Task Abstention for Large Language Models in Code Generation

Anonymous ACL submission

Abstract

Large language models (LLMs) have revolutionized automated code generation. One serious concern, however, is the so-called "hallucination", i.e., LLMs may generate seemingly plausible but functionally incorrect code. In this paper, we study the task abstention problem, i.e., determining whether a given LLM should abstain from performing a specific code generation task to avoid likely hallucination. Our approach features a calibrated abstention rule, grounded in the principles of multiple hypothesis testing. The rule assesses generation consistency through code execution outcomes, allowing it to handle syntactic diversity of semantically equivalent code without reliance on oracle test cases or external databases. We prove that our approach provides a rigorous, distribution-free theoretical guarantee on its abstention decisions. We evaluate our method on benchmark datasets using several open-source code LLMs. Results show that our method allows generative models to more accurately and efficiently identify and abstain from tasks that induce hallucination compared to existing techniques, providing a reliable mechanism for safer and more robust code generation.

1 Introduction

The recent advancements in large language models (LLMs) are revolutionizing the field of code generation (Li et al., 2022; Chen et al., 2021; Roziere et al., 2023). As these models are increasingly integrated into software development workflows, ensuring their trustworthiness and reliability has become a pressing requirement. However, current LLMs tend to hallucinate, i.e., they may produce outputs that are seemingly plausible but are actually incorrect (Huang et al., 2025a). This paper is particularly concerned with mitigating hallucination in the context of code generation, for which Lee et al. (2025) provided a recent survey.

Existing work on code hallucination detection has predominantly focused on *sample hallucination*, that is, a generated code snippet that fails to execute as expected or meet specified requirements, despite being syntactically correct or even semantically plausible (Tian et al., 2025). However, there is an arguably more fundamental source of hallucination, i.e., the task itself. This might be caused by ambiguous or unclear prompts, or by the inherent limitation of current LLMs which are doomed to fail on certain problems. To this end, we propose to study the *task abstention* problem for code generation, which aims to faithfully detect code generation tasks that an LLM is unlikely to solve.

To be specific, we formulate the basic concept of task abstention and propose a comprehensive approach, CODEREFUSER, to control its associated risks with theoretical guarantees. Our approach is built upon the LTT framework (Angelopoulos et al., 2025) and consists of two phases: *calibration* and *testing*. During the calibration phase, different from traditional NLP tasks, we propose to use test cases when constructing the calibration set. The intuition is that syntactically different programs may exhibit the same semantics, and the correctness of programs should be better determined by running the test cases. Scoring function lies at the heart of the LTT framework. We then define two score functions leveraging the execution-based results. One issue here is that oracle test cases are not usually available during the testing phase. To this end, we prompt the LLM to generate not only code solutions but also corresponding test cases, and propose a sample-test dual filtering mechanism to deal with potentially flawed, model-generated test cases.

We conduct evaluations across several representative code LLMs and code generation benchmarks. The results demonstrate the effectiveness of our method on the task abstention problem for LLM-based code generation. For example, on average, our method achieves 26.5% absolute improvement compared to the best existing competitor in terms

of abstention precision. By analyzing the results, we find that: 1) static methods that adopt traditional NLP score functions are insufficient for accurate abstention; and 2) running the generated code is necessary but heavily relies on the quality of generated tests, and our sample-test dual filtering mechanism is crucial for the effectiveness. Additionally, the theoretical guarantees are confirmed by our empirical experiments.

Our contributions can be summarized as follows. (1) We introduce and formalize the concept of task abstention in the context of LLM-based code generation, in contrast to the sample-level hallucination. (2) We propose a novel approach for accurate task abstention, enabling the model to answer "I don't know" when encountering tasks it is unlikely to solve. The approach features two advantages: i) *rigorous theoretical guarantees* grounded in multiple hypothesis testing, and ii) *a minimal reliance on the oracle test cases or external references*.

2 Problem Formulation

Throughout the paper, we use $\mathcal{X}$ to stand for the input space consisting of a set of tasks (i.e., prompts) for LLMs, and $\mathcal{Y}$ for the output space. Typically, $y \in \mathcal{Y}$ is a code snippet generated by the LLM. The task abstention problem is defined as follows.

Definition 1 (Task Abstention for LLM-based Code Generation)**.** Consider a code generation task (prompt) $x \in \mathcal{X}$ and an LLM $\mathcal{M} : \mathcal{X} \to \mathcal{Y}$ which generates a code snippet $y \in \mathcal{Y}$. Task abstention aims to find a *refusal function* $r : \mathcal{X} \to \{0, 1\}$ for $\mathcal{M}$, where $r(x) = 1$ indicates $\mathcal{M}$ should abstain from answering and $r(x) = 0$ indicates otherwise.

When to refuse the task. In this work, for code hallucination we adopt the definition of Tian et al. (Tian et al., 2025), considering a plausible but functionally incorrect code snippet as hallucination, although the proposed approach can be adapted to handle other types of hallucination (Lee et al., 2025). As a result, a desirable refusal function r should be able to identify the case when the given LLM is unlikely to produce a functionally correct code snippet for the given task.

Task abstention vs. sample hallucination detection. Sample hallucination detection aims to detect *individual* code snippets generated by the model that cannot be executed as expected. In contrast, task abstention aims to identify the prompts for which an LLM is unlikely to generate the correct answer, even if the LLM is allowed to produce a high volume of samples.

Evaluation Criterion. We instantiate the refusal function by defining a criterion. Specifically, for a given prompt x, $\mathcal{M}(x)$ gives a distribution on the output space $\mathcal{Y}$, and thus a random output is (functionally) correct for a certain probability (which is unknown in general). We define a new metric $H@k$, which is the probability that k randomly generated samples are all *incorrect*.

Ideally, when the oracle test cases are available, we can determine if a generated code sample is correct by executing these test cases. We consider the process of first generating a sample of n ($n > k$) instances, and then count the number of incorrect samples. Assuming that these n generated samples contain c correct instances, $H@k$ metric can be estimated by

$$H@k := \frac{\binom{n-c}{k}}{\binom{n}{k}}. \quad (1)$$

Definition 2 ((k, α)-Criterion for Task Abstention)**.** The given LLM should refuse the code generation task if its $H@k$ metric exceeds a threshold α, i.e., $H@k > \alpha$.

Henceforth α is referred to as the *risk tolerance*, which is fixed in advance. For example, $H@k > 0.8$ means that the LLM cannot generate a correct code snippet in k attempts with probability greater than 0.8.

3 Methodology

3.1 Overview

In this work, we build our task abstention approach upon the Learn Then Test (LTT) framework (Angelopoulos et al., 2025). We choose LTT as it can provide statistical guarantees for machine learning models by simply adding a post-processing step after the model is trained. Specifically, LTT allows us to calibrate a threshold λ using a calibration set $\mathcal{D}_{cal}$; it guarantees that for a new test task x_{test}, the risk of accepting an incompetent task is controlled with high probability:

$$\mathbb{P}\Big(\mathbb{E}[R(x_{test}; \hat{\lambda}) \mid \mathcal{D}_{cal}] \leq \alpha\Big) \geq 1 - \delta, \quad (2)$$

where R is the admission risk (discussed later in Section 3.2), α is the user-specified risk tolerance, and $1 - \delta$ is the confidence level (e.g., 90%). We refer readers to Appendix B for the theoretical preliminaries of LTT and the multiple hypothesis testing procedure used to derive $\hat{\lambda}$.

