UNCERTAINTY QUANTIFICATION FOR LLM-BASED CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Prediction sets provide a theoretically grounded framework for quantifying uncertainty in machine learning models. Adapting them to structured generation tasks, in particular, large language model (LLM) based code generation, remains a challenging problem. An existing attempt proposes PAC prediction sets but is limited by its strong monotonicity assumption on risk and single-label classification framework, which severely limits the space of candidate programs and cannot accommodate the multiple valid outputs inherent to code generation. To address these limitations, we propose an approach RISCOSET that leverages multiple hypothesis testing to construct risk-controlling predictions for LLM-based code generation. Given a trained code generation model, we produce a prediction set represented by a partial program, which is guaranteed to contain a correct solution with high confidence. Extensive experiments on three LLMs demonstrate the effectiveness of the proposed method. For instance, compared with the state-of-the-art, our method can significantly reduce the code removal by up to 24.5%, at the same level of risk.

1 INTRODUCTION

Large language models (LLMs) have been widely adopted by millions of users worldwide due to their strong performance (Singh et al., 2025; Team et al., 2026). However, they often hallucinate facts and may generate toxic or biased outputs (Kalai et al., 2025), limiting their usage in a wide range of applications such as code generation and legal analysis (Zhang et al., 2025; Magesh et al., 2025).

This trustworthiness gap is, to a large extent, an uncertainty quantification problem: modern LLMs offer no reliable signal of when their outputs should be trusted (Yao et al., 2023). As an uncertainty quantification method, conformal prediction (CP) addresses this gap by providing model-agnostic, distribution-free uncertainty guarantees for machine learning models, and thus has attracted recent attention (Angelopoulos et al., 2022; Zhou et al., 2025). For example, several studies have applied CP into LLM generation by identifying sound components within the LLM output (Quach et al., 2024; Cherian et al., 2024). However, these methods assume near independence among components, an assumption that holds for sentences in NLP but fails for structured outputs like code, where components interact systemically (Chen et al., 2022; Jiang et al., 2026).

To address this, Khakhar et al. (2023) proposed probably approximately correct (PAC) prediction sets for code generation, constructing structured prediction sets and providing statistical guarantees via partial programs (Guo et al., 2021; Li et al., 2025). However, this approach suffers from two key limitations: (1) it relies on a strong monotonicity constraint that restricts the space of admissible partial programs, which may lead to a suboptimal solution; (2) it is designed for single-label classification and does not easily accommodate the multiple valid programs (e.g., multiple syntactically different programs with identical, correct semantics) that naturally arise in code generation.

In this paper, we introduce a new method, RISCOSET, for constructing risk-controlling prediction sets tailored to code generation. To overcome the limitations in PAC prediction sets, we reframe code uncertainty quantification as a multi-label classification problem, and adopt the Learn Then Test (LTT) framework (Angelopoulos et al., 2025), a recent risk-controlling method, to construct structured prediction sets with rigorous statistical guarantees. That is, given an LLM-generated code, RISCOSET yields a partial program (i.e., a subtree of the program's abstract syntax tree (AST), see Fig. 1), which is guaranteed, with high confidence, to contain a correct solution.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

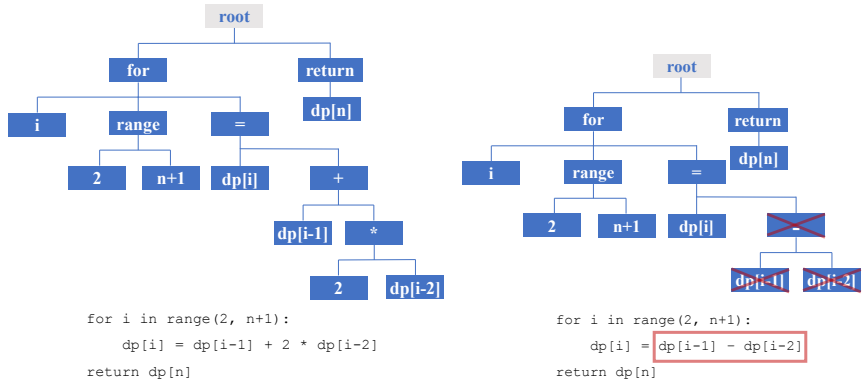


Figure 1: An illustrative example from MBPP. The left part is a correct code snippet, and the right part is a generated but incorrect one. RISCOSET removes three nodes in the AST, resulting in a prediction set (i.e., a partial program) that contains the correct program.

However, applying LTT to code generation tasks is still challenging. The first challenge lies in the complexity of code search space. Unlike traditional applications such as classification and regression whose search space is typically one-dimensional, the search space of structured data (such as code) is complex and multi-dimensional (Gan et al., 2026). To this end, we design a tailored risk function and the corresponding optimization problem for obtaining partial programs. Furthermore, in multi-label settings, we need to sample additional candidate programs from LLMs and verify their correctness via test-case execution during calibration. This is a computationally expensive step. To reduce this cost, we propose a selective execution strategy that efficiently verifies the correctness of sampled programs by using a well-calibrated threshold to decide which programs require test execution.

Example. Fig. 1 illustrates a real example from the MBPP dataset (Austin et al., 2021). The left code snippet is from a correct program, while the right one is generated by the model and is incorrect. Our method, RISCOSET, removes the uncertain nodes in the AST (highlighted in red), resulting in a partial program (AST subtree) which can be completed to match a correct program. This example also shows the potential implication of our method: RISCOSET can provide an almost-correct partial program of a given generated program, helping to mitigate code hallucinations (Liu et al., 2024; Gao et al., 2025) and reduce the cost of manual review (Tian et al., 2025).

Contributions. The main contributions of the paper include: (1) instead of designing a new code generation method, we propose RISCOSET, an uncertainty quantification approach for LLM-based code generation that reframes the construction of prediction sets as a multi-label problem with weaker constraints; (2) we propose a selective execution strategy to balance the trade-off between risk-controlling and sampling costs; (3) we conduct comprehensive experiments across three LLMs and five code generation datasets, confirming our method's effectiveness. To the best of our knowledge, this is the first work to introduce the LTT framework into risk-controlling code generation.

2 BACKGROUND

Notations. For any natural number n, let $[n] := \{1, \ldots, n\}$. We use $\mathcal{D}$ to denote a distribution over $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the label space. Let $\mathcal{D}_{\text{cal}} = \{(X_i, Y_i)\}_{i=1}^{n} \overset{\text{i.i.d.}}{\sim} \mathcal{D}$ denote a calibration set, in which each $X_i \in \mathcal{X}$ is an input and Y_i is its corresponding label. In the single-label case, $Y_i \in \mathcal{Y}$; in the multi-label case, $Y_i \subseteq \mathcal{Y}$. Let $(X_{n+1}, Y_{n+1}) \sim \mathcal{D}$ be a test sample. An abstract syntax tree (AST) of a program is denoted by $T = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of nodes and edges, respectively.

2.1 PAC PREDICTION SET

The objective of a PAC prediction (Park et al., 2020) is to produce a prediction set, instead of a single prediction, that satisfies statistical guarantees while remaining small in size. Formally, for a test input $X_{n+1} \in \mathcal{X}$, PAC prediction set $C_\tau(X_{n+1}) \subseteq \mathcal{Y}$ ensures that the true label Y_{n+1} is contained within

the set for at least a $1-\varepsilon$ fraction of the population, with a probability at least $1-\delta$, i.e.,

$$\mathbb{P}\Big(\mathbb{P}_{(X,Y)\sim\mathcal{D}}\big(Y \in C_\tau(X)\big) \geq 1-\varepsilon\Big) \geq 1-\delta,$$

where $C_\tau(x)$ is parameterized by $\tau \in \mathbb{R}$, and ε and error level δ are chosen in advance by the user.

Technically, the prediction set is constructed by fitting the parameter τ on the calibration set $\mathcal{D}_{\text{cal}}$. Existing methods (Park et al., 2020; Khakhar et al., 2023) rely on the assumption that $\mathbb{P}(Y_{n+1} \in C_\tau(X_{n+1}))$ is monotone-decreasing in τ, which is typically guaranteed by the monotonicity of prediction sets, i.e.,

$$\tau < \tau' \to C_\tau(X_{n+1}) \supseteq C_{\tau'}(X_{n+1}). \tag{1}$$

Therefore, a standard PAC formulation defines $C_\tau(X_{n+1})$ as the set of all labels whose model-assigned score exceeds the threshold τ:

$$C_\tau(X_{n+1}) := \{y \in \mathcal{Y} \mid f(y \mid X_{n+1}) \geq \tau\},$$

where $f(y \mid X_{n+1})$ is a scoring function (e.g., the softmax output of a trained classifier).

2.2 LTT Framework

The Learn Then Test (LTT) framework (Angelopoulos et al., 2025) extends PAC prediction sets by providing control over the expectation of an arbitrary risk function, conditional on the calibration data. It achieves this by reframing the selection of the prediction set parameter as a multiple-hypothesis testing problem. Specifically, for a parameter λ, let $\mathcal{T}_\lambda(x) \subseteq \mathcal{Y}$ denote the prediction set returned for an input $x \in \mathcal{X}$. For a given $\mathcal{T}_\lambda$, we define a corresponding *risk* $R(\mathcal{T}_\lambda) \in \mathbb{R}_{\geq 0}$ that quantifies a problem-specific statistical error rate. The goal of the LTT framework is to learn a parameter $\hat{\lambda}$ such that the function $\mathcal{T}_{\hat{\lambda}}$ satisfies the following error-control guarantee for a test instance (X_{n+1}, Y_{n+1}).

