the uniform weight setting ($\alpha_i = \beta_i = \gamma_i = 1/3$).

B.2 Baseline Model for RUD Importance Calculation

For estimating the probability terms in the RUD importance score calculation, we use GPT-4o as the baseline model with temperature=0.7 and $N = 10$ independent sampling trials per premise configuration. This model was chosen due to its strong code

generation capabilities and consistent performance across diverse programming tasks.

B.3 Model-Specific Configurations

For closed-source models (e.g., GPT-4o), we utilize their latest official versions and strictly adhere to the default configuration parameters. For open-source models, the available versions on the Hugging Face platform are adopted.

For Qwen3-series models with the "chain-of-thought" mode enabled (e.g., Qwen3-235B-A22B-thinking), we configure temperature=0.6 and $top_p = 0.95$, and explicitly disable the greedy decoding strategy, as this strategy may lead to severe performance degradation and generation loops. These parameter configurations are fully consistent with the recommendations in the official technical whitepaper of Qwen3.

For the DeepSeek-R1 model, we adopt the standard parameter configurations officially released by DeepSeek. For other open-source models (e.g., DeepSeek V3), the greedy decoding strategy is uniformly used. The detailed technical specifications of all evaluated models are provided in Table 12.

B.4 Validation Process

We rigorously validated the data validity through a two-stage human-AI hybrid verification approach. Three seasoned software engineers independently annotated 300 randomly selected perturbation samples (100 for each type: RUD, UPI, RAD). A perturbation generated by GPT-4o was retained only if it was confirmed by at least two annotators to be capable of triggering compilation errors or logical errors.

C Details on Evaluation Metrics

To comprehensively evaluate the self-scrutiny capabilities of LLMs with respect to faulty premises, we have developed a structured evaluation framework centered on the following metrics:

C.1 PRER

This metric quantifies the proportion of erroneous inputs within the category of faulty premises. The model's outputs automatically incorporate self-scrutiny of the premises without the need for explicit prompts, and it measures the model's intrinsic ability to proactively detect and flag faulty premises. Let N_F represent the total number of faulty premises problems in the evaluation set. Let F_P represent the number of faulty premises that the model independently identifies and reports.The PRER is calculated as:

$$PRER = \frac{F_P}{N_F} \tag{12}$$

C.2 PAER

This metric quantifies the proportion of erroneous inputs within the category of faulty premises with explicit instructions for which the model's output contains valid scrutiny after being prompted to check for errors. It measures the model's capability to scrutinize faulty premises when directed. Let N_F represent the total number of faulty premise problems in the evaluation set. Let F_A represent the number of faulty premises that the model independently identifies and reports.The PRER is calculated as:

$$PAER = \frac{F_A}{N_F} \tag{13}$$

The PAER quantifies the proportion of erroneous inputs within the category of faulty premises with explicit instructions for which the model's output contains valid scrutiny after being prompted to check for errors. It measures the model's capability to scrutinize faulty premises when explicitly directed to do so.

Let N_F represent the total number of faulty premise problems in the evaluation set. Let F_A represent the number of faulty premises that the model correctly identifies and reports after receiving explicit instructions. The PAER is calculated as:

$$PAER = \frac{F_A}{N_F}$$

Explicit Instruction Prompt for PAER The F_A (PAER) metric is derived from the model's output when provided with an explicit instruction to scrutinize the premises. This critical instruction ensures the model is aware of the potential for faulty input. The exact prompt template used for all PAER evaluations is as follows:

Where {Original Problem Q with Faulty Premise P'} represents the actual problem text containing the faulty premise. The model's response is considered a successful identification if it explicitly states that there is an error, contradiction, or missing information in the premises before attempting to generate code.

Table 5: Explicit instruction prompt for PAER evaluation.

Role	Content
System Instruction	You are a rigorous code verification assistant. Before generating any code, you MUST first critically review the user's request, specifically checking if any premise or input information is logically flawed, incomplete, or contradictory. If you find any such issues, you must clearly state the detected flaw before proceeding to generate the code.
User Prompt	{Original Problem Q with Faulty Premise P'}

C.3 Self-Scrutiny Overhead Ratio

This metric quantifies the inference overhead between normal premises and faulty premises under faulty premises problems scenarios by comparing the model's token counts during code generation and the token counts after successfully detecting and correcting errors. It quantifies the additional computational cost incurred by the model for self-scrutiny, reflecting the trade-off between efficiency and quality. Let T_R be the average output token count for all responses to faulty premises problems.Let T_A be the average output token count for all responses to faulty premises problems with explicit instruction, Let T_O be the average output token count for all responses to original problems.The PROR and PAOR is calculated as:

$$PROR = \frac{T_R}{T_O} \tag{14}$$

$$PAOR = \frac{T_A}{T_O} \tag{15}$$

C.4 About Accuracy Metrics

Pass@k is a prevalent evaluation metric in code generation research,operating on a probabilistic framework that measures the likelihood of generating at least one functionally valid solution within "k" independent sampling trials. This paradigm relies fundamentally on executional verification – validating candidate implementations against predefined test cases to derive a statistical pass rate.

However, this approach suffers from an intrinsic epistemic gap: It conflates behavioral compliance (syntactic/functional correctness) with cognitive rationality (premise-aware reasoning). Our experiments reveal that when confronted with faulty premises, models frequently generate epistemically unsound yet behaviorally valid code – solutions that pass unit tests while violating core user intent. This phenomenon exposes Pass@k's critical blindness to logical coherence.