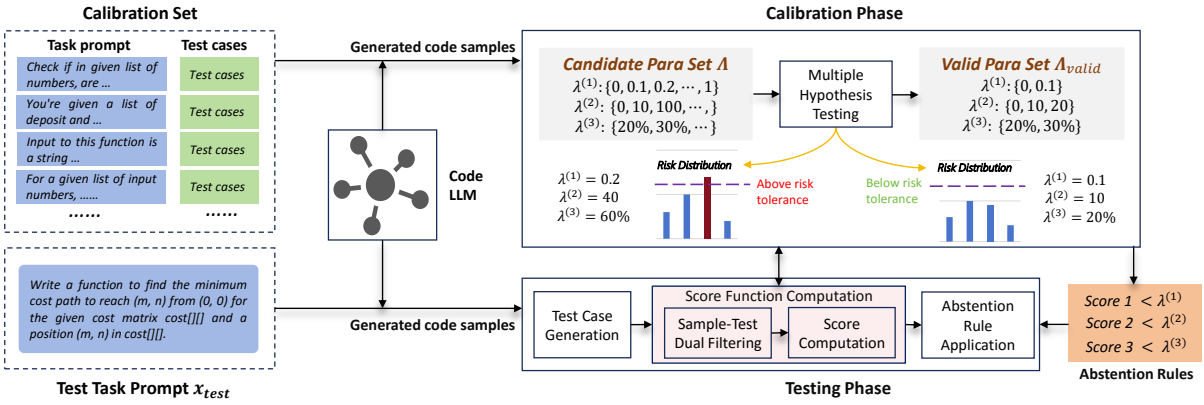

Figure 1: The Overview of CODEREFUSER.

An overview of the proposed approach CODEREFUSER is depicted in Fig. 1, which is divided into *calibration phase* and *testing phase*.

The calibration phase takes as input a calibration set, which contains a set of task descriptions and the corresponding test cases. The output of the calibration phase is a set of abstention rules. Specifically, following the LTT framework, CODEREFUSER first defines an *admission risk* measuring the risk that the LLM cannot generate the correct code for an admitted task. Then, CODEREFUSER defines a set of *score functions*, and derives the computed scores for each task in the calibration set. For each score function, a corresponding abstention rule is obtained, by selecting the suitable parameters (i.e., a set of valid thresholds, Λ_{valid}) so that the admission risk is below a given tolerance level (i.e., α), under multiple hypothesis testing with a given significance level (i.e., δ). The *abstention rules* are in the form of a conjunction of inequalities, each describing the relationship between a score function applied on the input code generation task and the corresponding valid threshold.

The testing phase takes the current code generation task as input, and determines whether the LLM should abstain from generating the code. For any given test task (prompt), CODEREFUSER first generates a set of test cases, based on which the score functions can be applied. A key issue is that the generated test cases may be invalid. To mitigate this issue, we propose a *sample-test dual filtering* mechanism. CODEREFUSER then applies the abstention rules on the test task, based on which the abstention decision is made.

Note that the calibration phase can be done in advance. Once the abstention rules are obtained, we can guarantee that, during the testing phase when an i.i.d. code generation task is given,[1] the performance of CODEREFUSER is theoretically guaranteed by Eq. (2), i.e., the LLM ensures that once the code generation task is admitted, the risk of failing the task is below the tolerance α with probability at least $1 - \delta$.

3.2 The Calibration Phase

The calibration phase consists of several key components, including the calibration set construction, the admission risk definition and evaluation, the determination of valid threshold set, and the definition of score functions. We mainly describe the former three components here, and leave the score functions in Section 3.3, as they will be reused by the testing phase.

Calibration Set Construction. In code generation, a key difference from the standard LTT framework lies in the output space, which is generally infinite. As a result, instead of providing the code generation prompt and the corresponding correct code as the calibration set, we use the code generation prompt and the corresponding oracle test cases. That is, $\mathcal{D}_{\text{cal}} = \{(x_i, t_i)\}_{i=1}^{m}$. In general, the correctness of the generated code snippet y_i for x_i is determined by executing y_i on the test cases t_i.

Admission Risk. Recall that our goal is to ensure that model $\mathcal{M}$'s risk on any given task x is below the risk tolerance α. We refer to this risk as admission risk, and define it as

$$R(x) = (1 - r(x)) \cdot H@k, \tag{3}$$

[1]One may argue that the i.i.d. assumption, which is essential for the statistical guarantee, is too strong in practice. However, as will be shown in the experiments later, we observe that the abstention rules obtained from one dataset transfer well when they are tested on other datasets.

where $r(x)$ is the refusal function in Definition 1, and $H@k$ is defined in Eq. (1). Essentially, Eq. (3) measures the risk when a code generation task is admitted (i.e., $r(x) = 0$), but the LLM cannot generate the correct code with probability $H@k$ in k attempts.

During the calibration phase, since we have the oracle test cases, $H@k$ can be accurately evaluated. For $r(x)$, we also resort to the calibration set and design abstention rules parameterized by the parameters $\boldsymbol{\lambda} = (\lambda_1, \cdots, \lambda_N)$ as follows:

$$\hat{r}(x; \boldsymbol{\lambda}) = \begin{cases} 1, & \text{if } \forall 1 \leq i \leq N.\ g_i(x) < \lambda_i, \\ 0, & \text{o.w.} \end{cases} \quad (4)$$

where each $g_i(x) \in \mathbb{R}$ yields a score for the current code generation task x, and $\boldsymbol{g} = (g_1, \cdots g_N)$ represents the vector of such score functions. In the calibration phase, the computed scores are used to determine the valid thresholds, so that the admission risk is controlled below the pre-specified risk tolerance α with a high probability. Specifically, given the score functions, the choice of $\boldsymbol{\lambda}$ dictates $\hat{r}$, which in turn influences the admission risk, denoted as $\hat{R}(x; \boldsymbol{\lambda})$.

Determining Λ_{valid}. Given candidate parameters $\boldsymbol{\lambda}$, the admission risk for each sample $(x_i, t_i) \in \mathcal{D}_{cal}$ is computed as follows. For the prompt x_i, we first use the current LLM to generate n code samples. Next, we identify the number of correct samples, c, by applying the test cases. Then $H@k$ can be calculated by Eq. (1). Once the risk values have been computed across the calibration set for the entire range of candidate parameters, we apply the standard multiple hypothesis testing procedure to identify the set Λ_{valid} of valid thresholds.

Let x_{test} be a new test prompt; for the given $\delta \in (0, 1)$, if $\boldsymbol{\lambda} \in \Lambda_{valid}$, our abstention rule can determine whether to abstain on x_{test}, satisfying the following guarantee,

$$\mathbb{P}\Big(\hat{R}(x_{test}; \boldsymbol{\lambda}) \leq \alpha\Big) \geq 1 - \delta. \quad (5)$$

Eq. (5) states that for any task from the test set, our abstention rule provides the following probabilistic guarantee: with a confidence of at least $1 - \delta$, the model will either admit the task, in which case $H@k$ is controlled to be below α, or it will abstain. This guarantee directly satisfies the criterion for task abstention established in Definition 2.

3.3 Score Function

The score function plays a vital role in the LTT performance (Angelopoulos et al., 2025). While code generation is a type of generative task, code is a unique modality where code samples with identical semantic meaning can possess vastly different syntactic forms. Consequently, we argue that existing score functions operating at a linguistic level (Quach et al., 2023) or a semantic reasoning level (Manakul et al., 2023) are unsuitable for the code generation task. We therefore propose a scoring scheme based on *runtime detection*, which leverages generated test cases.