Definition 2.1 (Risk-controlling)**.** A random variable $\hat{\lambda} \in \Lambda$ is risk-controlling at (α, δ)-level, if it satisfies

$$\mathbb{P}(R(\mathcal{T}_{\hat{\lambda}}) \leq \alpha) \geq 1-\delta. \tag{2}$$

The risk tolerance α and error level $\delta \in (0,1)$ are user-specified. For brevity, we abbreviate $R(\lambda) = R(\mathcal{T}_\lambda)$ below. To control target risks, the LTT framework leverages the base model and the calibration set $\mathcal{D}_{\text{cal}}$ to identify valid λ via multiple hypothesis testing. Each $\lambda_k \in \Lambda = \{\lambda_1, \ldots, \lambda_N\}$ is associated with the null hypothesis $\mathcal{H}_k : R(\lambda_k) > \alpha$ and λ_k is considered valid if $\mathcal{H}_k$ is rejected. For each λ_k, a super-uniform p-value p_k is computed. The output of LTT framework is the reject set $\Lambda_{\text{valid}} = \mathcal{A}(p_1, \ldots, p_N) \subseteq \Lambda$, where $\mathcal{A}$ is any algorithm that controls the family-wise error rate (FWER). With these steps, we have the following theorem.

Theorem 2.2 (Learn Then Test (Angelopoulos et al., 2025))**.** *Suppose each p_k has a super-uniform distribution under $\mathcal{H}_k$. Let $\mathcal{A}$ be a FWER-controlling algorithm at level δ. Given a test instance (X_{n+1}, Y_{n+1}), Λ_{valid} satisfies*

$$\mathbb{P}\left(\sup_{\lambda \in \Lambda_{\text{valid}}} R(\lambda) \leq \alpha \mid \mathcal{D}_{\text{cal}}\right) \geq 1-\delta, \tag{3}$$

where the supremum over an empty set is defined as $-\infty$.

This theorem implies that any $\lambda \in \Lambda_{\text{valid}}$ can be selected to control the risk $R(\lambda)$ on test instances. In particular, we recover a procedure similar to the PAC prediction set by defining $\mathcal{T}_\lambda(X_{n+1}) := \{y \in \mathcal{Y} \mid f(y \mid X_{n+1}) \geq \tau\}$ and $R(\lambda) := \mathbf{1}\{Y_{n+1} \notin \mathcal{T}_\lambda(X_{n+1})\}$.

2.3 PAC vs. LTT in Code Generation

Code generation is a typical structured prediction task with an exponentially large label space, rendering conventional conformal prediction intractable (Vovk et al., 2005). To address this, Khakhar et al. (2023) proposed to construct prediction sets via partial programs. Their method represents a program as an AST, denoted as $T = (\mathcal{V}, \mathcal{E})$, and formulates an optimization problem which selectively prunes nodes and edges from T, yielding a partial program $T_p = (\mathcal{V}_p, \mathcal{E}_p)$. The resulting prediction set is defined as the collection of all complete programs that can be obtained by expanding T_p, and

a correct program $T' = (\mathcal{V}', \mathcal{E}')$ belongs to this set iff $\mathcal{V}_p \subseteq \mathcal{V}'$ and $\mathcal{E}_p \subseteq \mathcal{E}'$. This construction enables the application of the PAC prediction set framework to code generation, providing statistical guarantees on program correctness.

However, PAC prediction sets exhibit two limitations when applied to code generation. First, they impose a monotonicity constraint (cf. Eq. (1)) as a compromise to PAC guarantee (Park et al., 2020) and can lead to suboptimal partial programs during optimization. Second, PAC prediction sets are inherently designed for single-label tasks, whereas code generation naturally admits multiple valid outputs (i.e., multiple syntactically different but semantically-identical programs), rendering it a multi-label problem. In contrast, the LTT framework avoids these limitations and is inherently flexible and scalable to multi-label settings. Crucially, LTT does not require defining a scoring function for each program in an exponentially large prediction set; it only requires specifying the prediction set itself and its associated risk without any constraint. We therefore argue that LTT is better suited for quantifying uncertainty in code generation than PAC prediction sets.

3 Approach

We now introduce our method, RisCoSet, for constructing structured, risk-controlling prediction sets for LLM-based code generation. At a high level, RisCoSet defines a structured risk function on code prediction sets via solving an optimization problem. For each calibration program $X_i \in \mathcal{D}_{\text{cal}}$, we sample multiple correct programs to obtain the enhanced label set $\tilde{Y}_i$, according to the corresponding generation task of X_i. Finally, we apply multiple hypothesis testing, following the procedure of LTT, to identify valid parameters Λ_{valid}. These parameters are then used to generate risk-controlling prediction sets for the test program X_{n+1}. Additionally, we propose a selective execution strategy to lower the computational cost of verifying sample programs' correctness, by tolerating a controlled rate of labeling errors.

3.1 Structured Risk Function

Unlike standard multi-label classification problems where the false discovery rate (FDR) is a common control target, our focus is on the semantic correctness of the structured output. Specifically, given an input program X, we require the prediction set $\mathcal{T}_\lambda(X)$ to contain at least one program that belongs to the set $\tilde{Y}$ of correct programs. Formally, we define the empirical structured risk on the enhanced calibration set $\{(X_i, \tilde{Y}_i)\}_{i=1}^n$ as

$$\widehat{R}_n^s(\lambda) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{\mathcal{T}_\lambda(X_i) \cap \tilde{Y}_i = \emptyset\}. \tag{4}$$

Since the label space in code generation is high-dimensional and exponentially large, directly enumerating all complete programs is intractable. Following Khakhar et al. (2023), we construct the prediction set $\mathcal{T}_\lambda(X)$ using partial programs, which provide a compact representation of the candidate program set, as mentioned in Section 2.3.

The remaining problem is to obtain partial programs for X_i accounting for uncertainty in code generation. Given an AST $T = (\mathcal{V}, \mathcal{E})$ and a parameter $\lambda_k \in \Lambda$, we introduce a binary variable $\beta_{i,v}^{(k)}$ for each node $v \in \mathcal{V}$, where $\beta_{i,v}^{(k)} = 1$ indicates that v is removed from T, and $\beta_{i,v}^{(k)} = 0$ otherwise. Let $\pi_v \in \mathbb{R}_{\geq 0}$ denote the negative log probability[1] of v conditioned on its ancestors, i.e., $\pi_v = -\log \mathbb{P}(v \mid v_1, \ldots, v_m)$. Partial programs can be obtained by optimization. For fixed $k \in [N]$ and input program X_i:

$$\min_{\beta} \sum_{v \in \mathcal{V}} \beta_{i,v}^{(k)}, \;\; \text{s.t.} \sum_{v \in \mathcal{V}} \pi_v (1 - \beta_{i,v}^{(k)}) \leq \lambda_k. \tag{5}$$

In this minimization problem, the goal is to retain as many nodes as possible while ensuring the total uncertainty of the partial programs does not exceed λ_k. Moreover, the solution must satisfy two

[1] Sequence models usually ignore AST structure. Heuristically, we use leaf-token probabilities following Khakhar et al. (2023). Since RisCoSet accepts any uncertainty score, this heuristic does not invalidate the coverage guarantee.

additional structure constraints derived from ASTs:

$$\begin{aligned} &\text{(SC1)} \ \ (\beta_{i,v}^{(k)} = 1) \rightarrow (\beta_{i,v'}^{(k)} = 1), \qquad \forall (v, v') \in \mathcal{E}, \\ &\text{(SC2)} \ \ \sum_{(v,v') \in \mathcal{E}} \beta_{i,v'}^{(k)} (1 - \beta_{i,v}^{(k)}) \leq t_{\max}, \end{aligned} \tag{6}$$

where (SC1) enforces tree connectivity and (SC2) imposes an upper-bound $t_{\max} \in \mathbb{N}^+$ on the number of permissible subtree removals.

In contrast to PAC prediction set (Khakhar et al., 2023), our optimization problem for constructing partial programs operates under a less restrictive set of constraints. The most significant relaxation is the removal of the monotonicity condition on prediction sets (cf. Eq.(1)), which substantially enlarges the feasible label space.

3.2 Multiple Hypothesis Testing for Code

With the structured risk defined above, we then employ multiple hypothesis testing to identify the reject set Λ_{valid}. The first step of this procedure is to compute a p-value p_k for each null hypothesis $\mathcal{H}_k$. Since the structured risk defined in Eq.(4) is an average of binary losses, a valid p-value can be computed by the following lemma.

Lemma 3.1 (Binomial trial p-values)**.** *Let $\widehat{R}_n^s(\lambda_k)$ be the empirical structured risk defined in Eq.*(4)*, and let $\mathrm{Bin}(n, \alpha)$ denote a binomial random variable with n trials and success probability α. Then,*

$$p_k^{BT} = e\mathbb{P}\big(\mathrm{Bin}(n, \alpha) \leq n\widehat{R}_n^s(\lambda_k)\big) \tag{7}$$

is a valid p-value for $\mathcal{H}_k : R^s(\lambda_k) > \alpha$.

In LTT, let $\mathcal{A}$ be any FWER-controlling algorithm at level δ. Applying $\mathcal{A}$ to the p-values $\{p_k^{BT}\}_{k=1}^N$ yields a set of rejected hypotheses, whose corresponding configurations form $\Lambda_{\text{valid}} \subseteq \Lambda$. If $\Lambda_{\text{valid}} = \emptyset$, the method abstains (i.e., returns `null`). Otherwise, any parameter in Λ_{valid} can be selected. In practice, we select the one that empirically minimizes the number of node removals. Since the number of removals decreases monotonically with λ, this amounts to choosing

$$\hat{\lambda} = \max(\Lambda_{\text{valid}}). \tag{8}$$

Note that the monotonicity here only affects the number of pruned nodes (i.e., the size of prediction sets), which naturally follows from the optimization objective in Section 3.1. This is distinct from and weaker than the explicit monotonicity constraint on prediction sets imposed in Eq. (1). Consequently, the parameter $\hat{\lambda}$ selected by Eq. (8), based on $\mathcal{D}_{\text{cal}}$, provides a risk-controlling guarantee for the resulting prediction sets.