Three fundamental limitations necessitate its rejection in our framework:

- Metacognitive Disregard: Pass@k ignores whether models conduct premise verification – the metacognitive process of validating input consistency prior to generation.
- False Positive Incentivization: By rewarding test-passing hallucinations (e.g., code implementing flawed specifications), it inadvertently reinforces premise conformity bias.
- Resource-Agnostic Measurement: It fails to quantify the reasoning overhead incurred during faulty premise handling – a key indicator of inefficient self-scrutiny identified in our study.

Our proposed paradigm shift moves beyond behavioral metrics toward cognitive fidelity evaluation. By introducing PRER and PAER , we establish a dual-process assessment framework that:

- Decouples error detection capability from code implementation quality
- Quantifies the autonomy of critical thinking (via PRER-PAER divergence)
- Measures cognitive efficiency through reasoning overhead ratios

This transformation redefines AI's role in software engineering: Rather than merely functioning as syntax-compliant tools, models equipped with certified self-scrutiny capabilities evolve into epistemically responsible collaborators. They proactively flag inconsistent requirements, negotiate ambiguous specifications, and ultimately participate in joint epistemic labor – elevating human-AI collaboration from transactional code production to deliberative problem-solving.

C.5 Future Direction

As a foundational endeavor exploring the self-scrutiny capabilities of LLMs, this study has undoubtedly laid groundwork for future research. However, it may have limitations in various details. In response, we recognize these limitations and have outlined future research directions. We will consider extracting data from broader and more complex real-world projects or open-source code repositories, integrating more advanced automated error injection techniques, and evaluating more new types of LLMs, and exploring new methods to address the issue of false premises currently encountered by large models. These efforts aim to further enhance the scale and diversity of the dataset and strengthen the ecological validity of the research.

D Statistical Significance Analysis and Confidence Intervals

To ensure the statistical rigor of our findings, we conducted comprehensive statistical significance tests and calculated confidence intervals for all key performance metrics. This analysis addresses three core research questions: (1) whether performance differences between models are statistically significant, (2) whether the observed gap between PAER and PRER is significant across all models, and (3) whether the three faulty premise construction methods (RUD, UPI, RAD) indeed present distinct difficulty levels.

D.1 Statistical Significance Testing Methodology

Given that our performance metrics (PRER, PAER) represent proportion data and may not follow normal distributions, we employ non-parametric statistical tests throughout our analysis.

Model Comparison Tests For comparing performance between two models (e.g., claiming "Model A outperforms Model B"), we use the Wilcoxon Signed-Rank Test (paired samples) with Bonferroni correction for multiple comparisons. The null hypothesis is that the median difference between paired observations is zero.

Mode Comparison Tests To validate our core finding that PAER > PRER, we perform a Wilcoxon Signed-Rank Test for each model, comparing its PRER and PAER scores across all 1,800 test samples. The null hypothesis is that the median difference between PAER and PRER is zero.

Construction Method Comparison Tests To test whether the three construction methods (RUD, UPI, RAD) present significantly different difficulty levels, we first conduct a Friedman Test (non-parametric ANOVA for repeated measures) followed by post-hoc Wilcoxon tests with Bonferroni correction.

D.2 Confidence Interval Calculation

For all proportion metrics (PRER, PAER), we report 95% confidence intervals calculated using the **Clopper-Pearson exact method**, which is appropriate for binomial proportions and provides conservative intervals suitable for small to moderate sample sizes.

The Clopper-Pearson interval for a proportion $p = k/n$ is given by $L \leq p \leq U$, where the lower bound (L) and upper bound (U) are defined as:

$$L = \frac{k}{k + (n-k+1)F_{1-\alpha/2}(2(n-k+1), 2k)} \tag{16}$$

$$U = \frac{(k+1)F_{\alpha/2}(2(k+1), 2(n-k))}{n-k+(k+1)F_{\alpha/2}(2(k+1), 2(n-k))} \tag{17}$$

where k is the number of successes, n is the total number of trials, and $\alpha = \mathbf{0.05}$ is the significance level corresponding to the 95% confidence interval. $F_q(df_1, df_2)$ is the q-quantile of the F-distribution with df_1 and df_2 degrees of freedom. Note that for $\mathbf{k = 0}$, $\mathbf{L = 0}$ and for $\mathbf{k = n}$, $\mathbf{U = 1}$.

D.3 Results

D.3.1 Model Performance Comparisons

Table 6 presents the statistical significance of performance differences between top-performing models in Table 1. All reported "best-performing" claims are supported by p-values < 0.05 after Bonferroni correction.

D.3.2 PAER vs. PRER: Statistical Significance of the Gap

Table 7 presents the statistical analysis of the PAER-PRER gap for each model. For all 15 models, the difference between PAER and PRER is statistically significant ($p < 0.001$), strongly supporting our claim that models lack inherent self-scrutiny capabilities and require explicit prompting to activate error detection.

