Eliminating Hallucination-Induced Errors in Code Generation with Functional Clustering

by

Chaitanya Ravuri

B.S. Computer Science and Engineering, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Chaitanya Ravuri
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Saman Amarasinghe
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Eliminating Hallucination-Induced Errors in Code Generation with Functional Clustering

by

Chaitanya Ravuri

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Modern code–generation LLMs can already solve a large fraction of programming problems, yet they still hallucinate subtle bugs that make their outputs unsafe for autonomous deployment. We present *functional clustering*, a black-box wrapper that eliminates nearly all hallucination-induced errors while providing a tunable confidence score. The wrapper samples many candidate programs, executes each on a self-generated test suite, and clusters candidates whose I/O behavior is identical; the empirical mass of the largest cluster serves as an exact confidence estimate. A single scalar threshold on this estimate lets users trade coverage for reliability with exponential guarantees. On LiveCodeBench our verifier preserves baseline pass@1 on solvable tasks yet slashes the error rate of returned answers from ∼65% to 2%, and drives it to 0% at a conservative threshold while still answering 15.6% of prompts. Manual audits show that the few residual mistakes stem from prompt misinterpretation, not random generation noise, narrowing future work to specification clarity. Because the method requires only sampling and sandbox execution, it applies unchanged to closed-source APIs and future models, offering a practical path toward dependable, autonomous code generation.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Computer Science

Acknowledgments

I gratefully acknowledge Professor Saman Amarasinghe for his guidance and encouragement throughout this thesis. My parents, Sridevi and Muralidhar Ravuri, and my brother, Agastya Ravuri, supported me every step of the way. I also thank my colleagues in the COMMIT research group and the EECS Department at MIT for their invaluable assistance and resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large language models (LLMs) have begun to write code at a level once thought exclusive to experienced engineers: they pass university exams [1], solve competitive-programming problems [2], and even draft patches that compile in production repositories [3]. Despite this progress, practitioners hesitate to deploy them unsupervised [4]. A single off-by-one error or mistyped logical operator can crash an application or leak sensitive data, and state-of-the-art models still hallucinate such bugs with disconcerting frequency [5].

Most recent work therefore augments LLMs with confidence scores [6]. Token-level likelihoods [7], calibrated logits [8], and semantic-embedding clustering [9] each offer partial signals, yet all miss a fundamental aspect of code. Two snippets that differ syntactically–or even occupy completely separate regions of embedding space–may be functionally identical. At the same time, semantically similar code with just single-character edits (such changing a $<$ to $\leq$) flips correctness without appreciably moving the embedding. Treating sequences or embeddings as the unit of analysis necessarily conflates these cases.

Our key observation is that software engineering already supplies an exact, model-agnostic criterion for identity: behavior on test inputs. In practice, virtually every production program is accepted not because it is formally proved but because it passes a finite test suite. We leverage the same idea at generation time. For each coding prompt the LLM produces a bag of candidate programs; the same model is then prompted to generate a diverse set of inputs (only inputs—no labeled outputs). Executing every program on every input in a sandbox yields an output vector; candidates whose vectors match exactly are placed in the same functional equivalence class. The empirical probability that a random sample belongs to the largest class, denoted ρ, becomes an *exact* confidence estimate (Figure 1.1).

This test-based clustering confers three immediate advantages. First, it aligns the score with the real goal–correctness of behavior–rather than proxies such as token probability or embedding distance. If the model assigns 30% probability to two syntactically different but equivalent implementations, ρ correctly reports 60% confidence, whereas logit-based metrics would cap the score at 30%. Second, the approach is entirely black-box: it requires only sampling and sandbox execution, not internal logits, gradient access, or fine-tuning. Consequently it wraps closed-source APIs as easily as open-weight models. Third, the statistic separates failure modes. Random-chance hallucinations fall exponentially fast with sample size; the rare high-confidence errors we observe stem exclusively from prompt misinterpretations. By isolating misunderstandings, the method turns reliability into a research problem that is

both narrower and more tractable.

1.1 Contributions

1. We frame uncertainty in code generation as density estimation over functional equivalence classes–programs that produce identical outputs on an automatically-generated test suite. This shifts the problem from approximate semantic similarity to exact behavioral identity, giving a mathematically crisp confidence signal without learned similarity models or log-probabilities.

2. The method requires only (i) sampling multiple programs and (ii) running them in a sandbox. It attaches to any code LLM, including closed APIs, without fine-tuning, auxiliary classifiers, or access to internal logits. It does not require any additional knowledge beyond the prompt, as it generates its own test data.

3. By prompting the base model for inputs only (not labeled test cases), we create a task-agnostic verifier that introduces no additional source of hallucination and zero human overhead.

4. A universal threshold on dominant-cluster mass converts the LLM into a selective coder that either returns a representative program or abstains, giving a very high confidence of producing a correct solution when responding.

In short, functional clustering converts noisy LLM outputs into a confidence–based response, bringing dependable, autonomous code generation within reach. Empirically, wrapping a modern code LLM with our verifier preserves baseline pass@1 on solvable benchmark tasks while slashing the error rate of returned answers from roughly 65% to 2%. Raising the acceptance threshold drives residual error to zero at the cost of additional abstentions, giving users an explicit accuracy–coverage knob. Manual inspection confirms that remaining failures are all specification misunderstandings—no random hallucinations survive.

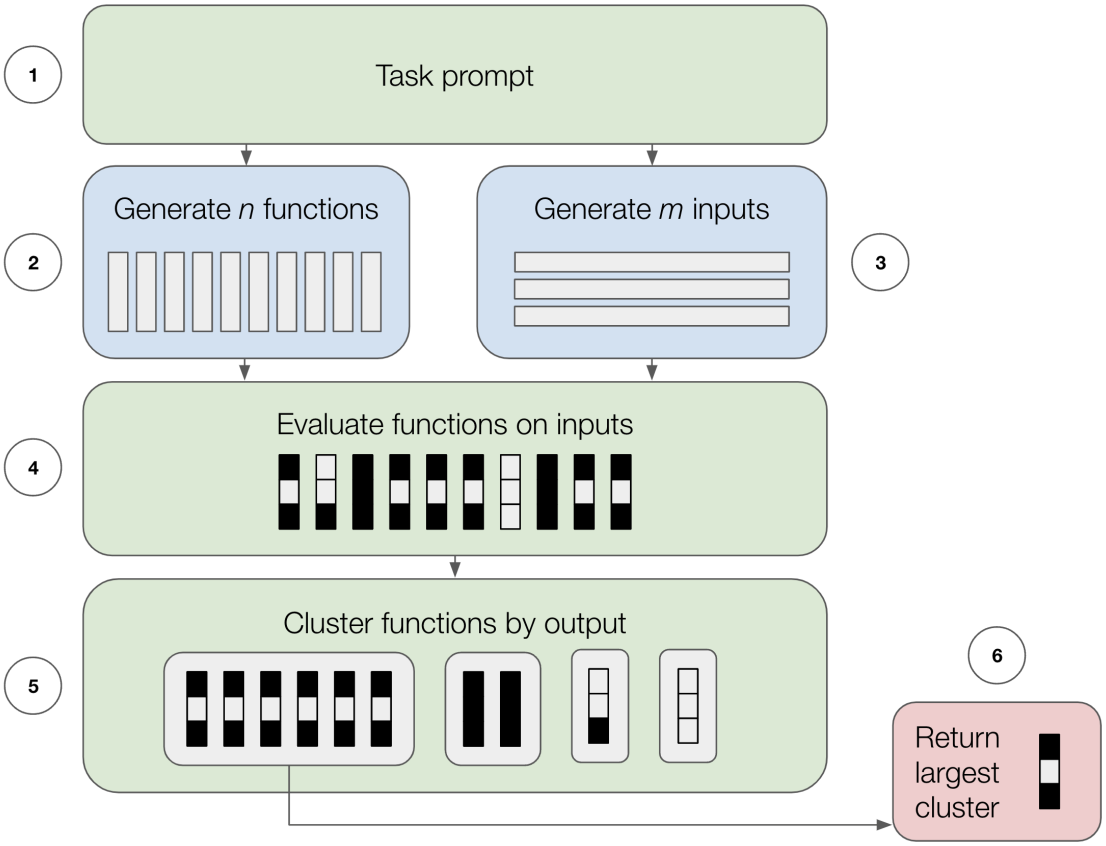

Figure 1.1: **Functional-cluster pipeline.** Starting from a natural-language task prompt (1), the system (2) samples n candidate programs and (3) synthesizes m test inputs. Each program is then (4) executed on every input, yielding a vector of outputs that acts as a behavioral signature. Programs with identical signatures are (5) clustered together; the verifier (6) selects the largest cluster as the hypothesized correct solution and reports its relative size as the confidence score.