Theorem 3.2 (Structured risk-controlling sets)**.** *The prediction set $\mathcal{T}_{\hat{\lambda}}(X_{n+1})$, constructed by the partial program from Algorithm 1, satisfies Eq.* (2)*.*

This result ensures that the prediction set $\mathcal{T}_{\hat{\lambda}}(X_{n+1})$ constructed by the selected partial programs is risk-controlling, while Algorithm 1 in Appendix B describes the overall procedure for obtaining these partial programs. To precisely search and test Λ in the context of code generation, in Appendix C, we discuss three alternative FWER control algorithms: *Bonferroni method*, *Holm–Bonferroni method* (Holm, 1979), and *fixed sequence testing* (Bauer, 1991).

3.3 Selective Execution Strategy

As mentioned in Section 2.3, code generation is inherently a multi-label problem. The LTT framework supports label augmentation on the calibration set $\mathcal{D}_{\text{cal}}$. For each generation task of $X_i \in \mathcal{D}_{\text{cal}}$, we sample additional m programs from the generative code model and execute test cases to identify correct instances. Such a sampling and verification step is performed only during calibration, incurring no computational overhead at test time.

Despite this, in scenarios with severely limited computational resources, we propose a selective execution strategy that maintains high-quality labels while improving sampling efficiency. Given a total of M sampled programs on $\mathcal{D}_{\text{cal}}$, we use program uncertainty scores $\{U_i\}_{i=1}^M$ (e.g., perplexity)

to decide whether executing test cases is necessary. We learn a threshold $\hat{u}$ such that programs with $U_i \leq \hat{u}$ are directly accepted as correct, while those with $U_i > \hat{u}$ are verified by executing test cases. This strategy does not require perfectly calibrated models, but better calibration can indeed enhance the precision of $\hat{u}$. Formally, we define the cumulative error function conditioned on u as

$$L(u) = \frac{1}{M} \sum_{i=1}^{M} \ell(S_i) \mathbf{1}\{U_i \leq u\},$$

where $S_i \in \{0, 1\}$ is a true indicator (1 for a correct program, 0 otherwise), and $\ell(S_i) = 1 - S_i$ computes the error.

Our goal is to select a threshold u such that $L(u) \leq \epsilon$, where ϵ is a label error rate. However, directly computing $L(u)$ by verifying all M sampled programs is computationally expensive. To circumvent this, we instead estimate an error upper bound $\hat{L}^u(\gamma)$ on $L(u)$ that holds pointwise with probability γ, i.e., satisfies the following inequality:

$$\mathbb{P}(\hat{L}^u(\gamma) \geq L(u)) \geq 1 - \gamma. \tag{9}$$

To compute an approximate error bound $\hat{L}^u(\gamma)$, we propose a procedure analogous to importance sampling, inspired by Candès et al. (2025). We begin by drawing h indices $\{i_1, \ldots, i_h\}$ uniformly with replacement from $[M]$. For each sampled index i_j, whether a test-case execution is performed is determined stochastically by a Bernoulli trial $\xi_{i_j} \sim \mathrm{Bern}(\omega_{i_j})$, where $\{\omega_i\}_{i=1}^{M}$ are predefined sampling weights. This procedure yields a set $\{Z_j(u)\}_{j=1}^{h}$ of h i.i.d. random variables, defined as $Z_j(u) = \ell(S_{i_j})(\xi_{i_j}/\omega_{i_j})\mathbf{1}\{U_{i_j} \leq u\}$.

Since $\mathbb{E}[\xi_{i_j}/\omega_{i_j} \mid i_j] = 1$, the expectation of $Z_j(u)$ equals the target quantity $L(u)$. Therefore, an unbiased estimate of $L(u)$ can be obtained from the sample mean of $\{Z_j(u)\}_{j=1}^{h}$. An upper confidence bound for $L(u)$ is then constructed by Hoeffding's inequality (Hoeffding, 1963), yielding a finite-sample upper bound

$$\hat{L}^u(\gamma) = \hat{\mu}_Z(u) + \delta_{HB}(\gamma), \tag{10}$$

where $\hat{\mu}$ denotes the empirical mean of $\{Z_j(u)\}_{j=1}^{h}$ and Hoeffding bound $\delta_{HB}(\gamma) = \sqrt{\log(2/\gamma)/2h}/\omega_{\min}$, where $\omega_{\min} = \min\{\omega_i\}_{i=1}^{M}$.

After computing $\hat{L}^u(\gamma)$, we determine $\hat{u} = \max\{u : \hat{L}^u(\gamma) \leq \epsilon\}$ with $\epsilon \in (0, 1)$. The final predicted label $\hat{S}_i$ is then given by

$$\hat{S}_i = S_i \mathbf{1}\{U_i > \hat{u}\} + \mathbf{1}\{U_i \leq \hat{u}\}.$$

We summarize the above selective execution procedure as Algorithm 3 (see Appendix D). Then, we have the following error upper bound.

Lemma 3.3 (Non-asymptotic error upper bound)**.** *If the label set $\{\hat{S}_i\}_{i=1}^{M}$ of sample programs is computed by Algorithm 3, then it meets Eq.* (9) *and $\hat{L}^u(\gamma) \leq \epsilon$.*

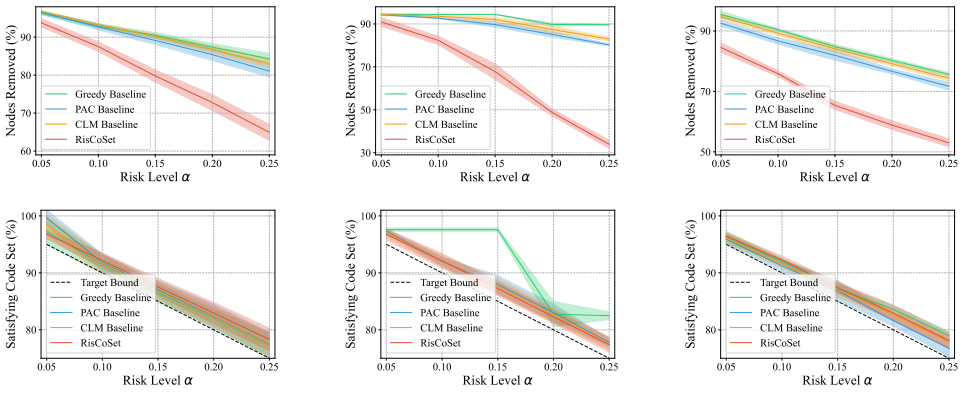
This result provides a non-asymptotic upper bound for $L(u)$, suitable for settings with limited sample sizes (small M). Alternatively, an asymptotic bound for large M is given in Appendix D.

Combining Lemma 3.3 and Theorem 3.2, we obtain the error-controlling guarantee when using the selective execution strategy.

Proposition 3.4 (Approximately risk-controlling sets)**.** *Let the output of Algorithm 3 be the calibration set $\mathcal{D}_{\mathrm{cal}}$. Then, the parameter $\hat{\lambda}$ selected by Eq.* (8) *satisfies*

$$\mathbb{P}\big(R^s(\hat{\lambda}) \leq \alpha + \epsilon(1 - \alpha)\big) \geq (1 - \gamma) \cdot (1 - \delta).$$

Remark 3.5. This guarantee presents a practical trade-off, allowing for controlled relaxation of the risk guarantee in exchange for computational savings by permitting a non-zero label error rate ϵ. When $\epsilon = 0$, the result reduces to the original guarantee of Theorem 3.2.



(a) DeepSeek-33B on HumanEval (b) Qwen-32B on MBPP (c) Llama-70B on APPS

Figure 2: Percentage of node removals (top row) and satisfying code sets (bottom row) w.r.t. risk level α. The results are the mean over 100 random splits. The smaller node removal is better, when the code set coverage exceeds the target bound $1 - \alpha$. Our approach constructs prediction sets that remove significantly fewer nodes compared to baselines for three LLMs on all datasets.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate our approach on four standard Python code generation benchmarks: HumanEval (Chen, 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and LiveCodeBench (Jain et al., 2025), as well as one text-to-SQL generation benchmark, Spider (Lei et al., 2025). HumanEval consists of 164 hand-written programming problems. We use the sanitized version of the MBPP dataset (nearly 500 problems) as provided by CodeT (Chen et al., 2022), and select the introductory-level problems of the APPS dataset (1,000 problems). Each dataset is then randomly partitioned into calibration and test sets using a 50/50 split. To mitigate variance from random partitioning, we repeat this process over 100 independent random trials.

Baselines. We compare RisCoSet against three baselines: the PAC prediction set approach (Khakhar et al., 2023), its greedy variant, and conformal language modeling (CLM) (Quach et al., 2024). The PAC baseline constructs partial programs by solving a complex optimization problem, whereas its greedy variant iteratively removes the AST node that most reduces the partial program's negative log-likelihood (NLL). CLM applies line-level conformal prediction by treating programs as natural language, and then enforces tree constraints to yield valid partial programs.

Evaluation metrics. To evaluate the structured prediction sets $\mathcal{T}_{\hat{\lambda}}(X_{n+1})$ produced by our RisCoSet, we report two metrics: (1) the percentage of nodes removed from AST, and (2) the coverage of code sets that contain at least one correct program. We aim to keep node removals small while ensuring that coverage remains above the target threshold $1 - \alpha$, thereby preserving more useful program information.