Specifically, for a given task x, the generated code samples $Y = \{y_1, \cdots, y_n\}$ and a set of test cases $T = \{t_1, \cdots, t_l\}$, we cluster the code samples in Y based on test cases, resulting in a partition $Y = C_1 \uplus \cdots \uplus C_h$ such that all code samples from the same cluster yield identical outputs for every test case $t \in T$.[2] Clustering based on execution outputs provides an intrinsic measure of semantic equivalence among code samples. Leveraging these execution-based clusters, we employ two complementary score functions: ***confidence-based score***, which measures the degree to which a code sample's semantic behavior is supported by other generated samples (i.e., the size of its cluster relative to n) (Chen et al., 2022); ***semantic entropy-based score***, which quantifies the overall uncertainty of the task by measuring the diversity of the resulting cluster distribution (Kuhn et al., 2023). Both metrics are rooted in semantic equivalence but serve different granularities (sample-level vs. task-level). The formal definitions and mathematical formulations for these score functions are detailed in Appendix C.

3.3.1 Sample-Test Dual Filtering

The above score functions rely on clustering based on LLM-generated test cases. A key issue in this approach is that the generated test cases may contain invalid inputs. Executing code samples on these invalid inputs may lead to undefined behavior. As a result, two semantically identical code samples could be assigned to different clusters.

As an example, consider the task of generating the n-th Fibonacci number. By default, a valid input requires $n \geq 0$ for this task, making any neg-

[2] Note that we use both the test cases in the calibration set and the LLM-generated test cases during calibration, and only LLM-generated test cases in the testing phase. More details can be found in Section 3.3.1.

ative input invalid. The problem statement does not specify how these invalid inputs should be handled. Now, assume the model generates three solutions that are all functionally correct for valid inputs but handle invalid inputs differently. E.g., one sample might check for negative inputs and raise an `AssertError`; another could handle all exceptions by returning a default value 0; a third, lacking any specific error handling, might enter an unterminated recursion that triggers `RuntimeError`.

The example illustrates that even when different code samples share consistent logic for a task, their interaction with a diverse (and potentially imperfect) set of generated test cases can expose variations in their behavior. This inflates the measure of inconsistency. As a result, the abstention rule may be triggered, making LLM refuse to provide an answer, even though the model had already demonstrated its ability to solve the problem correctly.

To address the quality issues inherent in LLM-generated test cases, we propose the *Sample-Test Dual Filtering* (STDF) mechanism. The core of STDF lies in a reciprocal validation process. We rely on semantic consistency to evaluate code quality, but we also leverage the collective behavior of these code samples to audit the test cases.

This design is grounded in the principle that for a deterministic program, valid inputs yield deterministic outputs, whereas invalid inputs trigger undefined behaviors (UB), resulting in highly divergent execution outcomes. By detecting test cases that induce high output entropy across the sample population, we can identify and prune invalid inputs. This refined test suite, in turn, eliminates evaluation noise, allowing for a more accurate assessment of the code samples' semantic consistency. The mechanism consists of two filtering steps, with algorithm implementation detailed in Appendix D.

1. *Filtering by Error Rate:* Pruning tests that cause widespread execution failures.

2. *Filtering by Output Diversity:* Pruning tests that yield high semantic entropy (indicative of invalid inputs triggering UB).

3.4 Testing Phase

When the abstention rules are obtained on the calibration set, in the testing phase, these rules are applied to make the abstention decisions. As outlined in Alg.1, the process begins by generating n code samples Y and test cases T for the input prompt x. Crucially, CODEREFUSER applies the STDF mechanism (Line 3) to purge invalid test cases using thresholds $[\lambda_1, \lambda_2, \lambda_3]$. The code samples are then partitioned into semantic clusters $\mathcal{C}$ based on their execution outputs on the remaining valid test cases. The final abstention decision depends on the selected scoring mode.

Cluster Ratio (CR). This mode enforces model consensus. CODEREFUSER filters the semantic clusters, retaining only those that account for a sufficient proportion of the total samples (i.e., $|C_i|/|Y| \geq \lambda_{score}$). If no cluster meets this confidence threshold (i.e., the filtered set Y becomes empty), the model implies a lack of consensus and *abstains*.

Semantic Entropy (SE). This mode limits uncertainty. CODEREFUSER calculates the semantic entropy of the cluster distribution. If the entropy exceeds λ_{score}, indicating high semantic diversity and confusion, the model *abstains*. If the task satisfies the criterion of the chosen mode, it is *admitted*.

Algorithm 1 CODEREFUSER Inference Procedure

Input: LLM $\mathcal{M}$; prompt x; samples n; calibrated thresholds $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \lambda_3, \lambda_{score}]$
Output: Abstention decision

1: **procedure** CODEREFUSER($\mathcal{M}, x, n, \boldsymbol{\lambda}$)
2: $Y, T \leftarrow$ Generate$(\mathcal{M}, n)$
3: $T \leftarrow$ STDF$(Y, T, [\lambda_1, \lambda_2, \lambda_3])$
4: $\mathcal{C} \leftarrow$ Clustering Y by Exec(Y, T)
5: **if** Mode is CR **then**
6: $Y \leftarrow \bigcup\{C_i \in \mathcal{C} : |C_i|/|Y| \geq \lambda_{score}\}$
7: **if** $Y = \emptyset$ **then return** *Abstain*
8: **end if**
9: **end if**
10: **if** Mode is SE **then**
11: **if** SE$(\mathcal{C}) > \lambda_{score}$ **then return** *Abstain*
12: **end if**
13: **return** *Admit*
14: **end procedure**

4 Experiments

In this section, we present the empirical evaluation of CODEREFUSER, focusing on two primary dimensions. First, we assess the ***effectiveness*** of our approach on the task abstention problem by benchmarking it against existing counterparts. Second, we examine the ***theoretical guarantee*** to verify whether the risk control promised by the calibration phase is empirically supported.

4.1 Experimental Setup

Datasets. Our experiments are conducted on two standard Python code generation datasets: *HumanEval* (Chen et al., 2021) and *MBPP* (Austin et al., 2021). HumanEval consists of 164 handcrafted programming problems. Each problem is associated with $5-10$ oracle test cases. MBPP is a larger dataset of nearly 1,000 problems. Each task is defined by a short description and includes three test cases to verify its correctness.

Evaluation Metrics. We use *Precision* and *F1-score* to evaluate the performance on the task abstention problem. To obtain the ground-truth label, we generate 256 samples for each task, and checks them the on the oracle test cases. The abstention label is ture if the $H@k$ is above the given risk tolerance. We deliberately avoid using *Recall* as a primary metric due to the following fact. A high recall value can be easily obtained for an abstention rule, by simply refusing most tasks. This is misleading, as it has little practical utility.

Baselines. Since no prior work directly addresses task abstention for code generation, we adapt the following baselines for comparison.

- *Execution.* This method directly calculates the $H@k$ score using generated test cases and abstains if it exceeds the risk tolerance threshold. It can be seen as an execution-based baseline without adopting the proposed framework.
- *PPL* (Huang et al., 2025b) and *NLI* (Manakul et al., 2023). We integrate these metrics into our LTT framework as alternative score functions. *PPL* uses model perplexity (static), while *NLI* queries the LLM for self-consistency checking.
- *CLM* (Quach et al., 2023). This method is originally proposed for natural language generation. We adapt it to our code generation context, and refuse the task if the calibrated valid response set is empty.[3]
- *CodeHalu* (Tian et al., 2025). This is an execution-based sample-level hallucination detector relying on oracle tests. We adapt it by employing LLM-generated tests as pseudo-oracles, and applying the algorithm to identify valid samples and aggregate these predictions to estimate the task-level $H@k$ for the abstention decision.

