Confidence-Gated Transductive Test Generation for Code Reranking

Sungjae Lee[1], **Youngsik Yoon**[1], **Seockbean Song**[2]
Siwei Wang[3], **Wei Chen**[3], **Jungseul Ok**[1,2*]
[1]Department of Computer Science and Engineering, POSTECH, South Korea
[2]Graduate School of Artificial Intelligence, POSTECH, South Korea
[3]Microsoft Research Asia, China
{sungjaelee25, ysyoon97, shinebobo, jungseul.ok}@postech.ac.kr,
{siweiwang, weic}@microsoft.com

Abstract

Test case synthesis is crucial for evaluating and ranking programs generated by large language models (LLMs). However, constructing high-quality test cases remains challenging because reliable expected outputs are often difficult to obtain. We propose Confidence-Gated Transductive Test Generation (CoTT), which first uses an efficient inductive procedure and invokes transductive generation only when inductive confidence is low. This adaptive design improves output reliability while allocating extra computation only when needed. On code reranking benchmarks, CoTT outperforms prior baselines across the reported metrics while reducing cost relative to applying transductive generation to every input. These results show that confidence-based allocation of test-time computation provides a favorable efficiency–effectiveness trade-off with a single efficient LLM.

1 Introduction

Test cases are a fundamental tool for evaluating and scoring code (Chen et al., 2021; Austin et al., 2021; Chen et al., 2022; Yu et al., 2024). With the rapid progress of large language models (LLMs) in code generation, test case synthesis has become an increasingly important component for providing execution-based signals to identify better program candidates (Ficek et al., 2025). In practice, this need is especially pronounced when high-quality human-written tests are sparse or unavailable, making synthesized tests a practical source of execution-based feedback (Ni et al., 2023). However, most prior work has focused primarily on improving code generation, often relying either on a limited number of ground-truth tests or on a small set of simply generated tests (Zhong et al., 2024; Yu et al., 2024).

One line of work executes multiple programs generated by an efficient LLM and uses agreement or consensus among them to validate synthesized tests (Chen et al., 2022, 2024). However, when the generated programs themselves are unreliable, such signals may still be insufficient for constructing synthesized test cases with reliable expected outputs. Another line of work uses input-conditioned generation or verifier-guided execution to reason about specific target inputs and assess candidate outputs (Lin et al., 2025; Lee et al., 2025a). While effective, these approaches may require additional training, search, or auxiliary-model computation.

Constructing synthesized test cases with reliable expected outputs using a single efficient LLM, instead of relying on larger and more expensive LLMs, raises two key challenges. First, improving test quality requires more than simply scaling parallel sampling in the same way (Li et al., 2025); when expected-output estimates are unreliable, a more accurate mechanism is needed to refine them through additional computation. Second, this stronger mechanism should be triggered only when necessary, so that additional computation is allocated efficiently (Li et al., 2024; Lee et al., 2025b). In this sense, test construction requires both better expected outputs and deciding when extra computation is worthwhile.

In this work, we propose Confidence-Gated Transductive Test Generation (CoTT), an adaptive method for constructing synthesized test cases with reliable expected outputs using a single efficient LLM as described in Figure 1. Rather than uniformly scaling programs, CoTT allocates additional computation only when stronger expected-output estimation is needed. CoTT combines an *inductive* procedure for expected-output prediction with a *transductive* procedure for input-specific program generation, invoking the latter only when inductive output agreement indicates low confidence. This design remains practical in a single-

*Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 36648–36660
October 24-29, 2026 ©2026 Association for Computational Linguistics

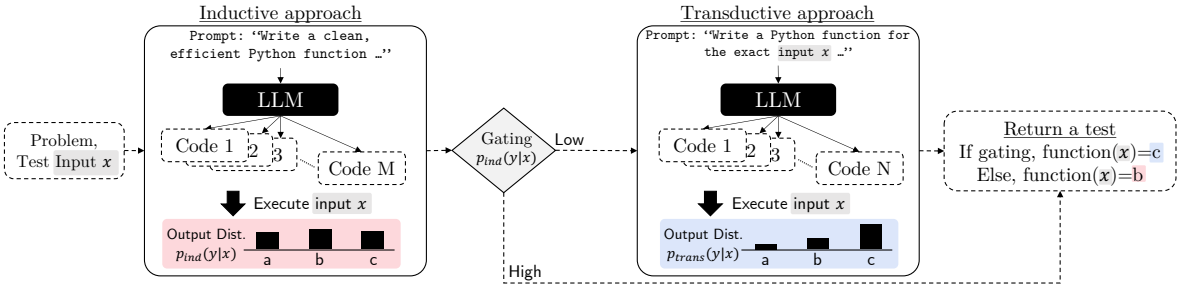


Figure 1: **Overview of CoTT.** CoTT predicts the expected output for a synthesized test input using inductive generation, and invokes transductive generation only when inductive confidence is low. This confidence-gated design improves output reliability while allocating additional computation only when needed.

model setting and improves the reliability of synthesized tests for code reranking. Experiments across datasets show that CoTT improves the expected-output reliability of synthesized tests, leading to better code reranking with a single efficient LLM.