Implementations.[2] We evaluate three open-source LLMs: the code-specialized DeepSeek-Coder-33B (Guo et al., 2024) and Qwen2.5-Coder-32B (Hui et al., 2024), along with the general-purpose Llama3.1-70B (Grattafiori et al., 2025). We choose these models because they are open-source and easy to deploy, although the proposed approach is applicable to any LLMs. The For all LLMs, we set temperature = 0.8 and top-p = 0.95 by default. We choose fixed sequence testing as the FWER algorithm for LTT. Moreover, we solve the optimization problem for generating partial programs using the Z3 solver, enforcing a 24-second timeout per instance in accordance with SMT-COMP[3] rules. All the experiments are carried out with Python 3.9.19, PyTorch 2.4.0 and vLLM 0.6.1, running on NVIDIA H800 GPUs with CUDA 12.7.

[2]Our code is available online at an anonymous repository: https://github.com/Anonymity26543/RisCoSet.

[3]SMT-COMP is the annual international competition for Satisfiability Modulo Theories (SMT) solvers.

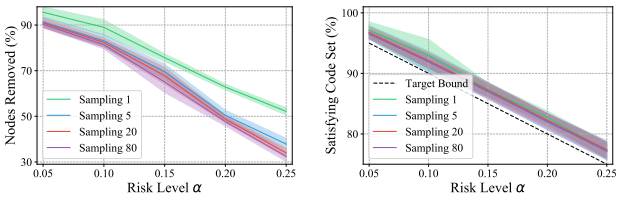


Figure 3: Parameter sensitivity analysis of sampling quantity m w.r.t. risk level α on MBPP. The average results over 100 trials show that a larger value of m leads to fewer node removals, while maintaining the required risk control.

ERROR ϵ	0.05	0.1	0.2	0.3
REMOVAL	71.8	50.6	27.8	3.39
COVERAGE	88.5 (85.5)	82.9 (81.0)	74.1 (72.0)	66.5 (63.0)
SAVE	12.0	29.4	57.6	84.4

Table 1: Percentage of node removals (%), coverage of satisfying code set (%), and fraction of calibration data saved from execution (%) w.r.t. label error ϵ when $\alpha = 0.1$ on MBPP. The values in () are the target coverage.

4.2 MAIN RESULTS

Our main results are presented in Fig. 2. For all plots, the solid line shows the average values over 100 trials, and the shaded regions indicate the standard deviation.

Validity of risk-controlling code generation. We implement our approach RISCOSET for different LLMs on three datasets. The results in Fig. 2 illustrate, with varying risk levels α, the percentage of nodes removed from the pruned AST (top row) and the coverage of constructed prediction sets that contain a correct program (bottom row).

Our empirical findings align with Theorem 3.2 in practice, as the average code set coverage consistently exceeds the target bound of $1 - \alpha$ (the dashed lines). Across all datasets, our method produces prediction sets that remove significantly fewer AST nodes than the baselines. For instance, as shown in Fig. 2(b) and Fig. 2(c), our approach reduces node removals by 24.5% and 20.3% compared to the best competitor when $\alpha = 0.15$ on the MBPP and APPS datasets, respectively. Compared to the PAC baseline, both the greedy and CLM baselines tend to remove more nodes. The greedy baseline does so due to its greedy pruning strategy, whereas the CLM baseline treats programs as natural language and thus does not fully account for structural information during pruning.

For the code-specialized models, our approach performs better with Qwen-32B than with DeepSeek-33B. This may be attributed to Qwen-32B's stronger code generation capability (Hui et al., 2024), suggesting that our method can benefit increasingly from more powerful models. Moreover, the proposed method also achieves performance comparable to that of specialized code generators when applied to the general-purpose Llama-70B in Fig. 2(c), showing its scalability for diverse LLMs.

The significant improvement of our method can be attributed to: (1) we define an optimization problem with fewer constraints, encouraging a better partial program solution with more remaining nodes; (2) our multi-label formalization allows for a more complete exploration of the potential correct program space for a specific LLM compared to baselines.

Parameter sensitivity analysis. We next analyze the sensitivity of our method to the sampling quantity m. We sample m additional programs for each task in the calibration set with $m = \{1, 5, 20, 80\}$, and pick correct ones as new labels via verification. The corresponding node removals and code set coverage on the MBPP dataset are given in Fig. 3(a) and Fig. 3(b). We observe a clear trend that a larger m helps reduce the number of pruned AST nodes while preserving the desired risk control. Insufficient exploration of the label space at $m = 1$ leads to more node removals. Nevertheless, the performance gains diminish for $m > 5$, demonstrating that our method already performs well with a relatively small sample size.

Selective execution strategy. We evaluate the effectiveness of the proposed selective execution strategy when $\alpha = 0.1$ and $m = 80$ on the MBPP dataset. In experiments, we set confidence level $\gamma = 0.01$, sampling size h as 10% of total sample quantity M, and sampling weights $\omega = \omega_i = 1.0$, for any $i \in [M]$. We report the average results with varying label error ϵ over 100 trials in Table 1. The key metric for efficiency is the percentage of sample programs saved from execution at the cost of $\epsilon\%$ error in program labels. First, the observed coverage of the satisfying code set always exceeds the target coverage, which controls risks, consistent with the theoretical guarantee in Proposition 3.4. Second, our selective execution strategy largely reduces the proportion of verifications by running test

cases. For instance, at $\epsilon = 0.1$, it avoids test-case execution for nearly 30% of generated programs, substantially lowering computational cost.

Additional experimental results. Additional experimental results are provided in Appendix E to further validate the effectiveness and generality of our approach. These include comparisons of different FWER-controlling algorithms (cf. Fig. 4(a), Table 2), a sensitivity analysis of the maximum subtree removals $t_{\max}$ (cf. Fig. 4(b) and 4(c)), a text-to-SQL evaluation on Spider (cf. Table 3), results on the more challenging LiveCodeBench benchmark (cf. Table 4), an evaluation of using partial programs to guide LLM-based code completion (which shows the practical downstream applicability of our method; cf. Table 5), an assessment of robustness to distribution shift via RISCOSET+ (cf. Table 6), and sample outputs of partial programs (cf. Figs. 5–7).

5 RELATED WORK

Conformal prediction and risk control. Traditional CP methods, designed for classification and regression, are ill-suited to generative tasks with infinite output spaces (Campos et al., 2024). Therefore, some risk-controlling frameworks are proposed for generation tasks (Angelopoulos et al., 2022; Pan & Wu, 2025). However, these methods, such as PAC prediction sets (Park et al., 2020), require an additional monotonicity constraint regarding risk, making them difficult to adapt to multi-label problems. To this end, the Learn Then Test (LTT) framework (Angelopoulos et al., 2025) separates model training from statistical validation and uses multiple hypothesis testing to control risk. Without imposing the monotonicity assumption on the output space, LTT is particularly applicable to multi-label settings like code generation.

Uncertainty estimation on LLMs. Recent work has investigated the application of CP to LLMs to achieve rigorous statistical guarantees on the correctness of generated outputs (Ashby et al., 2026). One line of work applies it to produce a set of candidate responses that, with high probability, contains a valid answer (Ye et al., 2024; Kotte, 2026). This task-level approach, however, has limited practicality because evaluating a large number of candidates is computationally expensive. Alternatively, sample-level methods apply conformal prediction internally to a single LLM output, identifying its sound components as a prediction set (Quach et al., 2024; Cherian et al., 2024). However, these approaches rely on the assumption of near-independence among components. While this may hold in NLP contexts, it breaks down for structured domains like code, where components are systemically interdependent (Chen et al., 2022).

Reliable code generation. Ensuring code correctness has long been a central concern in software engineering. Conventional methods include testing (e.g., unit testing and fuzzing) (Zhu et al., 2022), static analysis (Park et al., 2021), and formal verification (Ju et al., 2026). Nevertheless, such methods are inadequate for LLM-generated code, as they cannot effectively leverage the probabilistic nature and uncertainty of modern language models. The advancement of LLMs for code generation has spurred interest in reliability methods based on statistical, rather than purely deductive, guarantees (Spiess et al., 2025; Campos et al., 2026). For example, Guo et al. (2021) complete the uncertain code parts within the program sketches to improve the pass rate. Recently, research has explored constructing PAC prediction sets for code generation (Khakhar et al., 2023). By representing programs via ASTs and building prediction sets defined by partial programs, this method delivers probabilistic guarantees that the code set contains the ground-truth program. Unlike previous work, our approach introduces LTT framework to eliminate monotonicity constraints and naturally accommodates the multi-label essence of code.

6 CONCLUSION

This paper addresses the challenge of unreliable LLM outputs in structured generation tasks like code. We introduce a method for constructing statistically guaranteed prediction sets by reformulating the problem as multi-label classification, eliminating monotonicity constraints. Our approach applies the LTT framework to ensure a candidate set of programs contains a correct solution with high probability. Furthermore, a selective execution strategy is proposed to augment label sets for efficient and effective calibration. Extensive experiments across multiple models and benchmarks confirm the method's effectiveness. While our focus is on code generation, we believe the proposed method can be adapted to other structured prediction problems.

Reproducibility Statement

To facilitate the reproducibility of our work, we have made our source code available online through a public repository. All experimental details, including dataset partitions, hyperparameter configurations, and model details, are fully documented in Section 4.1. We are confident that these materials provide the necessary information to replicate our findings.

Usage of LLMs

Large Language Models (LLMs) were utilized exclusively as writing assistants to enhance the linguistic quality of this manuscript, focusing on improving clarity, grammar, and readability. Their involvement was strictly limited to this editorial function. LLMs played no role in any substantive research components, including conceptualization, experimental design, data analysis, interpretation of results, or scientific content creation. All intellectual contributions, methodological developments, findings, and conclusions originate solely from the authors.

References

Anastasios N Angelopoulos, Stephen Bates, Adam Fisch, Lihua Lei, and Tal Schuster. Conformal risk control. *arXiv preprint arXiv:2208.02814*, 2022.