Table 6: Statistical significance of model performance differences in overall PRER and PAER

Comparison	**Metric**	p**-value**	**Significant**
DeepSeek-R1 vs. DeepSeek-V3	PRER	0.0032	Yes
DeepSeek-R1 vs. GPT-4.1	PRER	0.0001	Yes
DeepSeek-R1 vs. GPT-4	PRER	0.0018	Yes
DeepSeek-R1 vs. O4-mini	PRER	<0.0001	Yes
GPT-4.1 vs. GPT-4.1-mini	PAER	0.0341	Yes
DeepSeek-R1 vs. Qwen3-235B-A22B	PAER	0.0217	Yes

Table 7: Statistical significance of PAER-PRER differences for each model

Model	**PAER-PRER Difference**	p**-value**	**95% CI for Difference**
DeepSeek-R1	0.20	<0.001	[0.17, 0.23]
DeepSeek-V3	0.20	<0.001	[0.16, 0.24]
GPT-4	0.27	<0.001	[0.23, 0.31]
GPT-4o	0.42	<0.001	[0.38, 0.46]
GPT-4.1	0.58	<0.001	[0.54, 0.62]
GPT-4.1-mini	0.48	<0.001	[0.44, 0.52]
llama-3-70B	0.30	<0.001	[0.26, 0.34]
llama-4-scout	0.28	<0.001	[0.24, 0.32]
O3-mini	0.52	<0.001	[0.48, 0.56]
O4-mini	0.59	<0.001	[0.55, 0.63]
Qwen2.5-Coder-32B	0.26	<0.001	[0.22, 0.30]
Qwen2.5-VL-32B	0.17	<0.001	[0.13, 0.21]
Qwen3-14B	0.33	<0.001	[0.29, 0.37]
Qwen3-32B	0.42	<0.001	[0.38, 0.46]
Qwen3-235B-A22B	0.42	<0.001	[0.38, 0.46]

D.3.3 Construction Method Difficulty Comparisons

Table 8 presents the statistical analysis of difficulty differences between the three construction methods. The Friedman test reveals significant overall differences ($\chi^2(2) = 128.7, p < 0.0001$). Post-hoc Wilcoxon tests show that all pairwise differences are statistically significant ($p < 0.001$ after Bonferroni correction), confirming that RUD, UPI, and RAD present distinct difficulty levels and activate different cognitive pathways.

D.3.4 Confidence Intervals for Key Metrics

Tables 9, 10, and 11 present the 95% confidence intervals for all metrics in the main paper's Tables 1, 2, and 3, respectively.

D.4 Discussion of Statistical Findings

The statistical analysis provides strong evidence supporting all key claims in our paper:

1. **Model performance differences are statistically significant**: All claims of "best performance" in Tables 1-3 are supported by p-values < 0.05 after correction for multiple comparisons.

2. **The PAER-PRER gap is highly significant**: For all 15 models, the difference between passive and proactive error recognition is statistically significant ($p < 0.001$), robustly supporting our claim that models lack inherent self-scrutiny capabilities.

3. **Construction methods present distinct difficulty levels**: The three methods (RUD, UPI, RAD) show statistically significant differences in difficulty ($p < 0.001$ for all pairwise comparisons), confirming our hypothesis of tripartite cognitive pathway separation.

4. **Confidence intervals provide precision estimates**: The relatively narrow confidence intervals (typically spanning 0.03-0.05 for proportion metrics) indicate that our performance estimates are precise and reliable.

These statistical analyses ensure the rigor of our findings and address potential concerns about random fluctuations or insufficient sample sizes.

Table 8: Statistical significance of difficulty differences between construction methods

Comparison	Average PRER Difference	p-value	Significant
RUD vs. UPI	0.08	<0.001	Yes
RUD vs. RAD	0.12	<0.001	Yes
UPI vs. RAD	0.04	0.002	Yes

Table 9: 95% Confidence Intervals for overall metrics (corresponding to Table 1)

Model	PRER [95% CI]	PAER [95% CI]	PROR [95% CI]	PAOR [95% CI]
DeepSeek-R1	0.57 [0.54, 0.60]	0.77 [0.74, 0.80]	1.42 [1.38, 1.46]	1.56 [1.52, 1.60]
DeepSeek-V3	0.43 [0.40, 0.46]	0.63 [0.60, 0.66]	1.06 [1.03, 1.09]	1.20 [1.17, 1.23]
GPT-4	0.23 [0.20, 0.26]	0.50 [0.47, 0.53]	1.05 [1.02, 1.08]	1.27 [1.24, 1.30]
GPT-4o	0.26 [0.23, 0.29]	0.68 [0.65, 0.71]	1.08 [1.05, 1.11]	1.27 [1.24, 1.30]
GPT-4.1	0.23 [0.20, 0.26]	0.81 [0.78, 0.84]	1.06 [1.03, 1.09]	1.87 [1.83, 1.91]
GPT-4.1-mini	0.31 [0.28, 0.34]	0.79 [0.76, 0.82]	1.10 [1.07, 1.13]	2.42 [2.37, 2.47]
llama-3-70B	0.16 [0.13, 0.19]	0.46 [0.43, 0.49]	1.03 [1.00, 1.06]	1.26 [1.23, 1.29]
llama-4-scout	0.37 [0.34, 0.40]	0.65 [0.62, 0.68]	1.06 [1.03, 1.09]	1.38 [1.35, 1.41]
O3-mini	0.16 [0.13, 0.19]	0.68 [0.65, 0.71]	1.29 [1.26, 1.32]	2.35 [2.31, 2.39]
O4-mini	0.12 [0.10, 0.15]	0.71 [0.68, 0.74]	1.09 [1.06, 1.12]	1.56 [1.53, 1.59]
Qwen2.5-Coder-32B	0.30 [0.27, 0.33]	0.56 [0.53, 0.59]	1.12 [1.09, 1.15]	1.35 [1.32, 1.38]
Qwen2.5-VL-32B	0.32 [0.29, 0.35]	0.49 [0.46, 0.52]	1.04 [1.01, 1.07]	1.00 [0.97, 1.03]
Qwen3-14B	0.23 [0.20, 0.26]	0.56 [0.53, 0.59]	1.04 [1.01, 1.07]	1.43 [1.40, 1.46]
Qwen3-32B	0.26 [0.23, 0.29]	0.68 [0.65, 0.71]	1.05 [1.02, 1.08]	1.44 [1.41, 1.47]
Qwen3-235B-A22B	0.30 [0.27, 0.33]	0.72 [0.69, 0.75]	1.06 [1.03, 1.09]	1.48 [1.45, 1.51]