2 Related Work

Synthetic verification and scoring for code. Recent work uses synthetic tests, execution feedback, and learned critics to score or rerank code candidates produced by LLMs (Zeng et al., 2025; Ma et al., 2025; Yu et al., 2024; Chen et al., 2022, 2024). These methods highlight synthesized verification as a key component of downstream code selection, especially when high-quality human-written tests are limited. Recent studies further argue that synthetic verification should be treated as an important evaluation problem for code reranking (Ficek et al., 2025). Our work is most closely related to this line, but treats synthesized test reliability as a key bottleneck for code reranking and evaluates it directly on reranking benchmarks.

Test synthesis and the oracle problem. A core challenge in test synthesis is the *oracle problem*: reliable expected outputs are often harder to obtain than test inputs themselves. Existing approaches to this problem largely fall into two categories. One line of work estimates expected outputs by executing many general solution candidates and aggregating their outputs through agreement or majority voting (Chen et al., 2022, 2024; Liu et al., 2025). Another line of work relies on stronger or input-conditioned procedures to reason about a target input, or to select and verify candidate outputs with stronger external models (Lee et al., 2025a; Lin et al., 2025). While effective, these approaches are tailored either to settings where execution-based consensus is sufficiently reliable or to settings with access to stronger auxiliary models.

3 Method

We propose *Confidence-Gated Transductive Test Generation* (CoTT), an adaptive framework for constructing high-quality test cases with reliable expected outputs using a single efficient LLM. Given a synthesized test input x, CoTT first predicts its expected output with an efficient *inductive* procedure, then invokes a more expensive *transductive* procedure only when the inductive estimate is uncertain. As illustrated in Figure 1, CoTT uses the inductive prediction for high-confidence inputs and switches to a transductive prediction only for low-confidence ones. The resulting expected outputs are used to construct test cases for downstream code scoring and reranking.

Inductive output distribution. For each synthesized test input x, we first estimate an inductive output distribution by sampling a pool of general solution candidates for the original problem with instruction "`Write a clean, efficient Python function ...`" as shown in Figure A1 and executing them on x. Let $n_{\text{ind}}(y; x)$ denote the number of inductive samples whose execution on x returns output y. We define the empirical inductive distribution as

$$p_{\text{ind}}(y \mid x) = \frac{n_{\text{ind}}(y; x)}{\sum_{y'} n_{\text{ind}}(y'; x)}. \quad (1)$$

The corresponding inductive confidence is

$$c(x) = \max_{y} p_{\text{ind}}(y \mid x). \quad (2)$$

Intuitively, $c(x)$ is the majority ratio of the most frequent inductive output. When this value is high, the set of general solution candidates already provides a reliable estimate of the expected output.

Confidence-gated transductive output distribution. When the inductive confidence is low, CoTT triggers a transductive procedure specialized

Test Output Generation Method	HumanEval-R+			MBPP-R+			Avg. Cost ↓
	Top-1 ↑	Bottom-1 ↑	Spearman ↑	Top-1 ↑	Bottom-1 ↑	Spearman ↑	
CoT (Wei et al., 2022)	42.5	48.3	0.440	42.0	43.9	0.349	0.060
SOL-VER (Lin et al., 2025)	41.1	52.1	0.464	42.8	47.1	0.395	0.080
SYNTRA (Lee et al., 2025a)	56.6	59.8	0.585	51.8	56.7	0.539	0.069
rStar-Coder (Liu et al., 2025)	59.8	65.4	0.611	59.8	58.1	0.559	0.058
TT w/o Gating (Ours)	**63.1**	66.2	**0.676**	58.5	56.1	0.576	0.077
CoTT (Ours)	62.5	**66.7**	0.650	**61.6**	**58.7**	**0.580**	0.065

Table 1: Comparison of reranking performance and average cost for test output generation methods on HumanEval-R+ and MBPP-R+. Metrics include Top-1 accuracy, Bottom-1 accuracy, Spearman's ρ, and average cost ($). **Bold** values denote the best results, and underlined values indicate the second-best.

to the target input x using the instruction "`Write a Python function for the exact input ...`" as shown in Figure A2. In this step, the model generates instance-specific programs conditioned on both the problem and the particular target call, executes them on x, and forms a transductive output distribution

$$p_{\text{trans}}(y \mid x) = \frac{n_{\text{trans}}(y; x)}{\sum_{y'} n_{\text{trans}}(y'; x)}, \tag{3}$$

where $n_{\text{trans}}(y; x)$ counts how many transductive samples produce output y on x.