[3]We use the max version of CLM as it demonstrates the best empirical performance.

Implementation Details. We conduct the experiments with Python 3.9.19, Pytorch 2.4.0 and vLLM 0.6.1, running on NVIDIA H800 GPUs with CUDA 12.7. We evaluate the performance of the following four code LLMs: Deepseek-Coder-33B (Guo et al., 2024), Qwen2.5-Coder-32B (Hui et al., 2024), CodeLlama-7B (Roziere et al., 2023) and WizardCoder-33B-V1.1 (Luo et al., 2024). For all these code LLMs, we use a sampling temperature of 0.8 and a top-p of 0.95. For each dataset, we randomly allocate 60% of the data for calibration, and use the rest for testing. We set $\alpha = 0.2, \delta = 0.1, k = 3$ by default.

4.2 Effectiveness Results

For the score function computation in both the calibration and testing phases, we generate 64 code samples and 64 test cases for each problem. The results on the HumanEval and MBPP benchmarks are presented in Table 1, which are the average results of three independent runs. In the table, 'SE' stands for CODEREFUSER with semantic entropy, 'CR' stands for CODEREFUSER with cluster ratio, and 'SE+STDF' and 'CR+STDF' stand for the version when the STDF mechanism is included.

(1) The quality of generated tests matters. The poor performance of both *Execution* and *CodeHalu* demonstrates that LLM-generated tests cannot reliably replace oracle test suites. *Execution* fails to accurately estimate the pass rate ($H@k$), while *CodeHalu*'s performance degrades significantly when forced to treat potentially flawed generated tests as ground truth. These results confirm a critical quality gap, indicating that naive reliance on raw generated tests without rigorous filtering, is insufficient for effective abstention. In contrast, when the STDF mechanism is integrated, CODEREFUSER consistently yields the best performance across benchmarks. Compared with the best existing results, CODEREFUSER achieves an average of 26.5% and 11.9% absolute improvements w.r.t. abstention precision and F1-score, respectively.

(2) Static score functions are less effective in code generation tasks. Among the baselines, *PPL*, *CLM*, and *NLI* are static methods proposed for natural language generation, without actually running the code. The experimental result shows that these methods consistently underperform compared to our execution-based methods (the four bottom rows in the table). This confirms that traditional NLP metrics fail to capture the semantic correctness of

Table 1: Task abstention results on HumanEval and MBPP. Our approaches (i.e., 'SE+STDF' and 'CR+STDF') generally outperform the competitors. Similarly, when the calibration is conducted on MBPP and the testing is conducted on HumanEval ('MBPP→HumanEval'), our approaches still outperform the competitors and even achieve close performance to the in-distribution case in most cases.

Method \ Model	DeepSeek-Coder		Qwen2.5-Coder		CodeLlama		WizardCoder	
	P	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**
HumanEval								
Execution	39.28	55.27	24.29	38.23	72.15	83.82	35.46	51.55
CodeHalu	38.56	55.66	22.76	36.84	73.10	82.88	33.11	49.01
PPL	36.41	53.39	22.22	21.43	72.29	81.67	40.90	37.11
NLI	49.25	52.38	17.68	30.05	77.92	62.82	31.70	48.15
CLM	36.41	53.39	22.22	21.42	71.27	64.42	37.71	51.49
SE	61.19	65.07	38.23	41.67	85.95	88.51	52.63	62.01
CR	60.52	68.14	44.44	49.23	89.47	89.47	47.99	56.00
SE + STDF	63.63	66.67	47.36	50.91	**91.67**	89.19	**62.71**	**66.67**
CR + STDF	**72.00**	**69.92**	**73.33**	**53.85**	**91.67**	**91.70**	61.40	66.14
MBPP								
Execution	41.31	57.42	31.50	46.11	58.35	73.01	36.08	51.75
CodeHalu	40.48	57.08	30.21	45.32	57.33	72.60	32.70	48.60
PPL	37.16	51.11	39.28	34.55	71.24	55.47	38.46	34.33
NLI	39.40	49.88	30.50	39.61	67.95	58.29	33.13	36.98
CLM	36.32	53.28	31.60	44.25	56.60	72.28	35.10	50.00
SE	64.70	64.61	40.74	45.65	71.77	79.02	60.16	61.26
CR	61.99	66.28	40.00	45.58	67.24	65.67	62.99	62.50
SE + STDF	**72.65**	**72.88**	48.41	**52.84**	72.70	76.69	60.26	**64.70**
CR + STDF	66.43	66.48	**48.94**	47.44	**79.40**	**79.69**	**69.23**	63.07
MBPP → HumanEval								
SE	60.27	62.11	32.65	41.50	76.03	78.63	60.00	58.99
CR	63.01	65.08	32.65	37.68	79.79	79.48	57.41	56.88
SE + STDF	**71.11**	**70.00**	39.39	**50.00**	80.28	77.27	68.42	59.65
CR + STDF	66.67	66.67	**40.63**	40.00	**91.51**	**91.23**	**72.10**	**62.39**

Table 2: Performance comparison when significant more samples are used for static methods ($N = 256$ vs. $N = 64$). Static methods fail to narrow the performance gap even with $4\times$ sample budget.

Method	DeepSeek-Coder		Qwen2.5-Coder		WizardCoder	
	P	**F1**	**P**	**F1**	**P**	**F1**
PPL ($N = 256$)	37.02	54.36	17.79	30.21	39.42	52.22
CLM ($N = 256$)	36.41	53.39	20.32	32.89	36.29	51.64
CODEREFUSER ($N = 64$)	**72.00**	**69.92**	**73.33**	**53.85**	**61.40**	**66.14**

code, reinforcing the necessity of execution for reliable risk estimation.

One might argue that execution-based approaches inherently incur higher computational costs than static baselines. To demonstrate its necessity, we conducted an experiment where we significantly increased the sampling budget for the static baselines. Specifically, we allowed PPL and CLM to generate $N = 256$ samples, while keeping CODEREFUSER (CR+STDF) with the original $N = 64$ samples. The results are summarized in Table 2, where we use HumanEval and three code LLMs for simplicity. The result reveals that even with quadrupled samples, the performance of static methods remains significantly lower than CODEREFUSER. This confirms that the limitations of static methods stem from their inability to capture semantic correctness, a fundamental deficit that cannot be overcome simply by scaling up the sample size.

(3) CODEREFUSER *transfers effectively across different datasets.* We also show in Table 1 the results when the calibration is conducted on MBPP and the testing is conducted on HumanEval. Despite the distribution shift, CODEREFUSER maintains robust performance, consistently outperforming baselines even when they are evaluated in-distribution (compared with results of 'HumanEval'). This demonstrates that the calibration results of CODEREFUSER exhibit transferability across different datasets.