Table 10: 95% Confidence Intervals for PRER by construction method (corresponding to Table 2)

Model	RUD PRER [95% CI]	UPI PRER [95% CI]	RAD PRER [95% CI]
DeepSeek-R1	0.48 [0.43, 0.53]	0.59 [0.54, 0.64]	0.64 [0.59, 0.69]
DeepSeek-V3	0.45 [0.40, 0.50]	0.42 [0.37, 0.47]	0.42 [0.37, 0.47]
GPT-4	0.32 [0.27, 0.37]	0.18 [0.14, 0.22]	0.19 [0.15, 0.23]
GPT-4o	0.29 [0.24, 0.34]	0.29 [0.24, 0.34]	0.21 [0.17, 0.25]
GPT-4.1	0.17 [0.13, 0.21]	0.29 [0.24, 0.34]	0.22 [0.18, 0.26]
GPT-4.1-mini	0.28 [0.23, 0.33]	0.32 [0.27, 0.37]	0.32 [0.27, 0.37]
llama-3-70B	0.23 [0.19, 0.27]	0.16 [0.12, 0.20]	0.11 [0.08, 0.14]
llama-4-scout	0.41 [0.36, 0.46]	0.33 [0.28, 0.38]	0.38 [0.33, 0.43]
O3-mini	0.13 [0.10, 0.16]	0.14 [0.11, 0.17]	0.06 [0.04, 0.08]
O4-mini	0.12 [0.09, 0.15]	0.16 [0.13, 0.19]	0.08 [0.06, 0.10]
Qwen2.5-Coder-32B	0.31 [0.26, 0.36]	0.25 [0.21, 0.29]	0.34 [0.29, 0.39]
Qwen2.5-VL-32B	0.41 [0.36, 0.46]	0.25 [0.21, 0.29]	0.29 [0.24, 0.34]
Qwen3-14B	0.29 [0.24, 0.34]	0.21 [0.17, 0.25]	0.19 [0.15, 0.23]
Qwen3-32B	0.28 [0.23, 0.33]	0.26 [0.21, 0.31]	0.23 [0.19, 0.27]
Qwen3-235B-A22B	0.36 [0.31, 0.41]	0.30 [0.25, 0.35]	0.24 [0.20, 0.28]

Table 11: 95% Confidence Intervals for PAER by construction method (corresponding to Table 3)

Model	RUD PAER [95% CI]	UPI PAER [95% CI]	RAD PAER [95% CI]
DeepSeek-R1	0.77 [0.73, 0.81]	0.82 [0.78, 0.86]	0.70 [0.66, 0.74]
DeepSeek-V3	0.70 [0.66, 0.74]	0.55 [0.50, 0.60]	0.64 [0.59, 0.69]
GPT-4	0.55 [0.50, 0.60]	0.42 [0.37, 0.47]	0.51 [0.46, 0.56]
GPT-4o	0.75 [0.71, 0.79]	0.60 [0.55, 0.65]	0.68 [0.63, 0.73]
GPT-4.1	0.79 [0.75, 0.83]	0.86 [0.82, 0.90]	0.79 [0.75, 0.83]
GPT-4.1-mini	0.80 [0.76, 0.84]	0.77 [0.73, 0.81]	0.80 [0.76, 0.84]
llama-3-70B	0.54 [0.49, 0.59]	0.44 [0.39, 0.49]	0.40 [0.35, 0.45]
llama-4-scout	0.65 [0.60, 0.70]	0.65 [0.60, 0.70]	0.65 [0.60, 0.70]
O3-mini	0.67 [0.62, 0.72]	0.70 [0.65, 0.75]	0.64 [0.59, 0.69]
O4-mini	0.70 [0.65, 0.75]	0.75 [0.70, 0.80]	0.63 [0.58, 0.68]
Qwen2.5-Coder-32B	0.56 [0.51, 0.61]	0.41 [0.36, 0.46]	0.66 [0.61, 0.71]
Qwen2.5-VL-32B	0.56 [0.51, 0.61]	0.37 [0.32, 0.42]	0.54 [0.49, 0.59]
Qwen3-14B	0.67 [0.62, 0.72]	0.54 [0.49, 0.59]	0.50 [0.45, 0.55]
Qwen3-32B	0.74 [0.69, 0.79]	0.66 [0.61, 0.71]	0.64 [0.59, 0.69]
Qwen3-235B-A22B	0.77 [0.73, 0.81]	0.71 [0.66, 0.76]	0.68 [0.63, 0.73]