Given a confidence threshold τ, CoTT selects the final expected output by

$$\hat{y}(x) = \begin{cases} \arg\max_y p_{\text{ind}}(y \mid x), & \text{if } c(x) \geq \tau, \\ \arg\max_y p_{\text{trans}}(y \mid x), & \text{if } c(x) < \tau. \end{cases} \tag{4}$$

Thus, CoTT does not combine inductive and transductive distributions; instead, confidence gating selects which distribution to trust for each test input, as illustrated in Figure 1 and Algorithm A1. Given a threshold τ, the inductive prediction is used by default, and transductive generation is invoked only when the inductive confidence is low. In all main experiments, we use a threshold $\tau = 0.8$ across datasets; hyperparameter settings are provided in Table A1. Each synthesized test input x is paired with its predicted output $\hat{y}(x)$ to form a test case $(x, \hat{y}(x))$ for executing and scoring candidate programs in reranking based on pass rate.

4 Experiments

We evaluate CoTT on full datasets of HumanEval-R+ and MBPP-R+ with Llama-3.1-8B-Instruct. Following Ficek et al. (2025), these are code reranking benchmarks derived from HumanEval+ and MBPP+ (Liu et al., 2023), where each task is paired with five candidate solutions of varying correctness and the goal is to recover the ground-truth ranking induced by the reference test suites. Our focus is on *test output generation*: given a shared set of synthesized inputs, each method predicts the corresponding outputs to form test cases, executes all candidate solutions on them, and ranks the candidates by their pass rate on these test cases.

We report Top-1 accuracy, Bottom-1 accuracy, and Spearman's ρ following Ficek et al. (2025), where the first two measure whether the highest- and lowest-ranked candidates match the reference ranking, and Spearman's ρ measures correlation with the full reference ordering. We report the average per-problem cost over both benchmarks, computed from all input and output LLM tokens used for synthesized test construction, including both test-input generation and expected-output generation.[1] Unless otherwise specified, all methods use the same synthesized test inputs, so comparisons isolate differences in test output generation. We construct these inputs to be challenging and discriminative by selecting high-disagreement seeds from temporary program executions and expanding them with type-aware mutation, without using ground-truth solutions.

Baselines. We compare CoTT with representative transductive and inductive baselines for test output generation. **CoT** (Wei et al., 2022) predicts the expected output for each input with a single chain-of-thought pass. **SOL-VER** (Lin et al., 2025) and **SYNTRA** (Lee et al., 2025a) are transductive baselines that perform input-conditioned output generation with multiple CoT samples or output selection among candidates. **rStar-Coder** (Liu et al., 2025) is our inductive baseline: it samples multiple general-purpose programs, executes

[1] Prices for Llama-3.1-8B-Instruct ($0.10 per 1M input tokens and $0.10 per 1M output tokens) were obtained from https://artificialanalysis.ai/models/llama-3-1-instruct-8b (accessed March 16, 2026).

them on each synthesized input, and predicts the expected output by majority vote. In our terminology, this corresponds to the *inductive* approach, since it derives outputs from general candidate programs rather than input-specific reasoning. For baselines that include training or downstream optimization components, we apply only their test-output generation component under the same reranking pipeline.

TT w/o Gating is an always-transductive variant that invokes transductive generation for every synthesized test input and predicts its expected output using $\arg\max_y p_{\text{trans}}(y \mid x)$. This variant isolates the effect of confidence gating.

Experimental Results. Table 1 shows that our methods achieve strong reranking performance relative to prior baselines. On HumanEval-R+, TT w/o Gating obtains the highest Top-1 accuracy and Spearman's ρ, whereas CoTT obtains the highest Bottom-1 accuracy. On MBPP-R+, CoTT achieves the best results across three reranking metrics. Compared with TT w/o Gating, CoTT reduces the average inference cost from \$0.077 to \$0.065, while preserving most of its benefit on HumanEval-R+ and performing better on MBPP-R+. Thus, confidence gating does not uniformly outperform always-transductive generation, but provides a favorable efficiency–effectiveness trade-off. We additionally conduct a sensitivity analysis by sweeping the confidence threshold τ on both benchmarks; the results are provided in Appendix E. Additional analyses of alternative confidence scores and per-problem cost variation are provided in Appendices G and F.