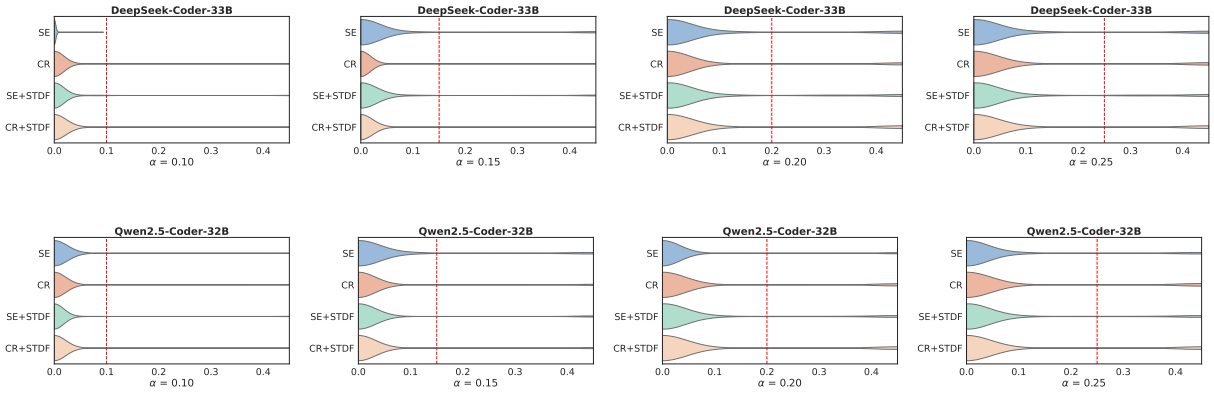

Figure 2: Admission risk distribution on HumanEval under different risk tolerance α. The thickness of each plot corresponds to the density of tasks at that risk level. Most of the admission risks are controlled under the given tolerance (the red dashed line).

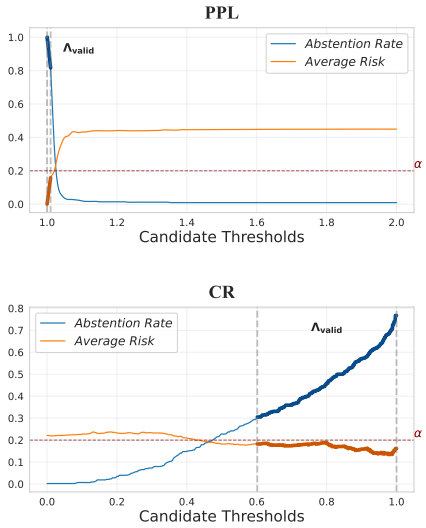

Figure 3: The trade-off between abstention rate and admission risk for Qwen2.5-Coder-32B on MBPP. While both PPL (top figure) and our 'CR' method (bottom figure) control the risk (within the valid range marked by the thick lines), PPL requires a substantially higher abstention rate (80% vs 30%).

4.3 Risk Control Guarantee

We next verify the theoretical risk control guarantee. We focus on the in-distribution setting (calibrated and tested on HumanEval) using DeepSeek-Coder and Qwen2.5-Coder. For each tolerance level α (with fixed $\delta = 0.1, k = 3$), we select the valid threshold $\lambda \in \Lambda_{valid}$ that yields the lowest abstention rate. Fig. 2 illustrates the distribution of the resulting admission risks. The results confirm that the empirical risk is reliably controlled below the target α. Similar results are observed in the transferability setting (calibrated on MBPP and tested on HumanEval), as detailed in Appendix E.1. In summary, CODEREFUSER successfully meets the theoretical guarantees. It remains effective in over 90% cases in terms of keeping the risk below the desired tolerance level (the area on the left side of the red tolerance line accounts for more than 90% of the total area).

Note that one method may simply refuse most code generation tasks to keep the admission risk low. Here, we also analyze Λ_{valid} to understand the trade-off between admission risk and abstention rate. We set the tolerance $\alpha = 0.2$, and compare the results of CODEREFUSER with cluster ratio[4] (i.e., 'CR') and *PPL*. The results of Qwen-Coder-32B on the MBPP dataset are shown in Fig. 3. It can be observed that the admission risk is under control for both methods. However, *PPL* must refuse a large number of tasks (over 80%) to meet the risk tolerance; in contrast, our method only needs to refuse 30% of tasks.

5 Conclusion

In this paper, we introduce and study the task abstention problem for LLM-based code generation, i.e., determining whether a given LLM should abstain from performing a specific code generation task to avoid potential hallucination. Our approach features a calibrated abstention rule, grounded in the principles of multiple hypothesis testing. A distinguished advantage is that it provides a rigorous, distribution-free theoretical guarantee on its abstention decisions, whose effectiveness is also confirmed by the experiments. Our work represents progress in the pursuit of *provably correct* LLM code generation.

[4]We choose 'CR' in this experiment for interpretation, as it has only one parameter.

Limitations

Generalization to out-of-distribution tasks. Our theoretical guarantee relies on the assumption that the calibration and test data are independent and identically distributed. However, in real-world scenarios, the problem a system encounters may differ from those it sees during calibration. This potential distribution shift could introduce bias and weaken the risk control guarantees in practice. To mitigate this threat, we have conducted experiments using MBPP as calibration data and HumanEval as test data, and the results show that there was no significant decline in performance.

Applicability to non-deterministic systems. The core of our approach–both the score functions and the STDF mechanism—assumes that all tasks have deterministic solutions. The assumption may not hold for complex, interactive software systems, which may exhibit nondeterminism. As our method relies on clustering identical outputs, its applicability is currently limited to deterministic problem domains.

Evaluation limited to a single programming language. Current experiments are conducted in Python, a language where modern LLMs have demonstrated strong performance (HUANG et al., 2024; Twist et al., 2025). However, many real-world systems are built using other languages such as Java or C, which may have different characteristics (e.g., pointer and memory management in C). The effectiveness of our method in these programming languages requires further evaluation.

References

Yasin Abbasi Yadkori, Ilja Kuzborskij, András György, and Csaba Szepesvari. 2024. To believe or not to believe your llm: Iterative prompting for estimating epistemic uncertainty. *Advances in Neural Information Processing Systems*, 37:58077–58117.

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

Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. 2023. Monitor-guided decoding of code LMs with static analysis of repository context. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Gustaf Ahdritz, Tian Qin, Nikhil Vyas, Boaz Barak, and Benjamin L. Edelman. 2024. Distinguishing the knowable from the unknowable with language models. ICML'24. JMLR.org.

Anastasios N Angelopoulos, Stephen Bates, Emmanuel J Candès, Michael I Jordan, and Lihua Lei. 2025. Learn then test: Calibrating predictive algorithms to achieve risk control. *The Annals of Applied Statistics*, 19(2):1641–1662.

Anastasios Nikolas Angelopoulos, Stephen Bates, Adam Fisch, Lihua Lei, and Tal Schuster. 2024. Conformal risk control. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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*.

Rina Foygel Barber, Emmanuel J. Candès, Aaditya Ramdas, and Ryan J. Tibshirani. 2023. Conformal prediction beyond exchangeability. *The Annals of Statistics*, 51(2):816 – 845.

Evan Becker and Stefano Soatto. 2024. Cycles of thought: Measuring LLM confidence through stable explanations. *arXiv preprint arXiv:2406.03441*.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(1):993–1022.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, and 1 others. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712*.

Lang Cao. 2024. Learn to refuse: Making large language models more controllable and reliable through knowledge scope limitation and refusal mechanism. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3628–3646, Miami, Florida, USA. Association for Computational Linguistics.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Softw. Eng.*, 49(7):3675–3691.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *Preprint*, arXiv:2207.10397.

Bocheng Chen, Advait Paliwal, and Qiben Yan. 2023. Jailbreaker in jail: Moving target defense for large language models. In *Proceedings of the 10th ACM Workshop on Moving Target Defense*, MTD '23, page

29–32, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and 1 others. 2021. Evaluating Large Language Models Trained on Code. *Preprint*, arXiv:2107.03374.

Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. 1998. Evaluation metrics for language models.

Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. 2024. Universal self-consistency for large language models. In *ICML 2024 Workshop on In-Context Learning*.

Furui Cheng, Vilém Zouhar, Simran Arora, Mrinmaya Sachan, Hendrik Strobelt, and Mennatallah El-Assady. 2024. Relic: Investigating large language model responses using self-consistency. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–18.