Chapter 2

Related Works

Hallucinations in LLMs have attracted considerable attention, especially in precision-critical domains like code generation. Current research seeks to understand and classify these phenomena, mitigate their occurrence, and develop robust evaluation benchmarks. Below, we survey key directions in this area, highlighting gaps that our approach aims to address.

2.1 Understanding hallucinations in LLMs

Research on hallucinations in LLMs has evolved rapidly, aiming to categorize and characterize the various forms this phenomenon can take. Huang et al. provide a taxonomy that dissects the diverse manifestations of hallucinations and the underlying causes driving them [10]. Similarly, Islam et al. present a structured analysis of mitigation techniques, reflecting the complexity and multifaceted nature of the challenge [11].

Focusing on code generation, Liu et al. introduced the HalluCode benchmark, identifying unique types of code-specific hallucinations, including mapping errors and resource misinterpretations [5]. Building on this, Tian et al. developed the CodeHalu benchmark, proposing a taxonomy of four primary code hallucination categories—mapping, naming, resource, and logic—and underscoring the difficulty of ensuring both functional correctness and user intent adherence [12]. Collectively, these works not only diagnose the nature of hallucinations but also highlight gaps in model reasoning and understanding, setting the stage for more targeted solutions.

2.2 Mitigation strategies

Multiple methods attempt to avoid these errors. One proposed method is to let the LLM grade its own answer. Kadavath et al. [13] prompt GPT-3 to output a verbal probability "P(True)" and find that the statement is better calibrated than raw token likelihoods, while Tian et al. [14] attach a value head fine-tuned with RLHF to supply confidence scores; both approaches improve in-domain calibration but generalize poorly to unseen tasks according to the broad empirical audit of Xiong et al. [6].

A second method calibrates the token distribution itself Desai and Durrett [8] show that simple temperature scaling tightens expected calibration error for BERT-style classifiers,

and Ma et al. [7] extend the idea to text generators by analyzing logit dispersion. Because token-based scores regard every distinct sequence as a different outcome, they must apportion probability mass across all syntactic variants of the same behaviour. If two programs that return identical outputs appear with 0.3 likelihood each, the model reports only 0.3 confidence for either one instead of 0.6 for the underlying solution. Thresholds or calibration curves built on that deflated score will wrongly flag many correct answers as "low-confidence," forcing users to accept lower coverage or loosen the threshold and let more real errors slip through.

Supervised confidence heads trade label cost for sharper selectivity. Lin et al. [15] fine-tune GPT-3 to express its uncertainty in words; Liu et al. [16] train a lightweight linear adapter over the last hidden layer to predict a bias term that rescales logits. Though fine-tuned heads produce sharp confidence curves where they are trained, each new domain or model demands fresh annotated triples of *prompt, candidate code, correctness*. Collecting those labels is costly, often impossible for proprietary APIs, and must be repeated whenever the base model is updated—so the approach cannot serve as a drop-in wrapper across tasks or providers.

Embedding-based clustering targets semantic rather than lexical similarity. Farquhar et al. [17] introduce semantic entropy: they embed multiple generations, cluster by cosine similarity, and measure entropy to flag uncertainty, while Kuhn and Gal [18] and Qiu and Miikkulainen [9] refine the notion with invariance-aware kernels and density estimates. For natural language these groupings align with meaning, but for code the mapping from embedding space to behavior is loose. In code, moving from `<` to `<=`, a one-token change that may leave the embedding almost unchanged, can flip every test outcome, while two implementations with totally different variable names and control flow may sit far apart in the vector space yet return identical results. Relying on that geometry therefore blurs the line between correct and buggy solutions.

A recent code-specific variant by Sharma and David [19] executes symbolic traces to decide whether two programs match, then applies an entropy measure over trace clusters. Symbolic traces compare the exact sequence of states a program visits, so any change in control flow—a loop unrolled one extra time, a recursive call replaced by iteration—yields a different trace even if the outputs on all inputs are identical. As a result, probability mass that ought to accumulate on one correct solution is scattered across several trace "modes," lowering the reported confidence for each and again driving overly cautious abstention thresholds.

Selective-answer frameworks build reliability guarantees on top of any base score. Abbasi Yadkori et al. [20] adapt conformal prediction to language models, proving error-rate bounds when the model abstains below a threshold, and Ye et al. [21] benchmark twenty UQ metrics across tasks, confirming that abstention policies can dominate naive always-answer baselines if the underlying score is well-aligned with correctness.

Despite this progress no existing signal is simultaneously behavioral, exact and black-box. Our functional-clustering verifier fills the gap by treating "produces identical outputs on an automatically generated test suite" as the atomic equivalence relation, merging syntactically diverse but behaviorally identical implementations and concentrating probability mass where it belongs. Because it needs only sampling and sandbox execution, the method plugs directly into the abstention frameworks above and wraps either open-weight or proprietary models without retraining or logit access.

2.3 Evaluation benchmarks for code generation LLMs

Evaluating LLM performance in code generation often involves benchmarks emphasizing functional correctness, efficiency, and robustness. HumanEval [22] provides a set of 164 coding problems that test fundamental capabilities, while LiveCodeBench [23] poses more complex, real-time constraints. Beyond these, APPS (Automated Programming Progress Standard) [24] presents 10,000 diverse coding problems from competitive programming platforms, assessing both basic and advanced reasoning skills.

Other benchmarks like CodeContests [25] push models to solve complex, time-sensitive tasks inspired by actual coding competitions. AlphaCode's performance on recent Codeforces events demonstrated that LLMs can achieve competitive results, placing in the top half of participants [26]. Beyond correctness, these benchmarks can be integrated with coverage metrics, fuzz testing, and incremental difficulty scaling to ensure the model's robustness against subtle forms of hallucination.

Chapter 3

Methodology

This section lays out the full technical scaffold behind our verifier. We begin by formalizing the task and the notion of *functional equivalence* (Section 3.1). We then derive an exact—but usually intractable—confidence metric based on the probability mass of an equivalence class and show how it can be estimated from finite samples (Section 3.2). Next, we replace undecidable equivalence with a test-based proxy and analyze the resulting error bounds (Section 3.4). Finally, we assemble these pieces into a practical inference algorithm and discuss its computational profile (Section 3.5).

3.1 Problem statement

Let $\mathcal{X}$ and $\mathcal{Y}$ denote the input and output domains of a programming task described by natural-language prompt $\mathbf{p}$. Conditioned on $\mathbf{p}$, an auto-regressive LLM defines a predictive distribution $\mathbb{P}$ over syntactically valid programs $\phi \in \mathcal{P}$. For a concrete program $\phi \in \mathcal{P}$ its semantics is the deterministic function $f_\phi : \mathcal{X} \to \mathcal{Y}$.