5 Analysis: When Does CoTT Help?

Effectiveness of Confidence Gating. We first analyze why confidence gating improves the efficiency–accuracy trade-off. Recall that CoTT uses the inductive confidence $c(x) = \max_y p_{\text{ind}}(y \mid x)$ to decide whether to trust $\arg\max_y p_{\text{ind}}(y \mid x)$ or invoke transductive refinement and use $\arg\max_y p_{\text{trans}}(y \mid x)$. Figure 2 shows that when only the inductive prediction is correct, $c(x)$ is typically high, so p_{ind} is already reliable and transduction is often unnecessary. In contrast, when only the transductive prediction is correct, $c(x)$ is much lower, indicating that p_{ind} is unreliable and that switching to p_{trans} is beneficial. This supports the inductive majority ratio as an effective trigger for allocating extra computation only when needed.

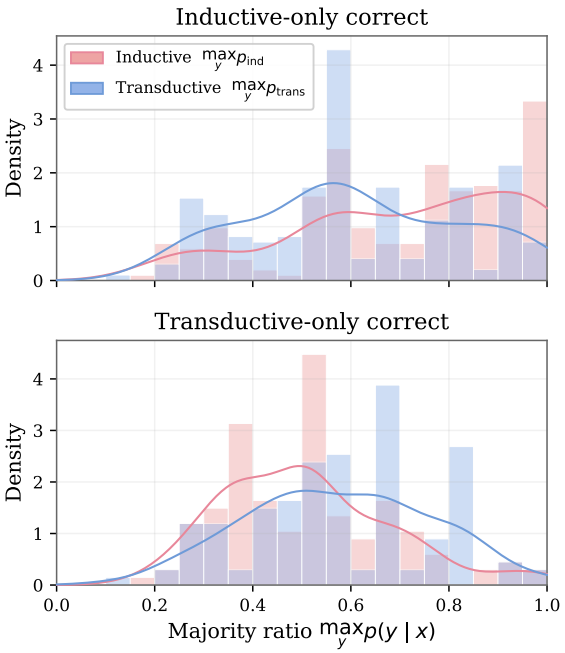


Figure 2: Comparison of the inductive and transductive majority-ratio distributions over synthesized test inputs for which the two predictions disagree. The top panel contains inputs for which only the inductive prediction is correct, whereas the bottom panel contains inputs for which only the transductive prediction is correct. The distributions correspond to $\max_y p_{\text{ind}}(y \mid x)$ and $\max_y p_{\text{trans}}(y \mid x)$, respectively.

Case Study: When Transduction Helps. Consider the MBPP problem "`Write a function to find the Eulerian number a(n, m).`" For the synthesized input `eulerian_num(0, 10)`, whose correct output is `0`, the inductive prediction is `1` while the transductive prediction is `0`. Although neither generated program is correct in general, the transductive program adds an input-specific boundary-case condition that correctly handles `n=0`, whereas the inductive program does not. Thus, even an imperfect transductive program can provide a useful complementary signal by exposing a target-specific error that the inductive prediction alone would miss. This illustrates how p_{trans} can recover the correct expected output even when general inductive solutions are weak. The corresponding prompts and generations are shown in Figures A1, A2, A3, and A4.

Case Study: When Induction Suffices. We also observe the opposite pattern. For the MBPP problem "`Write a python function to find quotient of two numbers (rounded down to the nearest integer).`" and synthesized input `find(10, -2)`, the inductive prediction is correct, whereas the transductive prediction is not. Here the inductive samples consistently recover the correct

rule `dividend // divisor`, while the transductive program unnecessarily introduces a negative sign, producing a rule like `a // -b`. The inductive executions are consistent on this input, yielding $c(x) = 1.0$. This means that CoTT would retain $\arg\max_y p_{\text{ind}}(y \mid x)$ rather than invoking transduction. This example shows that confidence gating preserves the efficient inductive path when the inductive prediction is reliable. The corresponding generations are shown in Figures A5 and A6.

6 Conclusion

We propose CoTT, an adaptive framework for constructing synthesized test cases for code evaluation and reranking with a single efficient LLM. CoTT selectively invokes transductive generation only when inductive confidence is low, allocating additional computation to uncertain test inputs. Across both datasets, CoTT provides a favorable efficiency–effectiveness trade-off by maintaining strong reranking performance at a lower cost than always-transductive generation. An interesting direction for future work is to co-evolve code and test generation using CoTT's constructive feedback to iteratively improve both programs and tests.

Limitations

Our study has several limitations. First, although CoTT does not rely on larger external LLMs, it may further benefit from external information sources, such as relevant code documentation or retrieval-augmented context, when available. Second, we do not explore integrating CoTT with code generation to develop a co-evolving code-and-test generation mechanism. Higher-quality synthesized tests may provide useful feedback for iterative code improvement in such a setting. Third, we evaluate CoTT using a single efficient LLM on two function-level Python code-reranking benchmarks. This controlled setting isolates test-oracle generation, but the results do not establish cross-model robustness or generalization to more realistic repository- or project-level tasks. Fourth, CoTT targets sandboxed code-reranking settings in which generated programs are executed under timeouts. The method may therefore be less applicable when code execution is limited, unavailable, or computationally expensive.