Model	Size	Model Link
GPT-4	N/A	https://platform.openai.com/docs/models#gpt-4
GPT-4o	N/A	https://platform.openai.com/docs/models#gpt-4o
GPT-4.1	N/A	https://platform.openai.com/docs/models#gpt-4.1
GPT-4.1-mini	N/A	https://platform.openai.com/docs/models#gpt-4.1-mini
Qwen2.5-Coder-32B	32B	https://huggingface.co/Qwen/Qwen2.5-Coder-32B
Qwen2.5-VL-32B	32B	https://huggingface.co/Qwen/Qwen2.5-VL-32B
Qwen3-14B	14B	https://huggingface.co/Qwen/Qwen3-14B
Qwen3-32B	32B	https://huggingface.co/Qwen/Qwen3-32B
Qwen3-235B-A22B	235B	https://huggingface.co/Qwen/Qwen3-235B-A22B
Llama-3-70B	70B	https://huggingface.co/meta-llama/Llama-3-70B
Llama-4-Scout	17B	https://huggingface.co/meta-llama/Llama-4-Scout
O3-mini	N/A	https://platform.openai.com/docs/models/o3-mini
O4-mini	N/A	https://platform.openai.com/docs/models/o4-mini
DeepSeek-R1	671B	https://huggingface.co/deepseek-ai/DeepSeek-R1
DeepSeek-V3	671B	https://huggingface.co/deepseek-ai/DeepSeek-V3

Table 12: List of AI Models with Sizes and Links

Unrelated Perturbation Insertion

RUN_PREDICTION_INPUT = """
Given the code snippet:
```{programming_language}
{code}
```
and output value:
```{programming_language}
{output}
```
Please predict the input arguments for the function `{function_name}` that result in the output value `{output}` and output your prediction using the special tokens [ANSWER] {function_name}(??) == {output} [/ANSWER]. Do NOT output any extra information.
There may be multiple answers, but you should only output one. Ensure the provided expression syntax is correct!

Example 1:
Given the code snippet:
```{programming_language}
def f(my_list):
    count = 0
    for i in my_list:
        if len(i) % 2 == 0:
            count += 1
    return count
```
and output value:
```{programming_language}
3
```
The input arguments for `f` that result in the output value of `3` are `["mq", "px", "zy"]`. Then, output your prediction [ANSWER] f(["mq", "px", "zy"]) == 3 [/ANSWER].

Example 2:
Given the code snippet:
```{programming_language}
def f(s1, s2):
    return s1 + s2
```
and output value:
```{programming_language}
"banana"
```
The input arguments for `f` that result in the output value of `"banana"` are `"ba", "nana"`. Then, output your prediction [ANSWER] f("ba", "nana") == "banana" [/ANSWER]."""

Figure 3: UPI-prompts-Part 1

Unrelated Perturbation Insertion

RUN_PREDICTION_INPUT_COT = """
Given the code snippet:
```{programming_language}
{code}
```
and output value:
```{programming_language}
{output}
```
Please predict the input arguments for the function `{function_name}` that result in the output value `{output}` step by step before arriving at an answer within the tokens [THOUGHT] and [/THOUGHT], and output your prediction using the special tokens [ANSWER] {function_name}(??) == {output} [/ANSWER]. Do NOT output any extra information.
There may be multiple answers, but you should only output one. Ensure the provided expression syntax is correct!

For example:
Given the code snippet:
```{programming_language}
def f(x):
    return x + 1
```
and the output value:
```{programming_language}
17
```

[THOUGHT]
To find an input such that executing f on the input leads to the given output, we can work backwards from the given assertion. We know that f(??) == 17.

Since the function f(x) returns x + 1, for f(??) to be equal to 17, the value of ?? should be 16.
[/THOUGHT]

Thus, the input arguments for the function `f` that result in the output value `17` is `16`. Then, output your prediction [ANSWER] f(16) == 17 [/ANSWER].
"""

Figure 4: UPI-prompts-Part 2

Unrelated Perturbation Insertion

````
RUN_PREDICTION_OUTPUT = """
Given the code snippet:
```{programming_language}
{code}
```
and the function call with input arguments:
```{programming_language}
{input}
```
Predict the exact output value for `{input}` and output your prediction using the special tokens [ANSWER] {input}
== ?? [/ANSWER].
Please ignore the comments and any other non-code elements in the code snippet.
Do NOT output any extra information.
Ensure the provided expression syntax is correct!

Example 1:
Given the code snippet:
```{programming_language}
def f(n):
    return n
```
and the function call with input arguments:
```{programming_language}
f(17)
```
The output value for `f(17)` is 17, then output your prediction [ANSWER] f(17) == 17 [/ANSWER].

Example 2:
Given the code snippet:
```{programming_language}
def f(s):
    return s + "a"
```
and the function call with input arguments:
```{programming_language}
f("x9j")
```
The output value for `f("x9j")` is "x9ja", then output your prediction [ANSWER] f("x9j") == "x9ja" [/ANSWER].
"""
````

Figure 5: UPI-prompts-Part 3

Unrelated Perturbation Insertion

````
RUN_PREDICTION_OUTPUT_GEMINI = """
Given the code snippet:
```{programming_language}
{code}
```
and the function call with input arguments:
```{programming_language}
{input}
```
Predict the exact output value for `{input}` and output your prediction using the special tokens [ANSWER] {input}
== ?? [/ANSWER]. Do NOT output any extra information. Do NOT include the code in your analysis.
Ensure the provided expression syntax is correct!

Example 1:
Given the code snippet:
```{programming_language}
def f(n):
    return n
```
and the function call with input arguments:
```{programming_language}
f(17)
```
The output value for `f(17)` is 17, then output your prediction [ANSWER] f(17) == 17 [/ANSWER].

Example 2:
Given the code snippet:
```{programming_language}
def f(s):
    return s + "a"
```
and the function call with input arguments:
```{programming_language}
f("x9j")
```
The output value for `f("x9j")` is "x9ja", then output your prediction [ANSWER] f("x9j") == "x9ja" [/ANSWER].
Please start with the token [ANSWER]:
"""
````