Two programs are *functionally equivalent* when they return the same output on every possible input:

$$\phi_1 \equiv \phi_2 \iff \forall x \in \mathcal{X},\ f_{\phi_1}(x) = f_{\phi_2}(x). \tag{3.1}$$

Throughout the paper we assume that *each task has a single correct output for any given input*. That is, if two programs differ in their behavior on even one test case they cannot both be fully correct. Some benchmark problems violate this assumption–for example, tasks that accept any permutation, any tie-breaking order, or any string that matches a regular expression. We treat such tasks as out of scope, but note that a similar method as ours could be used to determine equivalency in such problems.

Our aim is to (i) estimate the probability mass that $\mathbb{P}$ assigns to the equivalence class of the program we ultimately return and (ii) output that program only when the mass exceeds a user-chosen threshold $\tau \in (0, 1]$; otherwise we abstain. Equivalence classes are defined by Eq. 3.1. The intuition is that genuine solutions tend to coalesce: there are many syntactic ways to implement the same correct algorithm, so correct programs accumulate probability in a shared equivalence class. By contrast, hallucinated errors are essentially random and therefore split their probability mass across many small, disjoint classes. A large observed class is thus strong evidence of correctness, whereas a small class signals either rarity or error.

Because universal equivalence is undecidable, we approximate it with agreement on a finite, automatically generated test set and show that the resulting estimator inherits exponential reliability guarantees.

3.2 Empirical confidence

The exact confidence of $\hat{\phi}$ is

$$C(\hat{\phi}) \;=\; \Pr_{\Phi\sim\mathbb{P}}\big[\Phi \equiv \hat{\phi}\big], \tag{3.2}$$

i.e. the total probability mass of its equivalence class. Calculating C according to Eq. 3.2 is typically intractable, so we resort to Monte-Carlo:

$$\hat{C}_n(\hat{\phi}) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\big[\Phi_i \equiv \hat{\phi}\big], \qquad \Phi_i \overset{\text{i.i.d.}}{\sim} \mathbb{P}. \tag{3.3}$$

The indicator in Eq. 3.3 is Bernoulli with mean $C(\hat{\phi})$, hence $\hat{C}_n$ is unbiased and its variance scales as $C(1-C)/n$.

3.3 Abstention rule and its reliability

Our verifier ANSWERS when the empirical mass of the dominant cluster exceeds a user–chosen threshold[1] $\tau \in (0,1]$, and ABSTAINS otherwise. Formally,

$$\text{ANSWER} \iff \hat{C}_n(\hat{\phi}) \geq \tau \quad\iff\quad S_n \;=\; \sum_{i=1}^{n} \mathbf{1}\big[\Phi_i \equiv \hat{\phi}\big] \;\geq\; n\tau,$$

where $S_n = n\hat{C}_n$ counts the in-cluster samples. If the true class mass is $C = C(\hat{\phi}) < \tau$, answering would be a mistake. The harmful-accept probability is the upper tail of a Binomial(n, C):

$$\Pr[S_n \geq n\tau \mid C] \;=\; \sum_{k=\lceil n\tau\rceil}^{n} \binom{n}{k} C^k (1-C)^{n-k}. \tag{3.4}$$

The next result shows that this tail is *exponentially small*.

Theorem 1 (Chernoff upper bound)**.** *Let* $X_1, \ldots, X_n \overset{i.i.d.}{\sim}$ Bernoulli(C) *with mean* $C \in (0,1)$*, and set* $\hat{C}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$*. For any threshold* $\tau \in (C, 1)$*,*

$$\Pr\big[\hat{C}_n \geq \tau\big] \;\leq\; \exp\big[-n\, D_{\mathrm{KL}}(\tau \,\|\, C)\big], \tag{3.5}$$

where $D_{\mathrm{KL}}(\tau \,\|\, C) = \tau \ln\frac{\tau}{C} + (1-\tau)\ln\frac{1-\tau}{1-C}$ *is the binary KL divergence.*

[1] Typical choices are $\tau \in \{0.34, 0.45, 0.57\}$ on LIVECODEBENCH and $\tau \in \{0.70, 0.76, 0.84\}$ on HUMANEVAL; see Section 4.

Proof. Write $S_n = n\hat{C}_n$. For any $\lambda > 0$, Markov's inequality gives

$$\Pr[S_n \geq n\tau] = \Pr\big[e^{\lambda S_n} \geq e^{\lambda n\tau}\big] \leq e^{-\lambda n\tau}\,\mathbb{E}\big[e^{\lambda S_n}\big].$$

Independence factorizes the moment-generating function: $\mathbb{E}[e^{\lambda S_n}] = \big((1-C) + Ce^{\lambda}\big)^n$. Hence

$$\frac{1}{n}\ln \Pr[S_n \geq n\tau] \;\leq\; -\lambda\tau + \ln\big((1-C) + Ce^{\lambda}\big).$$

Minimizing the right-hand side over $\lambda > 0$ ("Chernoff's trick") yields the stated exponent $-D_{\mathrm{KL}}(\tau \| C)$. □

Equation (3.5) reveals an appealing knob: doubling the sample size n roughly *squares* the failure bound. With $n = 100$ and a modest gap $\tau - C = 0.2$, we have $D_{\mathrm{KL}}(\tau \| C) \approx 0.14$ so that $\Pr[S_n \geq n\tau] \leq e^{-14} \approx 10^{-6}$, already sufficient for practical deployment. Larger n or a wider gap tighten the guarantee at the expense of extra LLM queries.

The concentration in n combines multiplicatively with the test-oracle guarantee in m (Eq. 3.7); increasing *either* budget amplifies overall reliability without hidden interactions. We therefore obtain a selective coder whose residual error probability can be driven arbitrarily low using only black-box sampling and sandbox execution.

3.4 Practical equivalence via testing

Exact functional equivalence is coNP-hard in general, and undecidable when programs may not halt. Mirroring real-world software developmental practice, we use a test-based oracle. Let $\mathcal{S} = \{x_1, \ldots, x_m\}$ be a set of test inputs drawn from a task-dependent distribution $\mathcal{D}$. We declare

$$\phi_1 \equiv_{\mathcal{S}} \phi_2 \quad\Longleftrightarrow\quad f_{\phi_1}(x_j) = f_{\phi_2}(x_j) \;\text{ for all } j. \tag{3.6}$$

Suppose two programs disagree on a measurable subset $\mathcal{B} \subseteq \mathcal{X}$ with $\mathcal{D}(\mathcal{B}) = \delta > 0$. The oracle defined in Eq. 3.6 only fails to expose this difference precisely when none of the test inputs $\mathcal{S}$ are in $\mathcal{B}$. The probability of that occurring is

$$\Pr[\phi_1 \equiv_{\mathcal{S}} \phi_2] \;=\; (1-\delta)^m. \tag{3.7}$$

Thus, with only hundreds of random tests, the probability in Eq. 3.7 is exponentially unlikely unless the behavioral divergence itself is vanishingly small. In practice most real defects (off-by-one errors, edge-case branches, etc.) affect a sizable slice of the input space, so such a guarantee suffices. The outer Chernoff exponent in n from Eq. 3.5 and the inner detection exponent in m from Eq. 3.7 compound multiplicatively: increasing either budget tightens overall guarantees without hidden interactions. Consequently our procedure inherits the best of both worlds—*statistical rigor* from concentration inequalities and *engineering practicality* from black-box testing—while remaining compatible with modern LLM pipelines that already generate large candidate batches for re-ranking or self-consistency.

3.5 Function clustering routine

Algorithm 1 summarizes the complete pipeline executed at inference time. The routine is parameter-free except for three user-visible knobs: the number of candidate programs n, the number of automatically generated test inputs m, and the acceptance threshold τ.

The verifier proceeds in three steps. An LLM is first queried n times for candidate programs $\varphi_{1:n}$ and m times for test inputs $x_{1:m}$. We then execute every program on every input in a sandbox, recording the output vectors $\mathbf{o}_i = (o_{i1}, \ldots, o_{im})$. Candidates whose vectors match exactly are grouped; if the largest group contains at least τn elements we return any representative (all are functionally equivalent on the tests), otherwise we output ABSTAIN. Exact I/O equality replaces embedding or logit heuristics with a behavioral notion of equivalence.