Acknowledgements

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (RS-2019-II191906, Artificial Intelligence Graduate School Program (POSTECH); RS-2024-00436680, Global Research Support Program in the Digital Field; RS-2024-00509258, Global AI Frontier Lab). This work was also supported by the IITP (Institute of Information & Communications Technology Planning & Evaluation)-ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2026-RS-2024-00437866). This project was also supported by Microsoft Research Asia.

AI Assistants

We used large language models solely to improve the manuscript's clarity, grammar, and style, within the scope of "Assistance purely with the language of the paper". All scientific claims, experimental designs, and empirical results reported in this paper are the original work of the authors.

References

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

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

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

Aleksander Ficek, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. 2025. Scoring verifiers: Evaluating synthetic verification for code and reasoning. *arXiv preprint arXiv:2502.13820*.

Kang-il Lee, Jahyun Koo, Seunghyun Yoon, Minbeom Kim, Hyukhun Koh, Dongryeol Lee, and Kyomin Jung. 2025a. Program synthesis via test-time transduction. *arXiv preprint arXiv:2509.17393*.

Sungjae Lee, Hyejin Park, Jaechang Kim, and Jungseul Ok. 2025b. Semantic exploration with adaptive gating for efficient problem solving with language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 588–606.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. In *EMNLP (Findings)*, pages 15964–15978.

Yiwei Li, Peiwen Yuan, Shaoxiong Feng, Boyuan Pan, Xinglin Wang, Bin Sun, Heda Wang, and Kan Li. 2024. Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. *arXiv preprint arXiv:2401.10480*.

Zi Lin, Sheng Shen, Jingbo Shang, Jason E Weston, and Yixin Nie. 2025. Learning to solve and verify: A self-play framework for mutually improving code and test generation. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. 2025. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6917–6935.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Xingru Jiang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. 2024. Reasoning through execution: Unifying process and outcome rewards for code generation. *arXiv preprint arXiv:2412.15118*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870.

Appendix

A CoTT Algorithm

Algorithm A1 presents CoTT and its key components: the inductive approach, confidence gating, and the transductive approach for constructing reliable test outputs.

Algorithm A1 Confidence-Gated Transductive Test Generation (CoTT)

Require: Problem prompt p, synthesized test inputs $\mathcal{T}$, LLM $\mathcal{M}$, number of samples $N_{\text{ind}}, N_{\text{trans}}$, threshold τ
Ensure: Synthesized test cases $\hat{\mathcal{T}} = \{(x, \hat{y}(x)) : x \in \mathcal{T}\}$

Sample inductive programs $\{g_j\}_{j=1}^{N_{\text{ind}}} \leftarrow \mathcal{M}(p)$
for each test input $x \in \mathcal{T}$ **do**
  Execute $\{g_j\}$ on x and form $p_{\text{ind}}(y \mid x)$ from output frequencies
  Compute inductive confidence $c(x) = \max_y p_{\text{ind}}(y \mid x)$
  if $c(x) \geq \tau$ **then**
    $\hat{y}(x) \leftarrow \arg\max_y p_{\text{ind}}(y \mid x)$
  else
    Sample transductive programs $\{h_k\}_{k=1}^{N_{\text{trans}}} \leftarrow \mathcal{M}(p, x)$
    Execute $\{h_k\}$ on x and form $p_{\text{trans}}(y \mid x)$ from output frequencies
    $\hat{y}(x) \leftarrow \arg\max_y p_{\text{trans}}(y \mid x)$
  end if
end for
return $\hat{\mathcal{T}} = \{(x, \hat{y}(x)) : x \in \mathcal{T}\}$

B Experimental Settings

The hyperparameters used in our experiments are listed in Table A1.

Hyperparameter	Value
Test Construction and Code Generation	
Number of test cases per problem	10
Number of inductive code samples	20
Number of transductive code samples	10
Timeout per test case (seconds)	3
Threshold for confidence gating	0.8
LLM Settings	
Maximum context length (tokens)	18,000
Maximum generated tokens per round	1,500
Temperature	0.5

Table A1: Default hyperparameter settings used in our experiments. The hyperparameters are grouped into test construction and code generation settings, and LLM settings.

C Prompt

For code generation in the inductive approach, we follow the prompting style of Yu et al. (2024). Specifically, the model is prompted to generate a general-purpose solution to the original programming problem. For code generation in the transductive approach, we use a modified version of this prompt that is adapted to our setting. Instead of generating a solution intended to generalize broadly, the model is instructed to produce code specialized to the given target input. The full prompts used for inductive and transductive code generation are shown in Figures A1 and A2, respectively.