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

Alice Evelyn Ashby, Khuong An Nguyen, and Zhiyuan Luo. Uncertainty-aware large language models: a scoping review of conformal prediction methods. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 384(2327):20260061, 2026.

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

Stephen Bates, Anastasios Angelopoulos, Lihua Lei, Jitendra Malik, and Michael Jordan. Distribution-free, risk-controlling prediction sets. *Journal of the ACM (JACM)*, 68(6):1–34, 2021.

Peter Bauer. Multiple testing in clinical trials. *Statistics in medicine*, 10(6):871–890, 1991.

Vidmantas Bentkus. On hoeffding's inequalities. 2004.

Patrick Billingsley. The lindeberg-levy theorem for martingales. *Proceedings of the American Mathematical Society*, 12(5):788–792, 1961.

Margarida Campos, António Farinhas, Chrysoula Zerva, Mário AT Figueiredo, and André FT Martins. Conformal prediction for natural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 12:1497–1516, 2024.

Viola Campos, Robin Kuschnereit, and Adrian Ulges. Multicalibration for llm-based code generation. In *Proceedings of the 1st International Workshop on AI for Software Quality Evaluation-Judgment, Metrics, Benchmarks, and Beyond*, pp. 61–71, 2026.

Emmanuel J Candès, Andrew Ilyas, and Tijana Zrnic. Probably approximately correct labels. *arXiv preprint arXiv:2506.10908*, 2025.

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

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

John J. Cherian, Isaac Gibbs, and Emmanuel J. Candès. Large language model validity via enhanced conformal prediction methods, 2024. URL https://arxiv.org/abs/2406.09714.

Chengtao Gan, Zhiqiang Liu, Long Jin, Yushan Zhu, Lei Liang, and Wen Zhang. Craftqa: A code-driven adaptive framework for complex structured data reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 27860–27876, 2026.

Cuiyun Gao, Guodong Fan, Chun Yong Chong, Shizhan Chen, Chao Liu, David Lo, Zibin Zheng, and Qing Liao. A systematic literature review of code hallucinations in llms: Characterization, mitigation methods, challenges, and future directions for reliable ai. *arXiv preprint arXiv:2511.00776*, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. 2025.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. Learning to complete code with sketches. *arXiv preprint arXiv:2106.10158*, 2021.

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

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

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pp. 65–70, 1979.

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

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2): 1–72, 2026.

Haocheng Ju, Guoxiong Gao, Jiedong Jiang, Bin Wu, Zeming Sun, Shurui Liu, Leheng Chen, Yutong Wang, Yuefeng Wang, Zichen Wang, et al. Automated conjecture resolution with formal verification. *arXiv preprint arXiv:2604.03789*, 2026.

Adam Tauman Kalai, Ofir Nachum, Santosh S Vempala, and Edwin Zhang. Why language models hallucinate. *arXiv preprint arXiv:2509.04664*, 2025.

Kevin Kasa, Zhiyu Zhang, Heng Yang, and Graham W Taylor. Adapting prediction sets to distribution shifts without labels. *arXiv preprint arXiv:2406.01416*, 2024.

Adam Khakhar, Stephen Mell, and Osbert Bastani. Pac prediction sets for large language models of code. In *International Conference on Machine Learning*, pp. 16237–16249. PMLR, 2023.

Varun Kotte. Pasc: Pipeline-aware conformal prediction with joint coverage guarantees for multi-stage nlp and llm pipelines. *arXiv preprint arXiv:2605.18812*, 2026.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, volume 2025, pp. 28691–28735, 2025.

Zhiming Li, Junzhe Jiang, Yushi Cao, Aixin Cui, Bozhi Wu, Bo Li, Yang Liu, and Danny Dongning Sun. Logic-q: Improving deep reinforcement learning-based quantitative trading via program sketch-based tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 18584–18592, 2025.

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

Varun Magesh, Faiz Surani, Matthew Dahl, Mirac Suzgun, Christopher D Manning, and Daniel E Ho. Hallucination-free? assessing the reliability of leading ai legal research tools. *Journal of empirical legal studies*, 22(2):216–242, 2025.

Shuaidong Pan and Di Wu. Trustworthy summarization via uncertainty quantification and risk awareness in large language models. In *2025 6th International Conference on Computer Vision and Data Mining (ICCVDM)*, pp. 523–527. IEEE, 2025.

Jihyeok Park, Hongki Lee, and Sukyoung Ryu. A survey of parametric static analysis. *ACM Comput. Surv.*, 54(7), July 2021. ISSN 0360-0300. doi: 10.1145/3464457. URL https://doi.org/10.1145/3464457.

Sangdon Park, Osbert Bastani, Nikolai Matni, and Insup Lee. Pac confidence sets for deep neural networks via calibrated prediction. In *International Conference on Learning Representations*, 2020.

Victor Quach, Adam Fisch, Tal Schuster, Adam Yala, Jae Ho Sohn, Tommi S. Jaakkola, and Regina Barzilay. Conformal language modeling, 2024. URL https://arxiv.org/abs/2306.10193.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Calibration and correctness of language models for code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 540–552. IEEE, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

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

Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*. Springer, 2005.

Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, Yu-Yang Liu, and Li Yuan. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469*, 2023.

Fanghua Ye, Mingming Yang, Jianhui Pang, Longyue Wang, Derek Wong, Emine Yilmaz, Shuming Shi, and Zhaopeng Tu. Benchmarking llms via uncertainty quantification. *Advances in Neural Information Processing Systems*, 37:15356–15385, 2024.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503, 2025.

Xiaofan Zhou, Baiting Chen, Yu Gui, and Lu Cheng. Conformal prediction: A data perspective. *ACM computing surveys*, 58(2):1–37, 2025.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022. ISSN 0360-0300. doi: 10.1145/3512345. URL https://doi.org/10.1145/3512345.

Yuqi Zhu, Ge Li, Xue Jiang, Jia Li, Hong Mei, Zhi Jin, Yihong Dong, and Qibin Zheng. Uncertcot: Uncertainty-aware chain-of-thought for code generation with large language model. In *International Conference on Intelligent Computing*, pp. 426–437. Springer, 2025.

A TECHNICAL PROOFS

A.1 PROOF OF LEMMA 3.1

Proof. Let $A = \mathrm{Bin}(n, \alpha)$ and $B = n\widehat{R}_n^s(\lambda_k)$. Since the risk in Eq. (4) is an average of binary outcomes, B is also binomial with sample size n and some (unknown) success probability α'. Let F_A and F_B denote the CDFs of A and B, respectively. Under the null hypothesis $\mathcal{H}_k : R^s(\lambda_k) > \alpha$ on test instances, we have $\alpha' > \alpha$. This implies that B stochastically dominates A, i.e., $F_A(t) \geq F_B(t)$ for all t. Let $C = p_k^{BT} = eF_A(B)$, then

$$\begin{aligned}\mathbb{P}(C \leq c) &= \mathbb{P}(F_A(B) \leq \frac{c}{e}) \\ &\leq \mathbb{P}(F_B(B) \leq \frac{c}{e}).\end{aligned}$$

By the definition of a CDF, we have

$$\begin{aligned}\mathbb{P}(F_B(B) \leq \frac{c}{e}) &= \mathbb{P}(B \leq F_B^{-1}(\frac{c}{e})) \\ &= F_B\Big(F_B^{-1}\Big(\frac{c}{e}\Big)\Big) \\ &= \frac{c}{e} \leq c.\end{aligned}$$

Therefore, under $\mathcal{H}_k$, p_k^{BT} is super-uniform and hence a valid p-value.

Furthermore, by Bentkus inequality (Bentkus, 2004), we obtain

$$\mathbb{P}(\widehat{R}_n^s(\lambda_k) \leq t) \leq e\mathbb{P}\big(\mathrm{Bin}(n, \alpha) \leq \lceil nt \rceil\big).$$

This inequality is tightest when $nt = \lceil nt \rceil$, because $n\widehat{R}_n^s(\lambda_k)$ is binomial as discussed above. Therefore, p_k^{BT} is a near-tight upper bound for the CDF of $\widehat{R}_n^s(\lambda_k)$. □

A.2 PROOF OF THEOREM 3.2

Proof. Since the enhanced calibration set is i.i.d., the set losses $\mathbf{1}\{\mathcal{T}_\lambda(X_i) \cap \tilde{Y}_i = \emptyset\}$ in Eq. (4) are also i.i.d. By Lemma 3.1, each p_k^{BT} is a valid p-value under the null hypothesis $\mathcal{H}_k : R^s(\lambda_k) > \alpha$. Given a set-valued function $\mathcal{T}_\lambda$, and a FWER-controlling algorithm at level δ, we can apply Theorem 2.2 to identify Λ_{valid} such that

$$\mathbb{P}\Big(\sup_{\lambda \in \Lambda_{\text{valid}}} \{R^s(\lambda)\} \leq \alpha \mid \mathcal{D}_{\text{cal}}\Big) \geq 1 - \delta$$

on test programs. Therefore, we have

$$\mathbb{P}(R^s(\hat{\lambda}) \leq \alpha \mid \mathcal{D}_{\text{cal}}) \geq 1 - \delta,$$

as Eq. (3) holds for any $\lambda \in \Lambda_{\text{valid}}$. In particular, it holds for $\hat{\lambda}$ selected by Eq. (8).

□

A.3 PROOF OF LEMMA 3.3

Proof. We first analyze the properties of the random variables $Z_j(u)$ constructed in Algorithm 3. The random variables $Z_1(u), \ldots, Z_h(u)$ are independent, because index i_j is uniformly sampled from $[M]$, and ξ_{i_j} is independent of other random variables. Since binary loss $\ell(S_{i_j})$ is bounded by $[0, 1]$, if $\xi_{i_j} = 1$, we have

$$Z_j = \frac{\ell(S_{i_j})}{\omega_{i_j}} \leq \frac{1}{\min_j \omega_{i_j}} = \frac{1}{\omega_{\min}}.$$

If $\xi_{i_j} = 0$, $Z_j = 0$. Let $R = 1/\omega_{\min}$. Then $Z_j \in [0, R]$ for all $j \in [h]$.