Jinhao Duan, Hao Cheng, Shiqi Wang, Alex Zavalny, Chenan Wang, Renjing Xu, Bhavya Kailkhura, and Kaidi Xu. 2024. Shifting attention to relevance: Towards the predictive uncertainty quantification of free-form large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5050–5063.

Aryaz Eghbali and Michael Pradel. 2024. Dehallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding. *arXiv preprint arXiv:2401.01701*.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023a. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023b. Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1469–1481. IEEE Press.

Shangbin Feng, Weijia Shi, Yike Wang, Wenxuan Ding, Vidhisha Balachandran, and Yulia Tsvetkov. 2024. Don't hallucinate, abstain: Identifying LLM knowledge gaps via multi-LLM collaboration. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14664–14690, Bangkok, Thailand. Association for Computational Linguistics.

Jiahui Geng, Fengyu Cai, Yuxia Wang, Heinz Koeppl, Preslav Nakov, and Iryna Gurevych. 2024. A survey of confidence estimation and calibration in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6577–6595.

Isaac Gibbs and Emmanuel Candes. 2021. Adaptive conformal inference under distribution shift. In *Advances in Neural Information Processing Systems*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025a. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Yuheng Huang, Jiayang Song, Zhijie Wang, Shengming Zhao, Huaming Chen, Felix Juefei-Xu, and Lei Ma. 2025b. Look before you leap: An exploratory study of uncertainty analysis for large language models. *IEEE Transactions on Software Engineering*.

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*.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, and 1 others. 2022. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*.

Minsu Kim and James Thorne. 2024. Epistemology of language models: Do language models have holistic knowledge? In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12644–12669, Bangkok, Thailand. Association for Computational Linguistics.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. In *The Eleventh International Conference on Learning Representations*.

Yunseo Lee, John Youngeun Song, Dongsun Kim, Jindae Kim, Mijung Kim, and Jaechang Nam. 2025. Hallucination by code generation llms: Taxonomy,

benchmarks, mitigation, and challenges. *arXiv preprint arXiv:2504.20799*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. Teaching models to express their uncertainty in words. *Transactions on Machine Learning Research*.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024a. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024b. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, 33(5).

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Potsawee Manakul, Adian Liusie, and Mark Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proc. ACM Softw. Eng.*, 1(FSE).

Ella Neeman, Roee Aharoni, Or Honovich, Leshem Choshen, Idan Szpektor, and Omri Abend. 2023. DisentQA: Disentangling parametric and contextual knowledge with counterfactual question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10056–10070, Toronto, Canada. Association for Computational Linguistics.

Victor Quach, Adam Fisch, Tal Schuster, Adam Yala, Jae Ho Sohn, Tommi S Jaakkola, and Regina Barzilay. 2023. Conformal language modeling. In *The Twelfth International Conference on Learning Representations*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Vaishnavi Shrivastava, Percy Liang, and Ananya Kumar. 2024. Llamas know what GPTs don't show: Surrogate models for selective classification.

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D Manning. 2023. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25300–25308.

Ryan J. Tibshirani, Rina Foygel Barber, Emmanuel J. Candès, and Aaditya Ramdas. 2019. *Conformal prediction under covariate shift*. Curran Associates Inc., Red Hook, NY, USA.

Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. 2025. Llms love python: A study of llms' bias for programming languages and libraries. *arXiv preprint arXiv:2503.17181*.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. Syncode: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*.

Neeraj Varshney, Pavel Dolin, Agastya Seth, and Chitta Baral. 2024. The art of defending: A systematic evaluation and analysis of LLM defense strategies on safety and over-defensiveness. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13111–13128, Bangkok, Thailand. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of the 11th International Conference on Learning Representations*.

Yuxia Wang, Haonan Li, Xudong Han, Preslav Nakov, and Timothy Baldwin. 2024. Do-not-answer: Evaluating safeguards in LLMs. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 896–911, St. Julian's, Malta. Association for Computational Linguistics.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards understanding the characteristics of code generation errors made by large language models. *Preprint*, arXiv:2406.08731.

Bingbing Wen, Jihan Yao, Shangbin Feng, Chenjun Xu, Yulia Tsvetkov, Bill Howe, and Lucy Lu Wang. 2025. Know your limits: A survey of abstention in large language models. *Transactions of the Association for Computational Linguistics*, 13:529–556.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, YIFEI LI, Jie Fu, Junxian He, and Bryan Hooi. 2024. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms. In *The Twelfth International Conference on Learning Representations*.

Yuqing Yang, Ethan Chern, Xipeng Qiu, Graham Neubig, and Pengfei Liu. 2024. Alignment for honesty. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Hanning Zhang, Shizhe Diao, Yong Lin, Yi Fung, Qing Lian, Xingyao Wang, Yangyi Chen, Heng Ji, and Tong Zhang. 2024. R-tuning: Instructing large language models to say 'I don't know'. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7113–7139, Mexico City, Mexico. Association for Computational Linguistics.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. 2024. Large language models are not robust multiple choice selectors. In *The Twelfth International Conference on Learning Representations*.

A Related Work

Code Hallucination. The phenomenon of code hallucination, where LLMs generate code that is illogical, incorrect, or unfaithful to user requirements (Fan et al., 2023a), presents a challenge to ensure the accuracy, reliability and security of AI-generated code (Agarwal et al., 2024; Eghbali and Pradel, 2024; Tian et al., 2025). Existing research has explored code hallucination from several angles, often categorizing failures based on when they occur. These categories primarily include *syntactic hallucination*, which are errors that violate the programming language's syntax and prevent code from being compiled or interpreted (Agrawal et al., 2023; Fan et al., 2023b; Wang et al., 2025); *runtime hallucination*, where the code is syntactically valid but produces errors such as exceptions or crashes (Fan et al., 2023b; Liu et al., 2024a; Tian et al., 2025; Zhang et al., 2025), and *functional hallucination*, where code that runs without error fails to meet the program's intended requirements (Fan et al., 2023b; Liu et al., 2024a; Wang et al., 2025; Zhang et al., 2025), among others.

To facilitate a more rigorous evaluation of hallucination in LLM-based code generation, a number of benchmarks have been developed. Notable examples include CodeHaluEval (Tian et al., 2025), CodeMirage (Agarwal et al., 2024), MultiPL-E (Cassano et al., 2023), HalluCode (Liu et al., 2024a), etc. Prior work has also proposed various strategies to mitigate code hallucination. The De-hallucinator (Eghbali and Pradel, 2024) pre-indexes a project's codebase and uses Retrieval-Augmented Generation (RAG) to inject relevant APIs into prompts; Liu et al. (Liu et al., 2024b) leverage the LLM's self-revision capabilities by providing it with feedback based on static analysis; SynCode (Ugare et al., 2025) uses a formal grammar representation (EBNF) to guide the model's decoding and ensure syntactic validity; ClarifyGPT (Mu et al., 2024) introduced a framework where the LLM proactively asks clarifying questions to help users refine their initial prompts. Different from the above work that focuses on the sample-level hallucination problem, we study the task abstention problem.

LLM Abstention. Abstention is increasingly recognized for its potential to mitigate hallucination and enhance safety in LLM systems (Wen et al., 2025; Varshney et al., 2024; Wang et al., 2024; Zhang et al., 2024). A guiding principle is that

a system should abstain when it is insufficiently confident in the correctness of its output or if there is a high probability of error (Ahdritz et al., 2024; Kim and Thorne, 2024; Cao, 2024). Existing work has proposed various strategies to determine when the LLM should abstain, targeting different stages of the model lifecycle.