D Generated Output

To illustrate the behavioral difference between the two prompting schemes, we provide actual generation outputs for two MBPP examples. Figures A3 and A4 show the inductive and transductive generation outputs, respectively, for MBPP problem 103. Figures A5 and A6 show the corresponding inductive and transductive generation outputs for MBPP problem 292. These examples highlight the difference between the two settings. The inductive prompt tends to elicit solutions intended to solve the original problem in a general manner, whereas the transductive prompt more readily produces code specialized to the given input instance.

E Threshold Sensitivity Analysis

CoTT uses the maximum probability of the inductive output distribution,

$$c(x) = \max_y p_{\text{ind}}(y \mid x),$$

as its confidence score. Given a threshold τ, CoTT uses the inductive prediction when $c(x) \geq \tau$ and invokes transductive generation otherwise. We evaluate $\tau \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ on HumanEval-R+ and MBPP-R+ using Llama-3.1-8B-Instruct. We additionally report the inductive-only and always-transductive variants as the two computational endpoints.

As τ increases, the transductive rate and inference cost increase monotonically. On HumanEval-R+, reranking performance improves up to approximately $\tau = 0.8$–0.9 and then largely plateaus,

Dataset	Setting	Top-1↑	Bottom-1↑	Spearman's ρ↑	Trans. rate	Cost ($)↓
HumanEval-R+	Inductive only	59.8	65.4	0.611	0.000	0.0578
	CoTT, $\tau = 0.5$	60.8	66.5	0.629	0.132	0.0603
	CoTT, $\tau = 0.6$	60.8	65.9	0.627	0.198	0.0616
	CoTT, $\tau = 0.7$	62.3	66.2	0.638	0.292	0.0634
	CoTT, $\tau = 0.8$	62.5	66.7	0.650	0.380	0.0651
	CoTT, $\tau = 0.9$	62.5	66.8	0.651	0.529	0.0679
	CoTT, $\tau = 1.0$	62.5	66.8	0.651	0.683	0.0708
	Always transductive	63.1	66.2	0.676	1.000	0.0769
MBPP-R+	Inductive only	59.8	58.1	0.559	0.000	0.0581
	CoTT, $\tau = 0.5$	59.9	58.5	0.564	0.136	0.0607
	CoTT, $\tau = 0.6$	60.9	58.9	0.574	0.210	0.0620
	CoTT, $\tau = 0.7$	61.5	58.7	0.579	0.279	0.0634
	CoTT, $\tau = 0.8$	61.6	58.7	0.580	0.343	0.0645
	CoTT, $\tau = 0.9$	61.6	58.7	0.578	0.502	0.0675
	CoTT, $\tau = 1.0$	61.4	58.1	0.574	0.660	0.0705
	Always transductive	58.5	56.1	0.576	1.000	0.0769

Table A2: Sensitivity of CoTT to the confidence threshold τ. A larger threshold routes more synthesized test inputs to transductive generation. The transductive rate denotes the fraction of test inputs for which transductive generation is invoked. The inductive-only and always-transductive variants represent the two computational endpoints.

Method	Cost ($, mean $\pm$ std)
SOL-VER	0.080 ± 0.024
TT w/o Gating	0.077 ± 0.024
SYNTRA	0.069 ± 0.025
CoTT	0.065 ± 0.023
rStar-Coder	0.058 ± 0.023

Table A3: Mean and standard deviation of per-problem LLM-token cost over HumanEval-R+ and MBPP-R+. The reported variation is across problems.

although always-transductive generation obtains the highest Top-1 accuracy and Spearman's ρ at a higher cost. On MBPP-R+, $\tau = 0.8$ achieves a strong balance between reranking quality and cost, while further increasing τ provides no consistent improvement. These results show that τ controls the efficiency–effectiveness trade-off and should be selected according to the available computational budget.

F Cost Statistics

We further examine across-problem cost variation and Table A3 reports the mean and standard deviation of per-problem cost over HumanEval-R+ and MBPP-R+. The standard deviation measures variation across problems. Although per-problem costs vary substantially, paired comparisons on the same problems show a consistent cost reduction. For the closest comparison, the mean paired cost difference between SYNTRA and CoTT is \$0.0039 per problem, with a standard error of \$0.0002.

G Confidence-Score Ablation

We compare four confidence scores computed from the inductive samples: maximum probability, top-2 margin, negative entropy, and subset agreement. The top-2 margin is the difference between the largest and second-largest probabilities in p_{ind}, negative entropy is $-H(p_{\text{ind}})$, and subset agreement measures how often two random halves of the inductive samples select the same modal output. For every score, a larger value indicates higher inductive confidence, and CoTT uses the inductive prediction when the score meets or exceeds its corresponding threshold. We sweep the threshold for each score and report its best-performing operating point in Table A4.