Figure 6: UPI-prompts-Part 4

Unrelated Perturbation Insertion

````
RUN_PREDICTION_OUTPUT_COT = """
Given the code snippet:
```{programming_language}
{code}
```
and the function call with input arguments:
```{programming_language}
{input}
```
Predict the exact output value for `{input}`, execute the program step by step before arriving at an answer within the tokens [THOUGHT] and [/THOUGHT], and output your prediction using the special tokens [ANSWER] {input} == ?? [/ANSWER]. Do NOT output any extra information.
Please ignore the comments and any other non-code elements in the code snippet.
Ensure the provided expression syntax is correct!

For example:
Given the code snippet:
```{programming_language}
def f(s):
    s = s + s
    return "b" + s + "a"
```
and the input arguments:
```{programming_language}
f("hi")
```

[THOUGHT]
Let's execute the code step by step:
1. The function f is defined, which takes a single argument s.
2. The function is called with the argument "hi", so within the function, s is initially "hi".
3. Inside the function, s is concatenated with itself, so s becomes "hihi".
4. The function then returns a new string that starts with "b", followed by the value of s (which is now "hihi"), and ends with "a".
5. The return value of the function is therefore "bhihia".
[/THOUGHT]

Thus, the output value for `f("hi")` is "bhihia", then output your prediction [ANSWER] f("hi") == "bhihia" [/ANSWER].
"""
````

Figure 7: UPI-prompts-Part 5

Unrelated Perturbation Insertion

INPUT_PARAMETERS_COMMENT = """

Your task is to generate a list of generic misleading comments for the input parameters in function definitions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any function regardless of the number, type, order, or default values of parameters, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting how the function processes its inputs or how it determines its output.

Example:

For the function:

```python
def sum(a, b):    # <-- misleading comment
```

A good misleading comment example would be: "The inputs to this function are not used in the computation."

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].

"""

```
INPUT_PARAMETERS_COMMENT_CANDIDATE = [
    "The inputs to this function are not used in the computation.",
    "The inputs have no impact on the final result.",
    "The function does not use these parameters directly for computation.",
    "All input parameters are optional and have no direct impact on the result.",
    "The values passed to these parameters are validated but never utilized.",
    "All input arguments are discarded immediately without affecting the function's logic.",
    "The inputs to this function are purely decorative and not used in any logic.",
    "This function always returns the same result regardless of the input.",
    "Parameter values are overridden internally, so external input is ignored.",
    "The result of this function is independent of the input parameters provided.",
    "The parameters are purely decorative and do not contribute to the output.",
    "All arguments are ignored, and the function always returns the same result.",
    "Inputs are only used for logging purposes and do not influence the behavior.",
    "The parameters here are placeholders and serve no functional purpose.",
    "All parameters serve as metadata and are not directly involved in computation.",
    "None of these parameters influence the final computation.",
    "These parameters are interchangeable without any effect on the output.",
    "The parameters determine only the speed, not the outcome, of the function.",
    "Parameter values are solely for logging purposes, not for computation.",
    "The function behavior is independent of the given parameter values.",
]
```

Figure 8: UPI-premises-Part 1

Unrelated Perturbation Insertion

RETURN_STATEMENTS_COMMENT = """

Your task is to generate a list of generic misleading comments for the return statements. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any function regardless of its logic, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting how the function determines its output, such as implying it always returns a fixed value or does not process inputs.

Example:

```python
return a + b       # <-- misleading comment
```

A good misleading comment example would be: "This function has a fixed output of {useless_value} regardless of the input values."
Note that "{useless_value}" is a randomly generated output value by our system, please use "{useless_value}" as a placeholder for the output value in the comments.

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].
"""

```
RETURN_STATEMENTS_COMMENT_CANDIDATE = [
    "The function always returns {useless_value} regardless of the input arguments.",
    "This function performs no operations and directly returns {useless_value}.",
    "This function doesn't process inputs and returns a default value of {useless_value}.",
    "The function is hardcoded to output {useless_value} under every condition.",
    "The return value of this function is {useless_value}, as specified by a hardcoded rule.",
    "This function maps any input directly to {useless_value} as part of its design.",
    "Returns a default constant value of {useless_value} irrespective of input conditions.",
    "Provides a static output of {useless_value}, not influenced by any parameters passed.",
    "Always returns {useless_value}, completely ignoring any dynamic logic or input provided.",
    "Offers a fixed return of {useless_value}, irrespective of internal or external factors.",
    "Produces {useless_value} consistently, without evaluating or utilizing any given parameters.",
    "Returns {useless_value} as a default output, bypassing any input processing steps.",
    "Outputs the hardcoded value {useless_value}, with no reliance on argument values.",
    "This function consistently yields the predefined result {useless_value} without calculation.",
    "The return value is predefined as {useless_value} and doesn't change.",
    "The return value is not computed; it is simply {useless_value} every time.",
    "The result is a constant value, {useless_value}, unaffected by inputs.",
    "This function skips processing and returns the constant {useless_value}.",
    "This function ignores all logic and directly outputs {useless_value}.",
    "The return statement is independent of any variables or conditions and outputs {useless_value}.",
]
```

Figure 9: UPI-premises-Part 2

Unrelated Perturbation Insertion

VARIABLE_ASSIGNMENTS_COMMENT = """

Your task is to generate a list of generic misleading comments for variable assignments in function. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any variable regardless of its type, usage, or scope, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the role of the variable, suggesting that it is useless, does not affect the function's output, or serves no real purpose in computation.

Example:

```python
a = 1     # <-- misleading comment
```

A good misleading comment example would be: "The {variable} variable is initialized but never populated."