We denote $\bar{Z}(u) = \frac{1}{h}\sum_{j=1}^{h} Z_j(u)$, and compute the expected $\bar{Z}(u)$ as

$$\begin{aligned}\mathbb{E}[\bar{Z}(u)] &= \frac{1}{h}\sum_{j=1}^{h}\mathbb{E}[Z_j(u)] \\ &= \frac{1}{h}\sum_{j=1}^{h}\Big[\omega_{i_j}\cdot\frac{\ell(S_{i_j})}{\omega_{i_j}}\mathbf{1}\{U_{i_j}\le u\} + (1-\omega_{i_j})\cdot 0\Big] \\ &= \frac{1}{h}\sum_{j=1}^{h}\ell(S_{i_j})\mathbf{1}\{U_{i_j}\le u\} = L(u).\end{aligned}$$

By Hoeffding's inequality and $L(u) = \mathbb{E}[\bar{Z}(u)]$, we obtain

$$\mathbb{P}\big(|\hat{\mu}_Z(u) - L(u)| \ge t\big) \le 2\exp\Big(-\frac{2ht^2}{R^2}\Big), \tag{11}$$

where $t > 0$ is arbitrary. Setting the right-hand side equal to γ yields

$$2\exp\Big(-\frac{2ht^2}{R^2}\Big) = \gamma \quad \Rightarrow \quad t = R\sqrt{\frac{\log(2/\gamma)}{2h}}.$$

Substituting this value of t into Eq. 11 yields

$$\mathbb{P}\Big(\hat{\mu}_Z(u) - L(u) \le -R\sqrt{\frac{\log(2/\gamma)}{2h}}\Big) \le \frac{\gamma}{2} \le \gamma.$$

Equivalently,

$$\mathbb{P}\Big(L(u) \le \hat{\mu}_Z(u) + R\sqrt{\frac{\log(2/\gamma)}{2h}}\Big) \ge 1-\gamma.$$

Let $\hat{L}^u(\gamma) = \hat{\mu}_Z(u) + R\sqrt{\log(2/\gamma)/2h}$. Then

$$\mathbb{P}(L(u) \le \hat{L}^u(\gamma)) \ge 1-\gamma,$$

which completes the proof. □

A.4 Proof of Proposition 3.4

Proof. We prove Proposition 3.4. In Section 3.3, we select $\hat{u} = \max\{u \in [0,1] : \hat{L}^u(\gamma) \le \epsilon\}$, and by Lemma 3.3 we have the following upper bound:

$$L(u) \le \hat{L}^u(\gamma) = \epsilon. \tag{12}$$

According to Theorem 3.2, for a test instance (X_{n+1}, Y_{n+1}), we have

$$\mathbb{P}\big(\mathbb{P}(\mathcal{T}_{\hat{\lambda}}(X_{n+1}) \cap Y_{n+1} = \emptyset) \le \alpha)\big) \ge 1-\delta.$$

In this guarantee, let $Y_{n+1} = Y^t_{n+1} \cup Y^f_{n+1}$, where Y^t_{n+1} and Y^f_{n+1} are disjoint sets and $|Y^t_{n+1}|/|Y_{n+1}| \ge 1-\epsilon$ according to Eq. (12). Let $C = \mathcal{T}_{\hat{\lambda}}(X_{n+1}) \cap Y_{n+1}$. Under the assumption that false positives are not systematically more likely to be included in the prediction set than true positives, i.e.,

$$\mathbb{P}(C \subseteq Y^f_{n+1} \mid C \ne \emptyset) \le \epsilon,$$

we have

$$\begin{aligned}\mathbb{P}(\mathcal{T}_{\hat{\lambda}}(X_{n+1}) \cap Y^t_{n+1} = \emptyset) &= \mathbb{P}(C = \emptyset) + \mathbb{P}(C \ne \emptyset)\mathbb{P}(C \subseteq Y^f_{n+1} \mid C \ne \emptyset) \\ &\le \alpha + \epsilon(1-\alpha).\end{aligned}$$

This holds with probability at least $(1-\gamma)\cdot(1-\delta)$ (since the bounds involving ϵ and α hold with probabilities at least $1-\gamma$ and $1-\delta$, respectively).

□

A.5 Proof of Proposition D.1

Proof. Let h be the number of sample programs from LLMs. By the Lindeberg–Feller central limit theorem (Billingsley, 1961), we obtain

$$\frac{\sqrt{h}(L(u)-\hat{\mu}_Z(u))}{\sigma_Z(u)} \to \mathcal{N}(0,1)$$

as $h\to\infty$, since the $Z_j(u)$ are i.i.d. By the weak law of large numbers, we have $\hat{\sigma}_Z(u) \xrightarrow{p} \sigma_Z(u)$, and thus, an application of Slutsky's theorem gives

$$\mathbb{P}\Big(\frac{L(u)-\hat{\mu}_Z(u)}{\sqrt{\hat{\sigma}_Z^2(u)/h}} \le z_{1-\gamma}\Big) \to 1-\gamma,$$

because $z_{1-\gamma}$ is the $(1-\gamma)$-quantile of the standard normal distribution. Equivalently, when h approaches positive infinity, we have

$$\mathbb{P}\Big(L(u) \le \hat{\mu}_Z(u) + z_{1-\gamma}\sqrt{\hat{\sigma}_Z^2(u)/h}\Big) \to 1-\gamma,$$

which completes the proof. □

B Risk-Controlling Algorithm for Code Generation

Algorithm 1 describes how to get partial programs via picking desirable $\hat{\lambda}$. It ensures that the prediction set $\mathcal{T}_{\hat{\lambda}}(X_{n+1})$ constructed by these partial programs is risk-controlling (cf. Theorem 3.2).

Algorithm 1 Risk-Controlling Partial Program

Input: enhanced calibration set $\mathcal{D}_{\text{cal}} = \{(X_i, \tilde{Y}_i)\}_{i=1}^n$; test program X_{n+1}; parameters $\Lambda = \{\lambda_k\}_{k=1}^N$; maximum of subtree removals $t_{\max}$; risk tolerance α; error level δ;
Output: partial program $\beta_{n+1,v}$ for X_{n+1}.
for $k = 1,\dots,N$ **do**
  for $i = 1,\dots,n$ **do**
    $\beta_{i,v}^{(k)} \leftarrow$ Solve Eqs.(5)-(6) for X_i, given λ_k;
  end for
  Define $\widehat{R}_n^s(\lambda_k)$ using Eq. (4);
  Compute p-value p_k^{BT} for $\widehat{R}_n^s(\lambda_k)$ using Eq. (7);
end for
Obtain $\Lambda_{\text{valid}} = \mathcal{A}(p_1^{BT},\dots,p_N^{BT})$ via controlling FWER;
Select $\hat{\lambda} = \max(\Lambda_{\text{valid}})$;
$\beta_{n+1,v} \leftarrow$ Solve Eqs.(5)-(6) for X_{n+1}, given $\hat{\lambda}$.

C FWER-controlling algorithm

After obtaining valid p-values by Eq. (7), we discuss procedures for computing $\Lambda_{\text{valid}} = \mathcal{A}(p_1^{BT},\dots,p_N^{BT})$ on structured risks. Since evaluating multiple hypotheses simultaneously increases the chance of false positives, we require family-wise error rate (FWER) correction.

Definition C.1. (FWER control) Let $\mathcal{A}(p_1^{BT},\dots,p_N^{BT})$ be an algorithm that outputs a subset $\Lambda_{\text{valid}} \subseteq \Lambda$. We say $\mathcal{A}$ controls the FWER at (α,δ)-level if

$$\mathbb{P}\Big(\sup_{\lambda\in\Lambda_{\text{valid}}} R^s(\lambda) \le \alpha\Big) \ge 1-\delta,$$

where the supremum over $\emptyset$ is defined as $-\infty$.

We provide three alternative FWER-controlling algorithms: the *Bonferroni method*, the *Holm–Bonferroni method* (Holm, 1979), and *fixed sequence testing* (Bauer, 1991). The Bonferroni method defines $\Lambda_{\text{valid}} := \{\lambda_k \mid p_k^{BT} \leq \delta/N\}_{k=1}^{N}$. The Holm–Bonferroni method is an improved version of the Bonferroni method. It sorts p-values as $p_{(1)} \leq \cdots \leq p_{(N)}$ and $\Lambda_{\text{valid}} := \{\lambda_{(1)}, \ldots, \lambda_{(k)}\}$, where $k \in [N]$ is the minimum index satisfying $p_{(k)} \leq \delta/(N-k+1)$ and $p_{(k+1)} > \delta/(N-k)$.

For fixed sequence testing, we leverage the prior knowledge that partial programs with more nodes removed correspond to larger prediction sets. This implies that smaller values of λ are more likely to satisfy the target risk. In the extreme case where $\lambda = 0$, the partial program prunes the entire tree, trivially achieving a risk of zero in Eq. (4). Therefore, a heuristic correction strategy is to test λ in increasing order and stop at the first accepted hypothesis. To broaden the search, this process can be repeated from different predefined starting points (e.g., a coarse uniform grid in Λ). Let the parameter set $\Lambda = \{\lambda_1, \ldots, \lambda_N\}$ be in descending order (i.e., $\lambda_1 \geq \cdots \geq \lambda_N$). The whole fixed sequence testing procedure for code is summarized in Algorithm 2.