Maximum probability, top-2 margin, and negative entropy achieve similar performance across the two benchmarks. Top-2 margin slightly reduces the transductive rate and cost relative to maximum probability while preserving or modestly improving reranking performance. Subset agreement invokes transductive generation less frequently and reduces cost further, but yields slightly weaker results. These results show that the confidence scores achieve similar peak performance under score-specific thresholds on the two benchmarks.

Dataset	Confidence score	Threshold	Top-1↑	Bottom-1↑	Spearman's ρ↑	Trans. rate	Cost ($)↓
HumanEval-R+	Maximum probability	0.8	62.5	66.7	0.650	0.380	0.0651
	Top-2 margin	0.6	62.5	66.7	0.650	0.340	0.0643
	Negative entropy	-0.7	62.2	66.7	0.648	0.427	0.0660
	Subset agreement	0.9	62.3	66.5	0.639	0.214	0.0619
MBPP-R+	Maximum probability	0.8	61.6	58.7	0.580	0.343	0.0645
	Top-2 margin	0.6	61.8	58.7	0.581	0.301	0.0638
	Negative entropy	-0.7	61.6	58.7	0.580	0.393	0.0655
	Subset agreement	0.95	61.2	58.7	0.579	0.229	0.0624

Table A4: Comparison of alternative confidence scores for gating. Thresholds are specific to each confidence score and are therefore not directly comparable across scores. Maximum probability, top-2 margin, and negative entropy achieve similar reranking performance, while subset agreement provides a lower-cost but slightly weaker point.

````
You are a Python programmer in a code interview. You're solving a coding problem while sharing
     your thoughts and discussing with an critic. Always follow the format below for each
    round:

1. First, share your thoughts as comments:
   - Your current approach (or an analysis of the problem and your plan if you haven't written
     any code yet)
   - What you learned from previous feedback (if there is any previous feedback, otherwise
    think about your plan, what might be missing from your previous plan)
   - Why you chose this approach (or how you plan to tackle the problem), do you need to shift
     your approach?
   - Be clear, concise, detailed and pay attention to the comments given by the critic, no
    chit-chat, no small talk.
   - Always use first person, e.g. I, we, our, etc.

2. Then write your solution:
   - Clean, efficient Python code that follows requirements exactly
   - No test cases in code, just the solution
   - Your code will then be tested by the critic, so do not include any test cases in your
    code, this is very important

Format your response as:
# === BEGIN PROGRAMMER THOUGHTS ===
# [Your response to previous feedback]
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
```python
[Your code]
```
# === END SOLUTION ===

Output strictly as shown above.

Here's your coding task:
[Problem Description]

I will test your implemented code together with the hidden test cases. Please implement code.

(Reference) Below are dataset test cases for guidance only.
[Test case examples]

IMPORTANT: You MUST define a Python function named [entry_point] (case-sensitive). Do not
    rename it.

Do not include test cases in your code.
Design an object-oriented Python solution. Explain your class structure in comments.


Think carefully and write your solution strictly following our format and rules.
````

Figure A1: Inductive code generation prompt. Following Yu et al. (2024), the model is prompted to generate a general-purpose solution to the original programming problem, with reasoning expressed in comments before the final code.

````
You are a Python programmer.

You will be given a coding problem and ONE specific TARGET CALL.
This is a transductive setting: you are allowed to condition your solution on the exact inputs
     appearing in the TARGET CALL.
Your objective is ONLY to write Python code so that the TARGET CALL returns the correct output
     when executed.

How to use the transductive advantage (recommended):
- Tailor the implementation to this exact input instance (its values, sizes, structure,
    reachable branches).
- You may ignore edge cases, constraints, and generalization to other inputs that do NOT occur
     in this TARGET CALL.
- Do NOT over-generalize unless a fully general solution is genuinely simpler than a tailored
    one.
- Pick any approach that is simplest for this instance (brute force, simulation, DP,
    precomputation, unrolling, etc.).

Hardcoding policy:
- Special-casing the exact input is allowed, but avoid blind guessing of the final answer.
- Prefer computing the output programmatically from the given input, even if the code is
    narrowly tailored to this instance.
- Use direct constants only when they are clearly correct and less error-prone than computing.

Rules:
- Correctness for the TARGET CALL is the ONLY objective (ignore other possible inputs).
- If "reference tests" are shown in the prompt, they are ONLY for I/O format guidance (NOT
    evaluated).
- Ensure the TARGET CALL runs without errors: define all required functions/classes/imports.
- No tests, no prints, no extra text outside the required format.

Format your response EXACTLY as:
# === BEGIN PROGRAMMER THOUGHTS ===
# (comments, first person, concise)
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
```python
[Your code]
```
# === END SOLUTION ===

Output strictly as shown above.

Here's your coding task:
[Problem Description]

Focus only on the TARGET CALL below. No other tests will be run.
Your goal is NOT to solve the whole problem generally.
Your goal is to write Python code so that the TARGET CALL returns the correct output.

IMPORTANT: You MUST define a Python function named [entry_point] (case-sensitive). Do not
    rename it.

(Reference) Dataset tests below are ONLY for I/O format guidance (NOT evaluated).
[Test case examples]

Do not include test cases in your code.

TARGET CALL (this is the only thing that must be correct):
[Target test input]

Focus on making the TARGET CALL correct. You may tailor the code to this exact call; do not
    over-generalize.
Write your response strictly following the required format.
````