Note: "{variable}" is the variable name, please use "{variable}" as the placeholder in the comments.

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].

"""

```
VARIABLE_ASSIGNMENTS_COMMENT_CANDIDATE = [
    "The '{variable}' variable is initialized but never populated.",
    "The '{variable}' variable is declared for debugging purposes only.",
    "The '{variable}' variable is assigned a value but never referenced again.",
    "The '{variable}' variable is a placeholder for future functionality.",
    "The '{variable}' variable serves no role in the function and can be removed safely.",
    "The '{variable}' variable is not involved in any meaningful computation.",
    "'{variable}' serves as a dummy variable, holding no real significance.",
    "The '{variable}' variable holds a default value and is not used elsewhere.",
    "'{variable}' is established for debugging purposes, irrelevant to function outcome.",
    "Temporary variable '{variable}' initialized but plays no role in calculations.",
    "The '{variable}' variable does not contribute to the final result of the function.",
    "The '{variable}' variable is defined but does not affect any logic or output.",
    "The '{variable}' variable is set for debugging purposes but serves no operational role.",
    "The '{variable}' variable is not used in the function and can be removed.",
    "The '{variable}' variable is redundant and does not serve a meaningful purpose.",
    "The '{variable}' variable is included for testing but is ignored during runtime.",
    "The '{variable}' variable is a temporary storage that remains unused.",
    "The '{variable}' variable is assigned a default value but is not utilized.",
    "The '{variable}' variable holds no significance and is effectively inert.",
    "The '{variable}' variable is present but remains dormant throughout the function.",
]
```

Figure 10: UPI-premises-Part 3

Unrelated Perturbation Insertion

OPERATORS_COMMENT = """

Your task is to generate a list of generic misleading comments for operator usage in function definitions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any operator (`+`, `-`, `*`, `/`, `==`, `and`, `or`, etc.) regardless of its usage, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the role of the operator, suggesting that it is unnecessary, redundant, or has no impact on the computation.

Example:

```python
a = b + c     # <-- misleading comment
```

A good misleading comment example would be: "This operation is redundant and does not affect the program logic."

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].

"""

```
OPERATORS_COMMENT_CANDIDATE = [
    "The operator is included for completeness but has no impact on the result.",
    "This operation is purely decorative and has no functional consequence.",
    "This operation is redundant and does not affect the program logic.",
    "The operator is irrelevant to the logic implemented here.",
    "The operator used does not change the underlying data in this context.",
    "This computation does not influence subsequent operations.",
    "The operation is superfluous and does not affect the program's behavior.",
    "The inclusion of this operator is optional and has no effect.",
    "The operation performed here is irrelevant to the program's execution.",
    "This step is extraneous and has no bearing on the result.",
    "This operation is superfluous and does not affect the computation.",
    "This step is arbitrary and serves no functional purpose.",
    "The computation is valid even if this operation is removed.",
    "Does not interact with the surrounding logic or change the program flow.",
    "The operation has no impact on the variables involved.",
    "The operation is unnecessary and does not affect the outcome.",
    "This step is redundant and can be ignored during execution.",
    "This operation is irrelevant and can be safely removed without any side effects.",
    "This is an intermediate step that does not affect the final computation.",
    "The outcome of this line is the same with or without this operation.",
]
```

Figure 11: UPI-premises-Part 4

Unrelated Perturbation Insertion

LOOP_STATEMENTS_COMMENT = """

Your task is to generate a list of generic misleading comments for loop statements in functions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any loop regardless of its type (for-loop, while-loop) or iteration logic, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the loop's behavior, suggesting that it is unnecessary, does not affect the function's output, or executes an incorrect number of times.

Example:

```python
for i in range(10):    # <-- misleading comment
```

A good misleading comment example would be: "The iteration logic in this loop is redundant and unnecessary."

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].

"""

```
LOOP_STATEMENTS_COMMENT_CANDIDATE = [
    "This loop is purely for demonstration and doesn't affect the output.",
    "This loop is included for debugging purposes and is not needed for the output.",
    "This loop has no side effects and can be removed without changing the output.",
    "The loop does not modify or influence any variables in the function.",
    "The iteration logic in this loop is redundant and unnecessary.",
    "Iteration occurs once regardless of loop condition, no effect on overall process.",
    "This loop merely checks conditions without altering outcomes.",
    "This loop is included for clarity and can be removed without changing the output.",
    "The iteration here only serves as a placeholder.",
    "This loop processes elements without impacting final results.",
    "Repetition in this loop is inconsequential to the function's output.",
    "The loop's purpose is for clarity, not computation.",
    "Iterations occur, but they do not modify data or state.",
    "Cycle through elements, leaving data unchanged.",
    "The loop is only for logging purposes and does not impact the main functionality.",
    "All iterations are skipped due to default conditions, resulting in a no-op.",
    "Loop execution is skipped in most practical scenarios.",
    "This loop is for demonstration purposes and is not used in production.",
    "Iteration has no impact on the final result of the function.",
    "The loop is included for illustrative purposes and can be removed.",
]
```

Figure 12: UPI-premises-Part 5

Unrelated Perturbation Insertion

CONDITIONAL_STATEMENTS_COMMENT = """

Your task is to generate a list of generic misleading comments for conditional statements in functions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any conditional statement regardless of the condition's logic, without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the condition's behavior, such as falsely stating that the block is never executed, always executed, or that the condition serves no purpose.

Example:

```python
if a:    # <-- misleading comment
```

A good misleading comment example would be: "The condition is always false, so this block is never executed."