Algorithm 2 Fixed sequence testing

Input: parameter set $\Lambda = \{\lambda_1, \ldots, \lambda_N\}$; error level $\delta \in (0,1)$; p-values $\{p_k^{BT}\}_{k=1}^{N}$; predefined starting points $\mathcal{I} \subset [N]$;
Output: reject set Λ_{valid}.
Initialize a reject set $\Lambda_{\text{valid}} = \emptyset$;
for $k \in \mathcal{I}$ **do**
  if $\lambda_k \notin \Lambda_{\text{valid}}$ **then** ▷ *Avoid duplicates*
    while $p_k^{BT} \leq \delta/|\mathcal{I}|$ **do**
      $\Lambda_{\text{valid}} = \Lambda_{\text{valid}} \cup \{\lambda_k\}$;
      $k = k - 1$;
    end while
  end if
end for

D Non-Asymptotic and Asymptotic Upper Bounds

In this part, we first summarize the overall procedure of Section 3.3 in Algorithm 3, which satisfies the non-asymptotic error upper bound in Lemma 3.3. Additionally, we consider another asymptotic bound for settings where the number of sampled programs M is large.

In Section 3.3, we estimate an unbiased upper bound for $L(u)$ by constructing a confidence interval for the average of $\{Z_j(u)\}_{j=1}^{h}$. Based on the central limit theorem (CLT), we obtain the following asymptotic upper bound.

Proposition D.1 (Asymptotic upper bound)**.** *Given a threshold u and the sample size h, let $Z_j(u)$ be the i.i.d. random variables defined in Algorithm 4 for $j \in [h]$. Define the error upper bound based on CLT as*

$$\hat{L}^u(\gamma) := \hat{\mu}_Z(u) + z_{1-\gamma} \cdot \frac{\hat{\sigma}_Z(u)}{\sqrt{h}},$$

where $\hat{\mu}$ and $\hat{\sigma}$ denote the empirical mean and standard deviation of $\{Z_j(u)\}_{j=1}^{h}$, respectively, and $z_{1-\gamma}$ is the $(1-\gamma)$-quantile of the standard normal distribution. Then, we have

$$\liminf_{m \to \infty} \mathbb{P}(L(u) \leq \hat{L}^u(\gamma)) \geq 1 - \gamma.$$

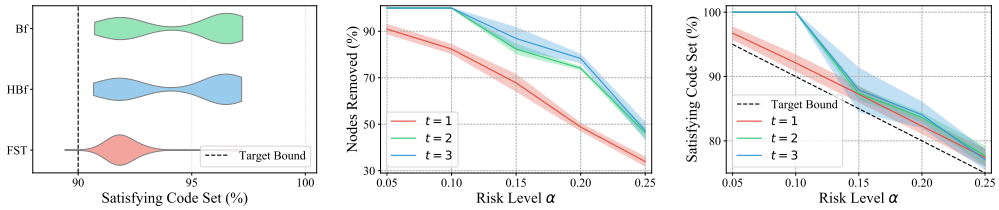
The proof is provided in Appendix A.5, and the procedure is described in Algorithm 4.

Algorithm 3 Selective Execution with Non-Asymptotic Error Bound

Input: unlabeled sampling programs $\{X_i\}_{i=1}^M$; uncertainty scores $\{U_i\}_{i=1}^M$; test cases $\{C_i\}_{i=1}^M$; sampling weights $\{\omega_i\}_{i=1}^M$; sampling size h; label error rate ϵ; confidence level γ;
Output: labeled sampling programs $\{X_i, \hat{S}_i\}_{i=1}^M$.
for $j = 1, \ldots, h$ **do**
  Sample an index $i_j \sim \mathrm{Unif}([M])$;
  Sample a random variable $\xi_{i_j} \sim \mathrm{Bern}(\omega_{i_j})$;
  if $\xi_{i_j} = 1$ **then**
    Execute the test case C_{i_j} to get S_{i_j};
    Compute the importance-weighted error using $Z_j = \ell(S_{i_j})/\omega_{i_j}$;
  else
    $Z_j = 0$;
  end if
end for
Given a threshold u, define $Z_j(u) = Z_j \cdot \mathbf{1}\{U_{i_j} \leq u\}$ for $j \in [h]$;
Compute error upper bound $\hat{L}^u(\gamma)$ on $\{Z_j(u)\}_{i=1}^h$ for all $u \in \{U_i\}_{i=1}^M$ using Eq. (10);
Determine $\hat{u} = \max\{u \in [0,1] : \hat{L}^u(\gamma) \leq \epsilon\}$;
Obtain $\hat{S}_i = S_i \mathbf{1}\{U_i > \hat{u}\} + \mathbf{1}\{U_i \leq \hat{u}\}$ for $i \in [M]$.

Algorithm 4 Compute Asymptotic Error Bound Based on Central Limit Theorem

Input: unlabeled sampling programs $\{X_i\}_{i=1}^M$; uncertainty scores $\{U_i\}_{i=1}^M$; test cases $\{C_i\}_{i=1}^M$; sampling weights $\{\omega_i\}_{i=1}^M$; sampling size h; confidence level γ;
Output: error upper bound $\hat{L}^u(\gamma)$.
for $j = 1, \ldots, h$ **do**
  Sample an index $i_j \sim \mathrm{Unif}([M])$;
  Sample a random variable $\xi_{i_j} \sim \mathrm{Bern}(\omega_{i_j})$;
  if $\xi_{i_j} = 1$ **then**
    Run the test case C_{i_j} to get S_{i_j};
    Compute the importance-weighted error using $Z_j = \ell(S_{i_j})/\omega_{i_j}$;
  else
    $Z_j = 0$;
  end if
end for
Given a threshold u, define $Z_j(u) = Z_j \cdot \mathbf{1}\{U_{i_j} \leq u\}$ for $j \in [h]$;
$\hat{\mu}_Z(u) \leftarrow \frac{1}{h}\sum_{j=1}^h Z_j(u)$;
$\hat{\sigma}_Z(u) \leftarrow \sqrt{\frac{1}{h}\sum_{j=1}^h (Z_j(u) - \hat{\mu}_Z(u))^2}$;
$z_{1-\gamma} \leftarrow (1-\gamma)$-quantile of the standard normal distribution;
$\hat{L}^u(\gamma) \leftarrow \hat{\mu}_Z(u) + z_{1-\gamma} \cdot \frac{\hat{\sigma}_Z(u)}{\sqrt{h}}$.



(a) Coverage distribution (b) Sensitivity analysis on $t_{\max}$ (c) Sensitivity analysis on $t_{\max}$

Figure 4: Fig. 4(a) is the distribution of code set coverage over 100 trials for three different FWER algorithms, respectively; Fig. 4(b) and 4(c) are parameter sensitivity results of maximum subtree removals $t_{\max}$ w.r.t risk level α. These experiments are conducted on MBPP.

Table 2: Performance of our method with different FWER algorithms when $\alpha = 0.1$. The results are the average over 100 random splits. The smaller node removal is better, when the code set coverage exceeds the 0.9.

FWER ALG.	BF	HBF	FST
REMOVAL	$87.9_{\pm 3.1}$	$87.0_{\pm 3.3}$	$82.3_{\pm 1.9}$
COVERAGE	$94.7_{\pm 2.2}$	$94.4_{\pm 2.3}$	$92.1_{\pm 1.3}$

E FURTHER EXPERIMENT DETAILS

E.1 DIFFERENT FWER-CONTROLLING ALGORITHM

To evaluate the performance of different FWER algorithms on our method, we plot the distribution of code set coverage ($\alpha = 0.1$) as a violin plot over 100 random splits of MBPP in Fig. 4(a), and list the mean results in Table. 2. The results demonstrate that compared to fixed sequence testing (FST), Bonferroni (Bf) and Holm–Bonferroni (HBf) methods are more conservative and remove more nodes, as confirmed by the results in Table 2. This indicates the importance of program prior knowledge used in FST.

E.2 PARAMETER SENSITIVITY ANALYSIS ON $t_{\max}$

We next perform parameter sensitivity analysis on the maximum subtree removals $t_{\max}$. The results on MBPP are shown in Fig. 4(b) and Fig. 4(c). We observe that the best result is achieved when $t_{\max} = 1$, and $t_{\max} \geq 2$ leads to more node removals than $t_{\max} = 1$. A plausible explanation is that when models are not well-calibrated, a larger $t_{\max}$ distributes pruned nodes and may lead to localized errors that cannot be fully corrected.

E.3 EXTENSIONS TO SQL

In addition to Python, we evaluate RISCOSET on text-to-SQL generation using the Spider dataset (Lei et al., 2025). Specifically, we use Spider 2.0-Snow (547 examples) and set a 5-second timeout for the Z3 solver. Following the same settings as Fig. 2(b), the results in Table 3 show that RISCOSET consistently achieves lower node removal than the PAC baseline while maintaining the target code set coverage ($1 - \alpha$) on SQL. For stochastic or externally dependent code, the guarantee holds as long as a deterministic evaluation environment can be established.

E.4 MORE REALISTIC DATASETS

Regarding a more challenging benchmark, we conduct experiments on LiveCodeBench (Jain et al., 2025), in its release_v1 version, which consists of 400 problems. Under the same settings as Fig. 2(a), the results in Table 4 show that RISCOSET consistently outperforms the PAC baseline, achieving lower node removal on this more challenging dataset.

Table 3: Percentage of node removals and satisfying code sets w.r.t. risk level α on Spider. The results are the average over 100 random splits. The smaller node removal is better, when the code set coverage exceeds the $1 - \alpha$.