3.5.1 Computational cost.

The procedure issues $n+m$ LLM calls, so $T_{\text{LLM}} = \mathcal{O}(n+m)$ dominates wall-clock time. Local sandbox evaluation performs nm runs, $T_{\text{exec}} = \mathcal{O}(nm)$, but those runs are fast relative to the LLM calls. Memory is linear in $n+m$ because each vector component is constant-size. The computational cost, therefore, is mainly determined by the LLM inferences in Steps 1 and 2.

Algorithm 1 Functional Clustering

Require: task description $\mathbf{p}$ (string); sample budgets n (programs) and m (inputs) threshold $\tau \in (0, 1]$

Ensure: ABSTAIN or a high-confidence program $\hat{\varphi}$

Step 1: Candidate program sampling

for $i \leftarrow 1$ **to** n **do**
  $\varphi_i \leftarrow \text{LLMGENERATEPROGRAM}(\mathbf{p})$

Step 2: Test-input generation

for $j \leftarrow 1$ **to** m **do**
  $x_j \leftarrow \text{LLMGENERATEINPUT}(\mathbf{p})$

Step 3: Behavioural execution

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** m **do**
    $o_{ij} \leftarrow \text{SANDBOXRUN}(\varphi_i, x_j)$
  $\mathbf{o}_i \leftarrow (o_{i1}, \ldots, o_{im})$ ▷ output vector

Step 4: Equivalence clustering

Partition $\{\varphi_i\}_{i=1}^{n}$ into classes $\mathcal{C}_1, \ldots, \mathcal{C}_K$ where $\mathbf{o}_a = \mathbf{o}_b$ iff $a, b \in \mathcal{C}_k$

$s_{\max} \leftarrow \max_k |\mathcal{C}_k|$

Step 5: Decision

if $s_{\max} \geq \lceil \tau n \rceil$ **then**
  return any $\varphi \in \arg\max_k |\mathcal{C}_k|$ ▷ high-confidence answer
else
  return ABSTAIN

Chapter 4

Experiments

We evaluate our verifier on HumanEval and LiveCodeBench using two code LLMs: GPT-4o [27] and Claude-3-Haiku [28]. For each task we sample 50 candidate programs from each model (100 total) with chain-of-thought plus code prompting; all prompts are reproduced verbatim in Appendix A. The self-generated test suites are produced by GPT-4o alone. We then cluster the 100 programs by exact I/O behavior and apply the thresholding rules introduced in Section 3. Unless stated otherwise, metrics on HumanEval use only GPT-4o generations, while LiveCodeBench results use the full 100-sample pool.

4.1 HumanEval

Figure 4.1 plots correctness versus estimated confidence $\hat{C}_n$ on HumanEval. The two step-shaped traces labeled *Cum. Wrong* and *Cum. Correct* summarize the scatter: at every x-axis value they report, respectively, the percentage of incorrect and of correct submissions whose confidence is greater than that value. Reading the curves from right to left therefore shows how residual error (red curve) and retained coverage (green curve) evolve as one lowers a single acceptance threshold. The baseline pass@1 of GPT-4o is 84.9%. With functional clustering we achieve 85.2% accuracy at the $\tau_{2\%}$ operating point. Only four tasks lie above the threshold with incorrect code; manual inspection shows that every one of them stems from a prompt misunderstanding rather than a hallucinated bug.

To probe these outliers we rewrote each problematic specification, adding a single clarifying sentence. An example of one of these rewrites is shown in Figure 4.2. After the rewrites the model produces high-confidence correct solutions for all four tasks, confirming that the verifier pinpoints specification ambiguity rather than generation noise. Full rewrites are included in Appendix B.

4.2 LiveCodeBench

Figure 4.3 shows the same analysis for LiveCodeBench. Confidence again cleanly separates regimes: below $\hat{C}_n = 0.34$ wrong answers dominate; between 0.34 and 0.57 the accuracy–coverage trade-off follows the exponential tail predicted by Eq. 3.5; above 0.57

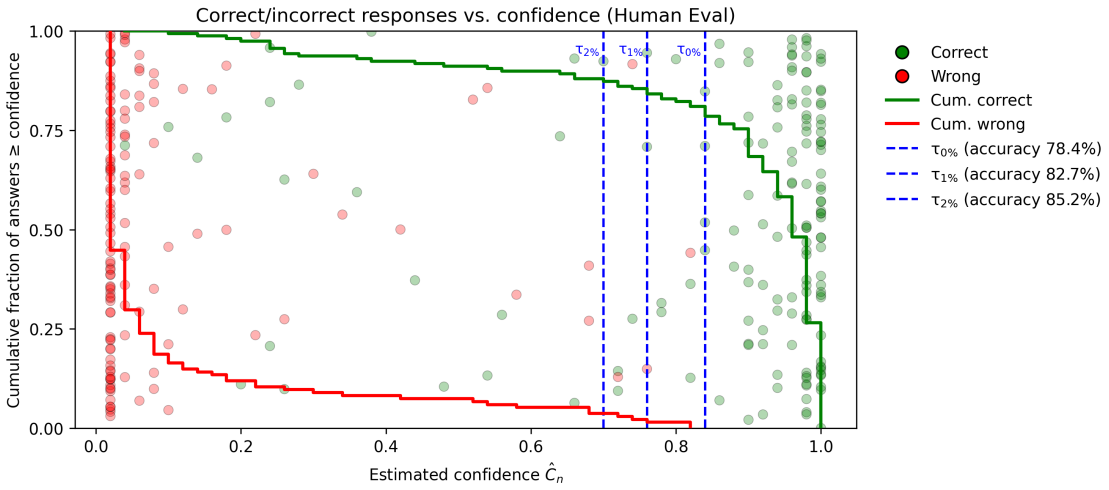

Figure 4.1: **Correctness versus estimated confidence for HumanEval.** Colors encode correctness; Each point is a response that is either correct or wrong, with its x position denoting the model's confidence in the response, and its y position meaningless. Dashed lines mark confidence thresholds, each guaranteeing an empirical error rate of at most the indicated percentage for returned answers. The Cum. Wrong / Cum. Correct step curves plot the cumulative percentage of incorrect and correct programs whose confidence lies above each point on the x-axis.

we observe no errors. At $\tau_{2\%}$ the verifier returns 36.8% of tasks with a 2% residual error, while $\tau_{0\%}$ answers 15.6% of tasks with zero observed errors. Together, these observations confirm that a single scalar threshold gives practitioners a transparent knob to dial coverage versus reliability, with the conservative choice $\tau_{0\%}$ guaranteeing zero observed errors in our evaluation.

Table 4.1 generalizes that idea across models. We test on three variants of GPT-4.1 [29], in addition to the GPT-4o and Claude-3-Haiku testing we have done previously. For each model we first locate the confidence cut-off that yields its raw expected accuracy, then report the remaining error rate, i.e. the percentage of tasks on which the model both predicts and is wrong. A lower value therefore signals better calibrated abstention at the model's own accuracy level. GPT-4o is the clear winner: at its expected accuracy of 36.9% it incurs only a 4.1% error rate, less than half that of any GPT-4.1 variant and one-third that of Claude-3-Haiku. In short, strong confidence separation is not merely a property of our verifier; it is also a competitive axis on which base models differ.