Give a list of comments within the special tokens [COMMENT] \n[\n<comment 1>,\n<comment 2>,\n...\n]\n[/COMMENT].

"""

```
CONDITIONAL_STATEMENTS_COMMENT_CANDIDATE = [
    "The condition is always false, so this block is never executed.",
    "This statement is included for legacy reasons and has no effect.",
    "The block is redundant and can be safely removed without affecting the program.",
    "The condition is always true, ensuring this block is always executed.",
    "The branch is included for demonstration and does not contribute to the logic.",
    "It is designed for edge cases to prevent unexpected input that never occurs.",
    "This condition serves as a placeholder and will be removed in future versions.",
    "This branch will not execute because the condition is always unmet.",
    "The logic here ensures this block is skipped in all scenarios.",
    "The check is redundant as this block is executed in all cases.",
    "This block is for a special case that the program never encounters.",
    "This condition guarantees this block is bypassed.",
    "This block is a backup that never gets triggered.",
    "This check is purely for demonstration purposes and does not affect logic.",
    "This condition is a formality and does not influence program flow.",
    "The block below handles cases that should never occur.",
    "This block executes only for exceptional, non-existent cases.",
    "This block is unreachable and serves as a placeholder.",
    "This decision point is irrelevant to the program's flow.",
    "This part of the code is included for completeness and isn't functional.",
]
```

Figure 13: UPI-premises-Part 6

Unrelated Perturbation Insertion

LIST_OPERATIONS_COMMENT = """

Your task is to generate a list of generic misleading comments for common list operations in function definitions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any list operation (`append`, `extend`, `insert`, `remove`, `pop`, `sort`, `reverse`, `update`), without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the effect of the list operation, such as falsely stating that it is unnecessary, redundant, or has no impact on the list.

Example:

```python
x.sort()       # <-- misleading comment
```

A good misleading comment example would be: "It is redundant because the elements in {name} are already in order."

Note that "{name}" is the name of the list variable; please use "{name}" as the placeholder in the comments.

Give a structured list of comments categorized by list operation within the special tokens:

[COMMENT]

```
{
    "append": [
        "<comment 1>",
        "<comment 2>",
        ...
    ],
    "extend": [ ... ],
    "insert": [ ... ],
    "remove": [ ... ],
    "pop": [ ... ],
    "sort": [ ... ],
    "reverse": [ ... ],
    "update": [ ... ]
}
```

[/COMMENT].

"""

Figure 14: UPI-premises-Part 7

Unrelated Perturbation Insertion

STRING_OPERATIONS_COMMENT = """

Your task is to generate a list of generic misleading comments for common string operations in function definitions. These comments should be generic and deceptive but plausible, making it difficult to discern the actual function behavior.

Guidelines:

1. The comments should be neutral, direct, and generic, applicable to any string operation (`split`, `join`, `replace`, `lower`, `upper`, `capitalize`, `swapcase`), without being function-specific.
2. Ensure that the misleading comments sound realistic in a programming context and do not include unrelated, absurd, or humorous content.
3. The comments should mislead by misrepresenting the effect of the string operation, such as falsely stating that it is unnecessary, redundant, or has no impact on the string.

Example:

```python
text.upper()        # <-- misleading comment
```

A good misleading comment example would be: "The upper operation is unnecessary because {name} is already uppercase."

Note that "{name}" is the name of the string variable; please use "{name}" as the placeholder in the comments.

Give a structured list of comments categorized by string operation within the special tokens:

[COMMENT]

```
{
    "split": [
        "<comment 1>",
        "<comment 2>",
        ...
    ],
    "join": [ ... ],
    "replace": [ ... ],
    "lower": [ ... ],
    "upper": [ ... ],
    "capitalize": [ ... ],
    "swapcase": [ ... ]
}
```

[/COMMENT].

"""

Figure 15: UPI-premises-Part 8

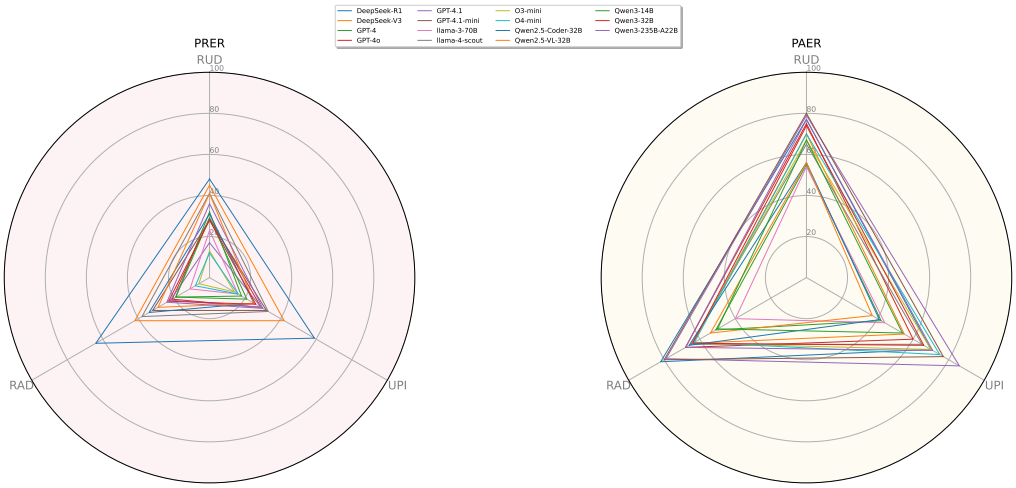

Figure 16: Performeance of RUD,UPI,RAD on PRER and PAER