During the training and alignment phase, Neeman et al. (Neeman et al., 2023) use data augmentation to fine-tune models to recognize unanswerable questions. Yang et al. (Yang et al., 2024) construct "honesty" alignment datasets by substituting a model's incorrect response with "I don't know" and then fine-tuning on this revised data. At inference time, a common approach is to use post-processing techniques based on model uncertainty. These include calculating the log probability of a 'True' token via indirect logit methods (Lin et al., 2022; Tian et al., 2023), using a surrogate LLM to approximate the confidence of a black-box model (Shrivastava et al., 2024), or assessing the semantic entropy of responses (Kuhn et al., 2023). A different inference-time strategy involves LLM collaboration, where a second "test" LLM is employed to examine the output of the first, helping identify harmful queries or correct the initial response before it is shown to the users (Feng et al., 2024; Chen et al., 2023). The existing methods are mainly proposed for the natural language generation problem. In this work, we argue that test code generation and execution are essential for the code generation task abstention problem.

Conformal Risk Control. Approaches based on Conformal Prediction (CP) are known for their ability to select a guaranteed threshold by analyzing the Guantile distribution of risk term on a calibration set (Tibshirani et al., 2019; Barber et al., 2023; Gibbs and Candes, 2021). However, standard CP methods are often constrained to operating on a single candidate threshold at a time. They are not equipped to handle a *vector* of thresholds derived simultaneously from multiple score functions, which limits their applicability in scenarios requiring multifaceted evaluation.

A related framework, Conformal Risk Control (CRC), also efficiently utilizes calibration sets to manage the prediction risk (Angelopoulos et al., 2024). Its primary limitation, however, is the requirement of a monotonic relationship between the risk function and the threshold. This monotonicity assumption frequently does not hold for the intricate score functions used in complex generative tasks, posing a significant challenge to adapting CRC to our domain.

Confidence Estimation for LLMs. Confidence estimation for LLMs aims to provide a measure of predictive uncertainty for LLM outputs. Well-calibrated confidence can further contribute to migrating the bias and alleviating the hallucination (Geng et al., 2024; Zheng et al., 2024; Bubeck et al., 2023). Recent research on LLM confidence estimation broadly falls into the following veins. *Perplexity confidence* (Huang et al., 2025b; Duan et al., 2024) derives confidence from the probabilities assigned to generated tokens, employing the geometric mean (i.e., perplexity (Chen et al., 1998; Blei et al., 2003)) to mitigate sensitivity to output length; *verbalized confidence* (Kadavath et al., 2022; Xiong et al., 2024; Tian et al., 2023) directly prompts the LLM to explicitly express its confidence alongside its answer (e.g., "Read the question and give your answer and corresponding confidence score"); *self-consistency confidence* (Xiong et al., 2024; Abbasi Yadkori et al., 2024; Becker and Soatto, 2024) assesses confidence by having the LLM generate multiple answers for the same input and then measuring the consistency among them (Wang et al., 2022; Chen et al., 2024; Cheng et al., 2024), with higher consistency indicating greater confidence.

B The LTT Framework

The Learn Then Test (LTT) framework (Angelopoulos et al., 2025) is designed to provide statistical guarantees for machine learning models by simply adding a post-processing step on a calibration set after the model is trained. Consider the task where each instance $x \in \mathcal{X}$ is associated with a ground-truth label $y \in \mathcal{Y}$. Let $\mathcal{D}_{cal} = \{(x_i, y_i)\}_{i=1}^{m} \subseteq \mathcal{X} \times \mathcal{Y}$ be a calibration set composed of the input x and its ground-truth label y, which are independently and identically distributed (i.i.d.) drawn. A post-processing function $\mathcal{T}_\lambda : \mathcal{X} \to \mathcal{Y}'$ with parameter λ is designed to map $\mathcal{X}$ to any space $\mathcal{Y}'$. (For instance, in classification $\mathcal{Y}'$ may be defined as $2^{\mathcal{Y}}$, i.e., all possible subsets of $\mathcal{Y}$.) In other words, instead of predicting a label for each instance, LTT aims to predict a subset of labels so that the true label is within the subset with a high probability. The choice of the subsets is decided by λ (e.g., labels with predictive probability greater than λ are included in the

subset).

Based on the post-processing function $\mathcal{T}_\lambda$, a *risk* $R(\mathcal{T}_\lambda(x)) \in \mathbb{R}$ on a given x can be defined to measure the task-specific statistical error (e.g., the miscoverage rate of true labels in $\mathcal{Y}'$ for the classification task). Since the risk is mainly decided by parameter λ, we rewrite $R(\mathcal{T}_\lambda(x))$ as $R(x;\lambda)$ for brevity.

The goal of LTT is to ensure the guarantee as stated in Eq. (2). Using the classification task as an example, intuitively Eq. (2) asserts that the risk of a wrong classification (e.g., the true label is not in the output subset) is below the threshold α with probability at most δ.

The core of LTT is to obtain the set Λ_{valid}. For this purpose, we can traverse all the plausible $\{\mathcal{T}_\lambda\}_{\lambda\in\Lambda}$ and estimate their risk on the calibration set $\mathcal{D}_{cal}$ using multiple hypothesis testing. Specifically, for each λ_j in a discrete set $\Lambda = \{\lambda_1, \cdots, \lambda_N\}$, we have a null hypothesis $\mathcal{H}_j : R(\lambda_j) > \alpha$ and $\mathcal{H}_j$ is rejected when λ_j controls the risk, i.e., $\lambda_j \in \Lambda_{valid}$. For each null hypothesis, we can compute a finite-sample valid p-value using a concentration inequality (Angelopoulos et al., 2025). Λ_{valid} can then be calculated by applying any *family-wise error rate* (FWER)-controlling algorithm, which receives the set of p-values and returns the set of λ that we should reject the associated null hypothesis. For example, the Bonferroni correction is a typical FWER-controlling algorithm which yields $\Lambda_{valid} = \left\{\lambda_j : p_j \leq \frac{\delta}{|\Lambda|}\right\}$.

Once Λ_{valid} is obtained on the calibration set, LTT ensures that the following theorem holds for an i.i.d. test sample x_{test}.

Theorem 1 (Learn Then Test (Angelopoulos et al., 2025))**.** *Suppose p_j is super-uniform for all j under $\mathcal{H}_j$, and assume a valid FWER-controlling algorithm at level δ. Then Eq. (2) holds for the test sample x_{test}.*

Key insight of applying LTT. The key insights of using LTT in our task abstention problem for LLM-based code generation are as follows. LTT was originally proposed to generate a set of responses, instead of a single response, so that the true response is within the response set with a guaranteed probability. In code generation, the LLM can refuse the code generation task if the response set is empty after a few attempts. Meanwhile, the statistical guarantee from LTT still stands.

C Score Function Definitions

In Section 3.3, we introduced two score functions based on execution-based clustering. Here, we provide their formal definitions. Let $Y = \{y_1, \ldots, y_n\}$ be the generated code samples and $\mathcal{C} = \{C_1, \ldots, C_h\}$ be the partition of Y derived from execution results on test cases T.

C.1 Confidence-based Score Function

The consistency function $\mathbb{I}(y_1, y_2; \mathcal{C})$ between two code samples is defined as:

$$\mathbb{I}(y_1, y_2; \mathcal{C}) = \begin{cases} 1, & \exists C_j \in \mathcal{C}, \{y_1, y_2\} \subseteq C_j \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

The confidence score for a sample $y \in Y$ is the average consistency with all other samples:

$$Conf(y) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(y, y_i; \mathcal{C}). \tag{7}$$

A high confidence score indicates that the sample belongs to a dominant semantic cluster, reflecting the model's conviction in that specific logic (Chen et al., 2022).