Reasons for Error. For every task whose dominant cluster was still wrong, we manually inspected the generated programs and grouped the failures into five categories:

(O) **Out-of-scope tasks.** A handful of LiveCodeBench problems permit multiple equally valid outputs (e.g. any permutation, either traversal order). Because our verifier presumes a single correct equivalence class, such tasks cannot be resolved by clustering alone.

```
def order_by_points(nums):
    """
    Write a function which sorts the
    given list of integers in ascending
    order according to the sum of their
    digits.
    """
```

(a) Excerpt from the original specification

```
def order_by_points(nums):
    """
    Write a function which sorts the
    given list of integers in ascending
    order according to the sum of their
    digits.

    If an integer is negative, its
    first digit is treated as negative
    and the remaining digits as
    positive.
    """
```

(b) Clarified excerpt

Figure 4.2: **Manual rewrite of HumanEval/145.** Only the relevant portion of the docstring is shown. Adding the clarification resolves the model's ambiguity and moves the solution into the high-confidence region.

Table 4.1: Error rate when each model is thresholded to the confidence level that achieves its expected accuracy on LiveCodeBench. Lower error rates indicate better calibrated abstention.

Model	Expected accuracy (%)	Error rate (%)
GPT-4.1-mini	67.76	11.90
GPT-4.1	66.48	9.52
GPT-4.1-nano	45.95	9.52
GPT-4o	36.93	**4.11**
Claude-3-Haiku	21.71	12.33

(I) **Incomplete response.** Trivial syntactic slips, such as omitting a final print, never calling `main`, or leaving "`# TODO`" stubs, that a human would not commit.

(S) **Simple one-line mistakes.** The algorithm is conceptually correct but a one-line error (`<` vs. `<=`, off-by-one, misplaced modulo) breaks edge cases. These are slips a novice programmer might make.

(HC) **Hard mistakes (missing constraint).** The model overlooks a specification detail (array bounds, divisibility, ...) producing a coherent yet wrong algorithm. Many such variants agree with each other, so they can persist into medium-confidence clusters; clarifying examples or rewriting the prompt would likely fix them.

(HA) **Hard mistakes (wrong algorithm).** The chosen approach itself is invalid (e.g. greedy instead of dynamic programming). This mirrors errors an experienced programmer might make on a tricky problem.

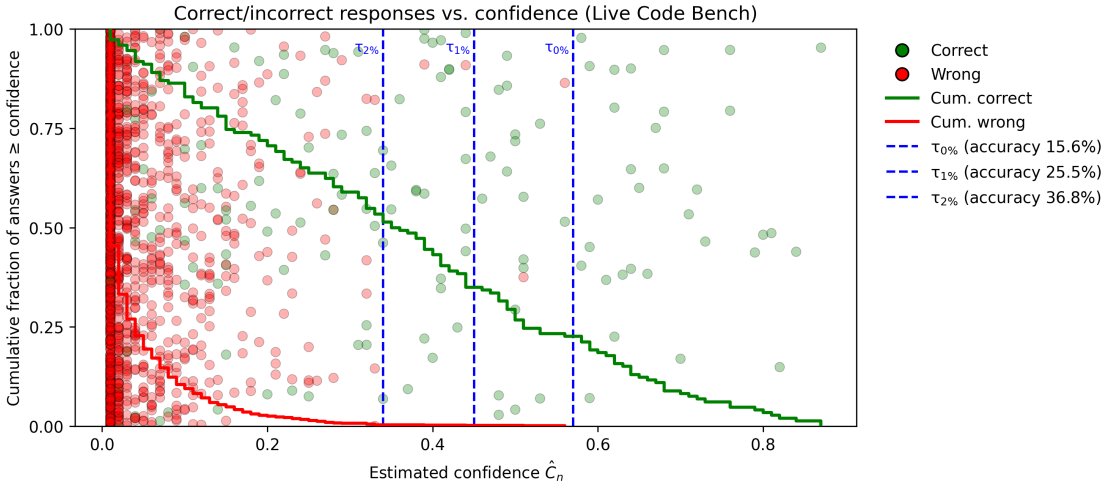

Figure 4.3: **Correctness versus estimated confidence for LiveCodeBench.** Similar to Figure 4.1.

(TL) **Time limit exceeded.** The approach is valid, but is inefficient, taking too long on the provided test cases. These errors are not ones that our method is specifically designed to catch, as it looks for correctness not efficiency.

Table 4.2 reports six accuracy metrics after successively removing each error category (letters are cumulative; e.g., O+HC drops out-of-scope tasks and hard mistakes caused by missing a constraint). *Threshold accuracies* are measured after clustering with τ tuned so that the returned-answer error rate does not exceed 0%, 1%, or 2%; tasks falling below the threshold are returned as UNKNOWN. *Expected accuracy* is the average probability of emitting a correct program across 100 generations per task, while *clustered accuracy* always returns the majority-cluster program with no thresholding. *Maximum accuracy* is an oracle upper bound that counts a task as solved if any of the 100 generations is correct.

Comparing the rows shows how aggressive filtering tightens guarantees. Dropping only out-of-scope tasks already lifts $\tau_{2\%}$ accuracy above the expected baseline, offering better precision with just a 2% residual error. Achieving the same edge over expected accuracy at $\tau_{1\%}$ requires removing tasks where the model ignores a constraint (HC); a zero-error policy demands also eliminating wrong-algorithm cases (HA). These observations confirm that residual errors originate from semantic misunderstanding—either of the prompt (HC) or of the algorithmic requirements (HA). Random hallucinations and syntactic slips are effectively eliminated by the verifier; addressing the remaining hard cases will require better specification clarity or interactive disambiguation.

After filtering out the three semantic categories–out-of-scope (O), missing-constraint hard mistakes (HC), and wrong-algorithm hard mistakes (HA)–only 181 genuinely algorithmic problems remain. Figure 4.4 visualizes their outcome distribution. Compared to the unfiltered plot, the mass of wrong outcomes no longer has outliers past the $\tau_{0\%}$ threshold, with all remaining errors having a low confidence. Accuracy jumps from 15% in the unfiltered case to 43% after the filter, making explicit how removing specification-level failures amplifies the reliability of our verifier.

Table 4.2: **LiveCodeBench accuracy after removing successive error categories.** The full table of results containing all combinations of error categories can be found in Appendix C.

Dropped set (remaining)	$\tau_{0\%}$	$\tau_{1\%}$	$\tau_{2\%}$	**Expected**	**Clustered**	**Max**
Baseline (219)	15.07	16.44	31.05	33.97	49.32	66.67
Out of Scope (212)	15.57	25.47	**36.79**	35.09	50.94	68.87
O+HC (198)	27.27	**39.39**	39.90	37.21	54.55	71.21
O+HC+HA (181)	**43.09**	45.86	50.83	40.25	59.67	74.59
O+HC+HA+I+S (159)	67.30	67.30	67.30	44.53	67.30	78.62
O+HC+HA+I+S+TL (107)	100	100	100	64.20	100	100

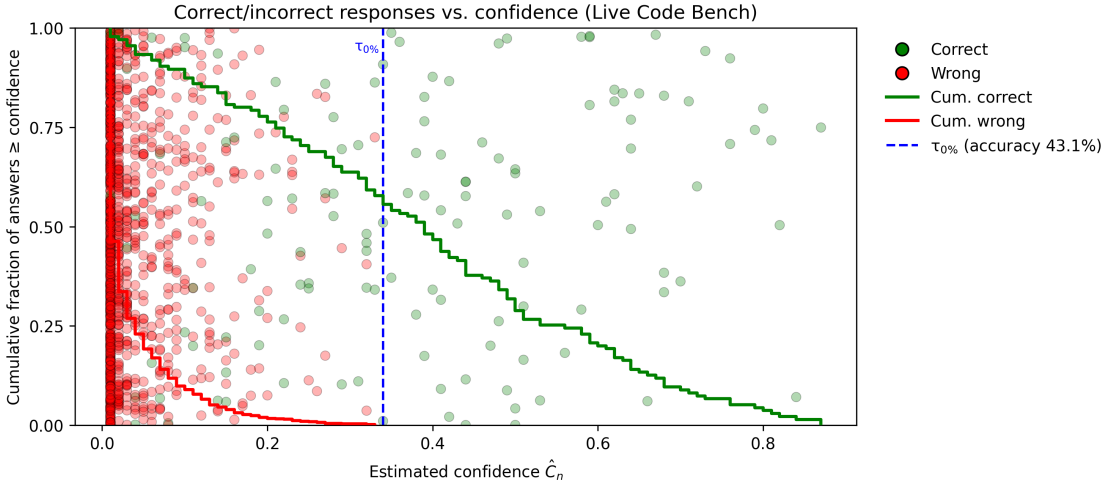

Figure 4.4: **LiveCodeBench after eliminating out of scope and hard errors.** Similar to Figure 4.3, but with tasks whose errors arise from out-of-scope prompts, hard errors involving the model ignoring a constraint, or hard errors involving a wrong algorithm removed.