Figure A2: Transductive code generation prompt. The prompt is adapted from the inductive version to explicitly condition on a single target call, instructing the model to generate code specialized to the given input instance rather than a broadly general solution.

```python
# === BEGIN PROGRAMMER THOUGHTS ===
# I understand that I need to write a function named eulerian_num that takes two parameters, n
    and m, and returns the Eulerian number a(n, m).
# From the reference test cases, I see that the function is called with different combinations
    of parameters, so I will design a generic solution that can handle any positive integers
    n and m.
# I will use the formula for the Eulerian number, which is the binomial coefficient (n + m - 1
    choose m - 1).
# This formula is a standard mathematical expression for the Eulerian number, and it can be
    easily computed using the math library's comb function.
# I will also include a check to ensure that n and m are non-negative integers, as the
    Eulerian number is not defined for negative numbers or non-integer values.
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
import math

def eulerian_num(n, m):
    """
    Compute the Eulerian number a(n, m) using the formula (n + m - 1 choose m - 1).

    This function is designed to be generic and adaptable for similar problems. The formula
    for the Eulerian number
    is a standard mathematical expression that can be easily computed using the math library's
     comb function. The
    function also includes a check to ensure that n and m are non-negative integers, as the
    Eulerian number is
    not defined for negative numbers or non-integer values.

    Parameters:
    n (int): A non-negative integer.
    m (int): A non-negative integer.

    Returns:
    int: The Eulerian number a(n, m).
    """
    # Check if n and m are non-negative integers
    if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0:
        raise ValueError("n and m must be non-negative integers")

    # Compute the Eulerian number using the formula (n + m - 1 choose m - 1)
    return math.comb(n + m - 1, m - 1)
# === END SOLUTION ===
```

Figure A3: Actual output generated by the inductive prompt for MBPP problem 103. The generated program is evaluated on the target input `eulerian_num(0, 10)`.

```python
# === BEGIN PROGRAMMER THOUGHTS ===
# I can directly calculate the Eulerian number for the given inputs (0, 10) using the formula
    a(n, m) = n + m * a(n - 1, m - 1), with a base case of a(0, m) = 0 when m > 0.
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
def eulerian_num(n, m):
    if n == 0 and m > 0:
        return 0
    elif n == 0:
        return 1
    else:
        return n + m * eulerian_num(n - 1, m - 1)
# === END SOLUTION ===
```

Figure A4: Actual output generated by the transductive prompt for MBPP problem 103, conditioned on the target input `eulerian_num(0, 10)`.

```
# === BEGIN PROGRAMMER THOUGHTS ===
# My initial approach is to use the built-in division operator in Python, which returns a
    float result by default. I will use the `//` operator to perform integer division and
    round down to the nearest integer.
# I've learned from previous feedback that I should always follow the problem requirements and
     provide well-commented code. In this case, I will include a docstring to explain my
    approach and ensure the function is clearly named as required.
# I will use the `//` operator because it is a more efficient and Pythonic way to perform
    integer division, which is exactly what I need in this problem.
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
def find(dividend: int, divisor: int) -> int:
    """
    This function calculates the quotient of two numbers and returns the result as an integer,
    rounded down to the nearest whole number.

    Args:
    dividend (int): The number being divided.
    divisor (int): The number by which we are dividing.

    Returns:
    int: The quotient of the two numbers, rounded down to the nearest integer.
    """
    # Check for division by zero
    if divisor == 0:
        raise ValueError("Divisor cannot be zero")

    # Use the // operator for integer division, which rounds down to the nearest whole number
    quotient = dividend // divisor

    return quotient
# === END SOLUTION ===
```

Figure A5: Actual output generated by the inductive prompt for MBPP problem 292. The generated program is evaluated on the target input `find(10, -2)`.

```
# === BEGIN PROGRAMMER THOUGHTS ===
# I need to write a function that takes two numbers and returns their quotient rounded down to
     the nearest integer.
# Since the second input is negative, I should use integer division (//) to get the quotient.
# === END PROGRAMMER THOUGHTS ===
# === BEGIN SOLUTION ===
def find(a, b):
    return a // -b
# === END SOLUTION ===
```

Figure A6: Actual output generated by the transductive prompt for MBPP problem 292, conditioned on the target input `find(10, -2)`.