C.2 Semantic Entropy-based Score Function

Semantic entropy measures the diversity of the semantic clusters. For a task x, it is defined as:

$$\begin{aligned} SE(x) &= -\sum_{C_i \in \mathcal{C}} p(C_i \mid x) \log p(C_i \mid x) \\ &\approx -|\mathcal{C}|^{-1} \sum_{C_i \in \mathcal{C}} \log p(C_i \mid x). \end{aligned} \tag{8}$$

A higher entropy value signifies greater inconsistency among the generated solutions, suggesting a higher likelihood of task-level hallucination (Kuhn et al., 2023).

D Implementation Details of STDF

In this section, we provide the formal procedure for the Sample-Test Dual Filtering (STDF) mechanism, as outlined in Algorithm 2.

The algorithm requires three calibrated thresholds: λ_1 (maximum error rate tolerance), λ_2 (maximum pruning ratio), and λ_3 (maximum entropy tolerance). The process consists of two phases:

Phase 1: Error Rate Pruning (Lines 4–6). We first calculate the error rate for each test case t across all generated code samples Y. If the proportion of samples that crash or fail on t exceeds

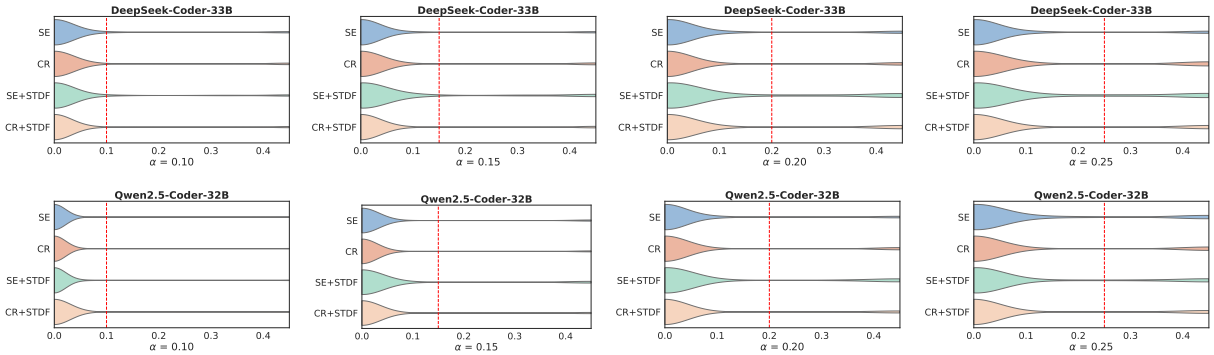

Figure 4: Admission risk distribution on HumanEval under different risk tolerance α, when the LLM is calibrated on MBPP. Most of the admission risks are still under the given tolerance.

λ_1, the test case is immediately discarded. This step effectively removes "toxic" inputs that cause widespread crashes.

Phase 2: Diversity-based Pruning (Lines 8–16). For the surviving test cases, we compute the semantic entropy (SE) of their execution outputs. A high SE implies that the code samples produce a wide variety of inconsistent outputs for this specific input, suggesting the test case is ambiguous or invalid. However, high variance might also stem from the model's own uncertainty rather than the test's flaw. To avoid aggressively discarding valid tests (which would artificially inflate the consensus score), we impose a safety limit: we sort the test cases by their entropy and remove at most $\lfloor \lambda_2 \cdot |T| \rfloor$ test cases, and only if their entropy exceeds λ_3.

E Additional Experimental Analysis

In this section, we provide further analysis to validate the robustness and efficiency of CODEREFUSER. Specifically, we investigate: (1) the transferability of our method under distribution shifts (i.e., cross-dataset evaluation); and (2) the sensitivity of our approach to different risk definitions (i.e., varying k).

E.1 Transferability Analysis

To assess the generalization capability of CODEREFUSER to out-of-distribution tasks, we further examine whether the theoretical risk control holds under distribution shifts. Figure 4 illustrates the distribution of admission risks on HumanEval using thresholds calibrated on MBPP.

Although the rigorous theoretical guarantee relies on the i.i.d. assumption, empirical results show that the admission risk remains largely controlled below the target tolerance α. This indicates that CODEREFUSER is practically robust to moderate distribution shifts.

Algorithm 2 Sample-Test Dual Filtering Mechanism

Input: Generated code samples Y; generated test cases T; thresholds $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \lambda_3]$
Output: The refined test cases T'

procedure STDF($Y, T, \boldsymbol{\lambda}$)
  $res \leftarrow \text{Exec}(Y, T)$
  $FilterSet \leftarrow \emptyset$
  for each test case $t \in T$ **do**
    if ErrorRate($res[t]$) $> \lambda_1$ **then**
      Erase t from T
    else
      $C_t \leftarrow$ Clustering Y by $res[t]$
      add $\{t, \text{SE}(C_t)\}$ to $FilterSet$
    end if
  end for
  sort $FilterSet$ in descending order of SE
  $max_{num} \leftarrow \lfloor \lambda_2 \cdot |T| \rfloor$
  for t, SE_t in top max_{num} elements of $FilterSet$ **do**
    if $SE_t > \lambda_3$ **then**
      Erase t from T
    end if
  end for
  return T
end procedure

E.2 Sensitivity to Risk Definition ($k = 5$)

In the main experiments, we defined the risk based on the pass rate $H@k$ with $k = 3$. To evaluate the sensitivity of CODEREFUSER to this hyperparameter, we conducted additional experiments setting $k = 5$. This adjustment implies a slightly more relaxed criterion for success, effectively allowing the model more attempts to yield a correct solution.

We re-calibrated and evaluated CODEREFUSER

Table 3: Task abstention results on HumanEval and MBPP with $k = 5$. Our approaches (i.e., 'SE+STDF' and 'CR+STDF') generally outperform the competitors.

Method \ Model	DeepSeek-Coder		Qwen2.5-Coder		WizardCoder	
	P	**F1**	**P**	**F1**	**P**	**F1**
			HumanEval			
PPL	36.41	53.39	17.79	30.20	39.42	52.23
CLM	36.97	53.67	20.32	32.89	37.71	51.49
CodeHalu	28.68	42.68	16.09	26.16	27.27	41.25
SE	60.27	66.66	28.57	41.51	52.43	**63.70**
CR	59.49	68.11	41.02	47.06	51.19	62.77
SE + STDF	56.71	64.95	46.66	50.91	53.33	62.50
CR + STDF	**68.75**	**71.54**	**70.33**	**53.90**	**60.00**	61.11
			MBPP			
PPL	36.32	53.28	26.15	41.04	31.75	47.45
CLM	36.32	53.28	31.60	44.25	35.10	50.00
CodeHalu	33.43	48.63	28.12	41.02	32.74	47.32
SE	51.12	64.60	34.80	45.65	50.00	61.26
CR	59.18	66.28	31.95	45.57	60.99	63.70
SE + STDF	**66.13**	**72.88**	46.76	**52.84**	60.26	**64.99**
CR + STDF	59.39	66.47	**47.22**	47.44	**62.59**	63.08

under this new setting. As shown in Table 3, the results exhibit negligible variance compared to the $k = 3$ case. CODEREFUSER continues to robustly identify and abstain from unsolvable tasks, maintaining its performance advantage over other methods. This consistency confirms that the effectiveness of CODEREFUSER is not dependent on a specific choice of k, demonstrating its robustness to different risk definitions.