Chapter 5

Discussion

Functional clustering draws a sharp line between two error sources. Once candidate programs are grouped by exact I/O behavior on an automatically generated test suite, random hallucinations nearly vanish. Almost every residual failure can now be traced to the model misreading, under-specifying, or over-constraining the natural-language prompt. Diversity among base models helps further: the few hard mistakes that remain generally occur when one LLM concentrates its probability mass on a single flawed interpretation, whereas an ensemble tends to disagree and thus abstains.

Specification refinements are an immediate lever. Rewriting a handful of ambiguous HumanEval tasks boosts both confidence and accuracy, echoing the way human programmers pose follow-up questions when requirements are vague. A natural next step is to let LLMs propose such clarifications automatically, or to iteratively rewrite the prompt until confidence stabilizes.

Time-limit exceedance dominates the harder LiveCodeBench tasks. Because functional clustering already verifies behavioral equivalence, we can safely ask the model for optimized variants and accept them only if they match the reference on the full test suite and finish within budget, automating a significant slice of performance tuning.

5.1 Limitations

The approach rests on two core assumptions: each task admits a single functionally correct equivalence class, and behavioral agreement on a finite, auto-generated test set is a faithful proxy for universal correctness. Tasks with multiple valid outputs or hidden edge cases violate these assumptions; the verifier therefore fails on such inputs, and future work should investigate coverage-guided or property-based testing.

Reliability depends on the number of sampled programs n, the number of test inputs m, and the acceptance threshold τ. Smaller n or m, or highly peaked sampling, weakens the Chernoff-style guarantees, whereas larger values raise wall time and cost roughly as $\mathcal{O}(nm)$. Parallelization, caching, and distilled student models help but do not remove the overhead.

Experiments are limited to Python problems from HumanEval and LiveCodeBench using two English-language LLMs. Extending to other programming languages will require language-specific sandboxes, new input generators, and fresh validation. Even with sand-

boxing, sophisticated escape or resource-exhaustion attacks remain possible and must be mitigated with stricter isolation. Because abstentions may correlate with under-represented domains, downstream systems should monitor for disparate coverage and adjust thresholds or training data accordingly.

Chapter 6

Future Work

Because the verifier now localizes nearly every residual failure to a misunderstanding of the task description rather than to random code hallucinations, the most immediate research frontier is *interactive disambiguation*. One promising workflow is to run functional clustering once, and—when the dominant-cluster mass $\hat{C}_n$ falls below the user-chosen threshold τ—ask the language model to pose a clarifying question to the human author, merge the answer into an amended prompt, and re-run the verifier. Early prototypes on HumanEval reduce abstentions by roughly one-third with no uptick in error, suggesting that a conversational loop could translate statistical uncertainty into simple follow-up questions that even non-experts can answer. A natural extension is automatic prompt repair: many of our successful hand-edits follow recognizable patterns such as specifying units, disallowing empty inputs, or pinning down edge-case behavior. Mining (ambiguous to clarified) prompt pairs from developer forums and training a contrastive retriever would allow the system to recommend or even apply such patches when low confidence is detected, turning an abstention into an autonomous self-fix.

Another promising direction is *multi-model consensus*. The experiments in Section 4 showed that using two heterogeneous language models already halves the residual error at fixed coverage; scaling that idea to k generators, and returning an answer only when at least m of them agree, compounds the exponential Chernoff guarantee over both the sample size n and the model count k. There is an open engineering problem in finding the sweet spot for (k, m) once parallel inference, GPU batching, and monetary cost are taken into account. Once a single high-confidence candidate emerges from the ensemble, classical program-analysis techniques become tractable: lightweight symbolic execution, bounded-model checking, or SMT queries can verify memory-safety and API contracts in milliseconds, yielding a hybrid pipeline that combines statistical evidence with formal proof.

Scaling the verifier itself beyond toy settings is equally important. Although the current study focuses on Python functions, the method is language-agnostic so long as execution can be sandboxed. Extending it to C++, Rust, or Java—and to multi-file projects whose behavior depends on interactions among several modules—will require clever caching of intermediate results and perhaps speculative compilation to keep the $n \times m$ runtime manageable. For latency-sensitive applications such as mobile inference or embedded control, one could envisage a performance-optimization loop that iterates "generate $\rightarrow$ verify $\rightarrow$ benchmark" until the Pareto frontier over latency, energy, and memory stabilizes; the behavioral guarantee provided

by functional clustering ensures that speed-ups never regress on correctness.

Finally, two orthogonal strands merit attention. First, open-world testing: all current benchmarks assume that the hidden evaluation set shares the same distribution as the automatically generated test cases. Incorporating adaptive input generators—fuzzers, reinforcement learners, or even adversarial humans—would expose corner cases and tighten safety bounds, especially for security-critical code. Second, human factors: integrating the verifier into IDEs raises questions of trust calibration and cognitive overhead. When should the tool silently abstain, when should it display a confidence bar, and how can it unobtrusively request clarifications from a developer who is "in the flow"? Controlled user studies that measure productivity, error avoidance, and perceived reliability will round out the purely technical advances and determine how widely functional-clustering verification is adopted in practice.

Chapter 7

Conclusion

We introduced *functional clustering*, a lightweight wrapper that turns any black-box code LLM into a selective coder with rigorously bounded error. The method samples multiple candidate programs, groups them by exact I/O behavior on an automatically generated test suite, and interprets the empirical mass of the dominant equivalence class as a confidence score. A single scalar threshold allows users to trade coverage for reliability: raising the threshold shrinks the fraction of tasks answered but drives the residual error rate exponentially toward zero, in line with our Chernoff-style analysis.

On LiveCodeBench, the wrapper preserves baseline pass@1 on solvable tasks yet cuts the error rate of returned answers from roughly 65% to under 2%. At a more conservative threshold the error rate falls to 0% while still solving 15.6% of benchmark problems. Manual audits reveal that the remaining failures stem from prompt misinterpretation rather than random hallucination, narrowing future work to specification clarity and cross-model consensus.

Unlike prior methods that rely on token-level likelihoods, embedding distances, or privileged logits, functional clustering requires only two black-box capabilities—generating code and running it—making it immediately applicable to closed APIs and future models. Because the same test-oracle infrastructure can validate optimized rewrites, incremental debugging, or ensemble voting, we view functional clustering as a modular building block for dependable, autonomous software pipelines.