METRIC	METHOD	$\alpha = 0.1$	$\alpha = 0.2$
REMOVAL	PAC BASELINE	$84.5_{\pm 1.6}$	$52.8_{\pm 1.9}$
	RISCOSET	$76.4_{\pm 1.3}$	$40.6_{\pm 2.2}$
COVERAGE	PAC BASELINE	$91.3_{\pm 0.7}$	$83.1_{\pm 1.4}$
	RISCOSET	$91.7_{\pm 1.1}$	$82.3_{\pm 0.9}$

Table 4: Percentage of node removals and satisfying code sets w.r.t. risk level α on LiveCodeBench. The results are the average over 100 random splits. The smaller node removal is better, when the code set coverage exceeds the $1 - \alpha$.

METRIC	METHOD	$\alpha = 0.1$	$\alpha = 0.2$
REMOVAL	PAC BASELINE	$93.6_{\pm 1.9}$	$85.7_{\pm 2.4}$
	RISCOSET	$87.6_{\pm 2.0}$	$72.5_{\pm 2.4}$
COVERAGE	PAC BASELINE	$91.8_{\pm 2.5}$	$83.3_{\pm 2.1}$
	RISCOSET	$92.5_{\pm 1.5}$	$83.0_{\pm 2.0}$

E.5 CODE COMPLETION FROM PARTIAL PROGRAMS

To demonstrate utility of our method, we can further leverage LLMs to complete partial programs, yielding executable code. Since modern LLMs offer no reliable signal of when their outputs should be trusted (Kalai et al., 2025), the partial programs produced by RISCOSET serve as an uncertainty-aware guide, identifying which AST nodes are uncertain and need completion. Specifically, we use the partial programs from Fig. 2(a) and 2(b) as prompts and ask the corresponding LLMs to fill in the missing parts. For the uncertainty-aware code generation baseline UnCert-CoT (Zhu et al., 2025), we adopt its entropy-based version and strictly adhere to its settings. We report the pass@1 results in Table 5, and RISCOSET outperforms both the competitor and direct generation, demonstrating the practical value of our method.

E.6 ROBUSTNESS TO DISTRIBUTION SHIFT

Our method RISCOSET, like most risk-controlling methods (Bates et al., 2021), assumes that the calibration and test sets are drawn from the same distribution. To handle distribution shifts, we introduce a simple yet effective extension, RISCOSET+, which reweights the calibration risk using uncertainty distribution ratios, drawing inspiration from Kasa et al. (2024). Specifically, for each generated program in the calibration set, we compute the ratio of its uncertainty distribution (e.g., perplexity) between the test set and the calibration set, and use these ratios as weights when computing the empirical risk. This requires only the uncertainty information of unlabeled test data.

We evaluate RISCOSET+ on Qwen-32B under the same settings as Fig. 2(b), using HumanEval as the calibration set and MBPP as the test set. As shown in Table 6, RISCOSET+ brings the empirical coverage much closer to the target level ($1 - \alpha$), demonstrating robustness to distribution shifts.

E.7 SAMPLE OUTPUTS

We present three real-world examples of partial programs constructed by RISCOSET in Figs. 5–7 to demonstrate its effectiveness.

Table 5: Pass@1 comparison of code generation. RISCOSET outperforms both direct generation and UnCert-CoT on HumanEval and MBPP.

METHOD	HUMANEVAL	MBPP
DIRECT	0.705	0.791
UNCERT-COT	0.752	0.837
RISCOSET	0.783	0.852

Table 6: Percentage of node removals and satisfying code sets w.r.t. risk level α under distribution shift between calibration and test sets. The smaller node removal is better, when the code set coverage exceeds the $1 - \alpha$.

METRIC	METHOD	$\alpha = 0.1$	$\alpha = 0.2$
REMOVAL	RISCOSET	57.5	6.40
	RISCOSET+	75.8	42.9
COVERAGE	RISCOSET	84.5	68.2
	RISCOSET+	90.1	79.7

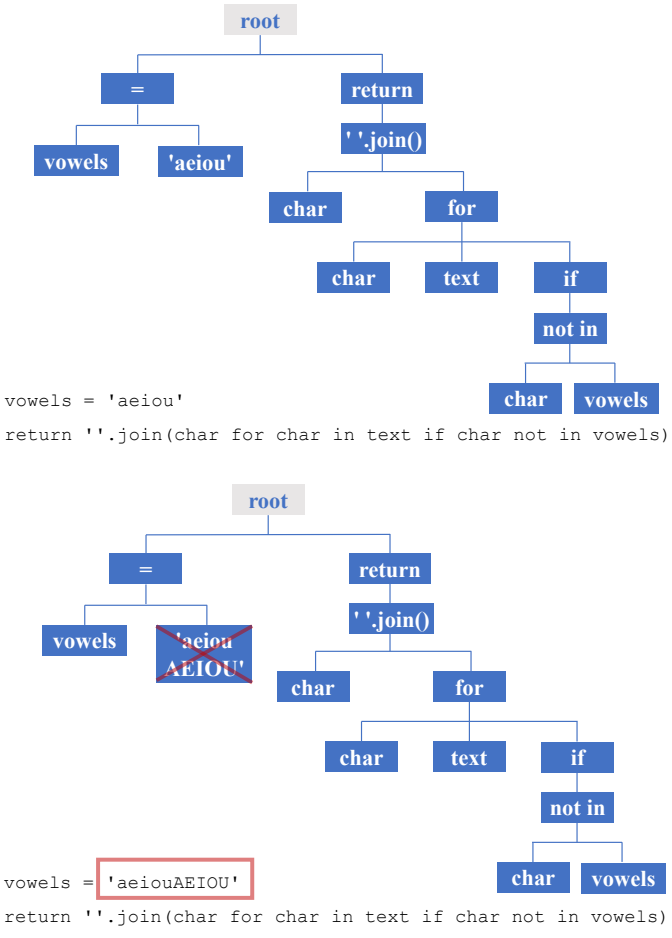


Figure 5: An illustrative example from HumanEval. The top part is a correct code snippet, and the bottom part is a generated but incorrect one. RISCOSET removes one node in the AST, resulting in a prediction set (i.e., a partial program) that contains the correct program.

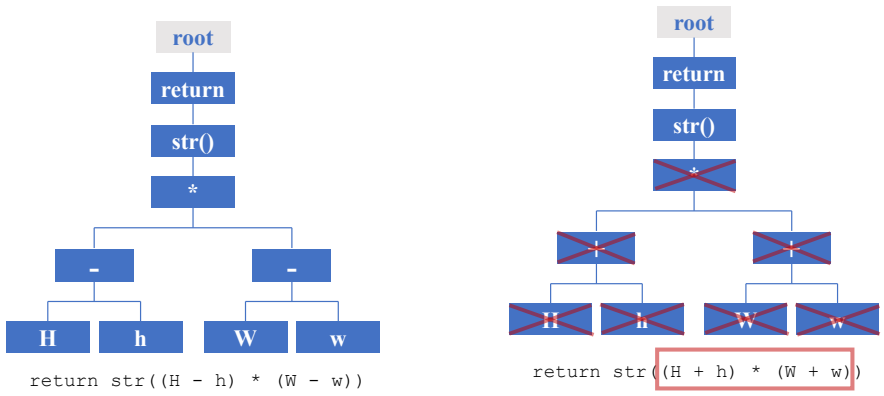


Figure 6: An illustrative example from APPS. The left part is a correct code snippet, and the right part is a generated but incorrect one. RISCOSET removes seven nodes in the AST, resulting in a prediction set (i.e., a partial program) that contains the correct program.

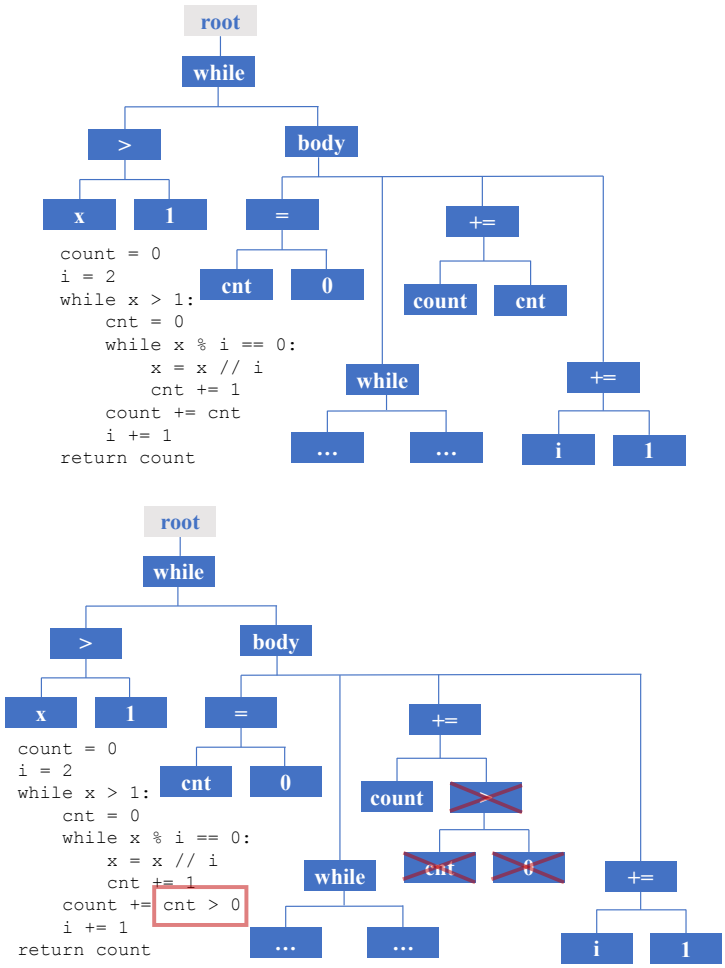


Figure 7: An illustrative example from APPS. The top part is a correct code snippet, and the bottom part is a generated but incorrect one. RISCOSET removes three nodes in the AST (we omit the overlapping AST structures for brevity), resulting in a prediction set (i.e., a partial program) that contains the correct program.