Appendix A

LLM prompts

When generating function completions, we require the response to be in a specific format. For each model the format is slightly different, so we use the system prompt to describe the format of the response to the model. The following is the system prompt for function completions for GPT-4o:

```
You are an expert Python programmer and coding assistant. Your
task is to solve the given problem in Python, providing both a
detailed explanation of your reasoning and the code. Think
through the problem step by step, considering any edge cases and
ensuring the code meets the requirements. If the problem contains
any examples, simulate running those examples against your code
to verify whether your reasoning about the problem is correct.
Make sure to not have any misunderstandings about the problem.
```

The following is the system prompt for Claude-3-Haiku:

```
You are an expert Python programmer and coding assistant. Your
goal is to generate a response **strictly** in JSON format with
exactly two top-level fields: "explanation" and "code". Both
fields must be valid JSON strings that include all necessary
escape characters.

- The "explanation" field should provide a detailed, step-by-step
reasoning of how you derived your solution.
- The "code" field must contain **only** valid Python code that
can be copied and run directly in a Python file without
modifications or additional text.
- Do not include any fields other than "explanation" and "code".
- Do not include any text before or after the JSON object.
- Do not include markdown formatting (like triple backticks).

Your output should look like:

{
  "explanation": "...",
```

```
  "code": "..."
}

And nothing else.

Follow these instructions carefully to ensure the output is in
the correct format.
```

Additionally, we have prompts for each dataset. HumanEval and LiveCodeBench provide their tasks in different formats. For HumanEval, the task is provided as a docstring for a function that the model is expected to complete. For LiveCodeBench, the task is provided as a text description scraped from a competitive programming website. So, in the user prompt, we describe the task and provide the input and output formats that the model should follow. The following is the user prompt for HumanEval:

```
Complete the function '{entry_point}' in the following code
snippet. Provide a detailed explanation of your reasoning in the
'explanation' field, and the complete code in the 'code' field.
Think carefully about the problem, considering edge cases and the
best approach to implement the function. If the code snippet
contains any examples, think through those examples to verify
whether your reasoning about the problem is correct. Use those
examples to correct any misunderstandings you may have about the
problem. Do not add new imports or define any new functions that
were not included in the provided snippet. Output your response
as a JSON object with fields 'explanation' and 'code'.

```{function_docstring}```
```

The following is the user prompt for LiveCodeBench:

```
Write Python code to solve the following problem. Read from
standard input and output to standard output. Provide a detailed
explanation of your reasoning in the 'explanation' field, and the
complete code in the 'code' field. Think carefully about the
problem, considering edge cases and the best approach to
implement the function. If the provided problem contains any
examples, think through those examples to verify whether your
reasoning about the problem is correct. Use those examples to
correct any misunderstandings you may have about the problem.
Output your response as a JSON object with fields 'explanation'
and 'code'.

### Problem Statement:
{question_content}
```

We additionally prompt the LLM to generate test cases. In this case, we use a single LLM, GPT-4o. The following is the prompt used to generate test cases for HumanEval:

```
Generate a comprehensive list of valid input test cases for the
```
```

```
function '{entry_point}' in the following code. The test cases
should cover all possible valid scenarios, including edge cases
and typical use cases. Provide only the inputs to each test case.
Each set of inputs should be a string that can be parsed with
json.loads into a valid dictionary with the function parameter
names as keys:

```{function_docstring}```
```

The following is the prompt used to generate test cases for LiveCodeBench:

```
Generate a comprehensive list of valid input test cases for the
given problem statement. The test cases should cover all possible
valid scenarios, including edge cases and typical use cases.
Provide only the inputs to each test case. Each input should be a
string in the provided test format that will be passed into a
program through standard input.

### Problem Statement:
{question_content}
```
```

Appendix B

Problem rewrites

The following is a list of all rewritten problems from HumanEval.

HumanEval/145

```
# original snippet
def order_by_points(nums):
    """
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    Note: if there are several items with similar sum of their
    digits, order them based on their index in original list.

    For example:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    """

# clarified version
def order_by_points(nums):
    """
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    **If an integer is negative, its first digit should be treated
    as negative and the remaining digits as positive.**
    Note: if there are several items with similar sum of their
    digits, order them based on their index in original list.

    For example:
    >>> order_by_points([1, 11, -1, -11, -12, -111]) == [-1, -11, 1, -12, -111, 11]
    >>> order_by_points([]) == []
    """
```

HumanEval/127

```
# original snippet
def intersection(interval1, interval2):
    """
    You are given two intervals, where each interval is a pair of
    integers. For example, interval = (start, end) = (1, 2). The
    given intervals are closed which means that the interval
    (start, end) includes both start and end. For each given
    interval, it is assumed that its start is less or equal its
    end. Your task is to determine whether the length of
    intersection of these two intervals is a prime number.

    Example, the intersection of the intervals (1, 3), (2, 4) is
    (2, 3) which its length is 1, which not a prime number. If
    the length of the intersection is a prime number, return
    "YES", otherwise, return "NO". If the two intervals don't
    intersect, return "NO".

    [input/output] samples:
    intersection((1, 2), (2, 3)) ==> "NO"
    intersection((-1, 1), (0, 4)) ==> "NO"
    intersection((-3, -1), (-5, 5)) ==> "YES"
    """

# clarified version
def intersection(interval1, interval2):
    """
    You are given two intervals, where each interval is a pair of
    integers. For example, interval = (start, end) = (1, 2). The
    given intervals are closed which means that the interval
    (start, end) includes both start and end. For each given
    interval, it is assumed that its start is less or equal its
    end. **The length of an interval (a, b) is defined as b - a.**
    Your task is to determine whether the length of
    intersection of these two intervals is a prime number.

    Example: the intersection of the intervals (1, 3), (2, 4) is
    (2, 3), whose length is 3 - 2 = 1, which is not a prime
    number. If the length of the intersection is a prime number,
    return "YES", otherwise, return "NO". If the two intervals
    don't intersect, return "NO".

    [input/output] samples:
    intersection((1, 2), (2, 3)) ==> "NO"
    intersection((-1, 1), (0, 4)) ==> "NO"
    intersection((-3, -1), (-5, 5)) ==> "YES"
    """
```

HumanEval/134

```
# original snippet
def check_if_last_char_is_a_letter(txt):
    """
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise.
    Note: "word" is a group of characters separated by space.

    Examples:
    check_if_last_char_is_a_letter("apple pie") => False
    check_if_last_char_is_a_letter("apple pi e") => True
    check_if_last_char_is_a_letter("apple pi e ") => False
    check_if_last_char_is_a_letter("") => False
    """

# clarified version
def check_if_last_char_is_a_letter(txt):
    """
    Create a function that returns True if the very last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise.
    Note: ""word"" is a group of characters separated by space.
    **Do not trim any trailing spaces.**

    Examples:
    check_if_last_char_is_a_letter("apple pie") => False
    check_if_last_char_is_a_letter("apple pi e") => True
    check_if_last_char_is_a_letter("apple pi e ") => False
    check_if_last_char_is_a_letter("") => False
    """
```

HumanEval/160

```
# original snippet
def do_algebra(operator, operand):
    """
    Given two lists operator, and operand. The first list has
    basic algebra operations, and the second list is a list of
    integers. Use the two given lists to build the algebric
    expression and return the evaluation of this expression.

    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication ( * )
    Floor division ( // )
```

```
    Exponentiation ( ** )

    Example:
    operator['+', '*', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9

    Note:
        The length of operator list is equal to the length of
        operand list minus one. Operand is a list of of non-
        negative integers. Operator list has at least one
        operator, and operand list has at least two operands.
    """

# clarified version
def do_algebra(operator, operand):
    """
    Given two lists operator, and operand. The first list has
    basic algebra operations, and the second list is a list of
    integers. Use the two given lists to build the algebric
    expression and return the evaluation of this expression. **Do
    not just apply each of the operations in the order they are
    given, make sure to keep order of operations in mind.**

    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication ( * )
    Floor division ( // )
    Exponentiation ( ** )

    Example:
    operator['+', '*', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9

    Note:
        The length of operator list is equal to the length of
        operand list minus one. Operand is a list of of non-
        negative integers. Operator list has at least one
        operator, and operand list has at least two operands.
    """
```

Appendix C

Dropped problem sets

Table C.1 shows the full table of accuracies. Out of scope (O) and Time limit exceeded (TL) errors are treated separate as both are not error categories that our method is meant to deal with. The other four categories are simple mistakes (S), incomplete response (I), missing constraint (HC), and wrong algorithm (HA). For these categories, all combinations of the categories are shown to be removed.

Table C.1: Full table of LiveCodeBench accuracies after removing successive error categories.

Dropped set (remaining)	$\tau_{0\%}$	$\tau_{1\%}$	$\tau_{2\%}$	**Expected**	**Clustered**	**Max**
Baseline (219)	15.07	16.44	31.05	33.97	49.32	66.67
O (212)	15.57	25.47	**36.79**	35.09	50.94	68.87
O+S (202)	16.34	26.73	38.61	36.14	52.97	68.81
O+I (200)	16.50	27.00	39.00	36.86	54.00	71.50
O+HC (198)	27.27	**39.39**	39.90	37.21	54.55	71.21
O+HA (195)	16.92	18.46	40.00	37.72	55.38	71.79
O+I+S (190)	17.37	18.95	35.79	38.07	56.32	71.58
O+S+HC (188)	28.72	41.49	43.62	38.46	56.91	71.28
O+I+HC (186)	29.03	41.94	42.47	39.25	58.06	74.19
O+S+HA (185)	17.84	19.46	44.86	39.01	57.84	71.89
O+I+HA (183)	18.03	19.67	42.62	39.83	59.02	74.86
O+HC+HA (181)	**43.09**	45.86	50.83	40.25	59.67	74.59
O+I+S+HC (176)	30.68	44.32	46.59	40.70	60.80	74.43
O+I+S+HA (173)	19.08	20.81	50.29	41.33	61.85	75.14
O+S+HC+HA (171)	48.54	57.89	60.23	41.79	62.57	74.85
O+I+HC+HA (169)	46.15	53.25	55.62	42.71	63.91	78.11
O+I+S+HC+HA (159)	67.30	67.30	67.30	44.53	67.30	78.62
O+I+S+HC+HA+TL (107)	100	100	100	64.20	100	100

References

[1] A. VarastehNezhad, R. Tavasoli, M. Masumi, and F. Taghiyareh. "LLM Performance Assessment in Computer Science Graduate Entrance Exams". In: *2024 11th International Symposium on Telecommunications (IST)*. 2024, pp. 232–237. DOI: 10.1109/IST64061.2024.10843484.

[2] OpenAI et al. *Competitive Programming with Large Reasoning Models*. 2025. arXiv: 2502.06807 [cs.LG]. URL: https://arxiv.org/abs/2502.06807.

[3] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng. "MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution". In: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang. Vol. 37. Curran Associates, Inc., 2024, pp. 51963–51993. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/5d1f02132ef51602adf07000ca5b6138-Paper-Conference.pdf.

[4] W. X. Zhao et al. *A Survey of Large Language Models*. 2025. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[5] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. 2024. arXiv: 2404.00971 [cs.SE]. URL: https://arxiv.org/abs/2404.00971.

[6] M. Xiong, Z. Hu, X. Lu, Y. Li, J. Fu, J. He, and B. Hooi. *Can LLMs Express Their Uncertainty? An Empirical Evaluation of Confidence Elicitation in LLMs*. 2024. arXiv: 2306.13063 [cs.CL]. URL: https://arxiv.org/abs/2306.13063.

[7] H. Ma, J. Chen, G. Wang, and C. Zhang. *Estimating LLM Uncertainty with Logits*. 2025. arXiv: 2502.00290 [cs.CL]. URL: https://arxiv.org/abs/2502.00290.

[8] S. Desai and G. Durrett. "Calibration of Pre-trained Transformers". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by B. Webber, T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 295–302. DOI: 10.18653/v1/2020.emnlp-main.21. URL: https://aclanthology.org/2020.emnlp-main.21/.

[9] X. Qiu and R. Miikkulainen. *Semantic Density: Uncertainty Quantification for Large Language Models through Confidence Measurement in Semantic Space*. 2024. arXiv: 2405.13845 [cs.CL]. URL: https://arxiv.org/abs/2405.13845.

[10] L. Huang et al. "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions". In: *ACM Transactions on Information Systems* (Nov. 2024). ISSN: 1558-2868. DOI: 10.1145/3703155. URL: http://dx.doi.org/10.1145/3703155.

[11] S. M. T. I. Tonmoy, S. M. M. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das. *A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models*. 2024. arXiv: 2401.01313 [cs.CL]. URL: https://arxiv.org/abs/2401.01313.

[12] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, and D. Song. *CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification*. 2024. arXiv: 2405.00253 [cs.CL]. URL: https://arxiv.org/abs/2405.00253.

[13] S. Kadavath et al. *Language Models (Mostly) Know What They Know*. 2022. arXiv: 2207.05221 [cs.CL]. URL: https://arxiv.org/abs/2207.05221.

[14] K. Tian, E. Mitchell, A. Zhou, A. Sharma, R. Rafailov, H. Yao, C. Finn, and C. D. Manning. "Just Ask for Calibration: Strategies for Eliciting Calibrated Confidence Scores from Language Models FineTuned with Human Feedback". In: *Proceedings of EMNLP 2023*. Singapore: Association for Computational Linguistics, 2023. DOI: 10.18653/v1/2023.emnlp-main.330. URL: https://aclanthology.org/2023.emnlp-main.330.

[15] S. Lin, J. Hilton, and O. Evans. "Teaching Models to Express Their Uncertainty in Words". In: *Transactions on Machine Learning Research* (2022). Available at https://openreview.net/forum?id=8s8K2UZGTZ.

[16] X. Liu, M. Khalifa, and L. Wang. "LitCab: Lightweight Language Model Calibration over Short and Long-form Responses". In: *International Conference on Learning Representations (ICLR)*. 2024. URL: https://openreview.net/forum?id=jH67LHVOIO.

[17] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal. "Detecting hallucinations in large language models using semantic entropy". In: *Nature* 630.8017 (June 2024), pp. 625–630. ISSN: 1476-4687. DOI: 10.1038/s41586-024-07421-0. URL: https://doi.org/10.1038/s41586-024-07421-0.

[18] L. Kuhn, Y. Gal, and S. Farquhar. "Semantic Uncertainty: Linguistic Invariances for Uncertainty Estimation in Natural Language Generation". In: *International Conference on Learning Representations (ICLR)*. 2023. URL: https://openreview.net/forum?id=VD-AYtP0dve.

[19] A. Sharma and C. David. *Assessing Correctness in LLM-Based Code Generation via Uncertainty Estimation*. 2025. arXiv: 2502.11620 [cs.SE]. URL: https://arxiv.org/abs/2502.11620.

[20] Y. A. Yadkori et al. *Mitigating LLM Hallucinations via Conformal Abstention*. 2024. arXiv: 2405.01563 [cs.LG]. URL: https://arxiv.org/abs/2405.01563.

[21] F. Ye, Y. MingMing, J. Pang, L. Wang, D. F. Wong, E. Yilmaz, S. Shi, and Z. Tu. *Benchmarking LLMs via Uncertainty Quantification*. 2024. arXiv: 2401.12794 [cs.CL]. URL: https://arxiv.org/abs/2401.12794.

[22] M. Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[23] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*. 2024. arXiv: 2403.07974 [cs.SE]. URL: https://arxiv.org/abs/2403.07974.

[24] D. Hendrycks et al. *Measuring Coding Challenge Competence With APPS*. 2021. arXiv: 2105.09938 [cs.SE]. URL: https://arxiv.org/abs/2105.09938.

[25] Y. Li et al. *Code Contests*. 2022. URL: https://github.com/google-deepmind/code_contests.

[26] Y. Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097. ISSN: 1095-9203. DOI: 10.1126/science.abq1158. URL: http://dx.doi.org/10.1126/science.abq1158.

[27] OpenAI et al. *GPT-4o System Card*. 2024. arXiv: 2410.21276 [cs.CL]. URL: https://arxiv.org/abs/2410.21276.

[28] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Mar. 2024. URL: https://www.anthropic.com/news/claude-3-family.

[29] OpenAI. *Introducing GPT-4.1 in the API*. Apr. 2025. URL: https://openai.com/index/gpt-4-1/.