

LogitsCoder: Towards Efficient Chain-of-Thought Path Search via Logits Preference Decoding for Code Generation

Jizheng Chen^{1*}, Weiming Zhang^{1*}, Xinyi Dai¹, Lingyue Fu¹, Kounianhua Du¹
Qingyao Li¹, Yong Yu¹, Weiwen Liu^{1†}, Weinan Zhang^{1†}

¹Shanghai Jiao Tong University
Shanghai, China

{humihuadechengzhi, WeimingZhang_2020, wwliu}@sjtu.edu.cn

Abstract

Code generation requires executable reasoning, making test-time chain-of-thought (CoT) search an attractive strategy for improving solution quality. However, existing CoT scaling methods often face a tension between underthinking, where shallow reasoning misses important algorithmic cases, and overthinking, where verbose or redundant reasoning consumes substantial tokens without improving correctness. We propose LogitsCoder, a lightweight framework that revisits CoT search from the perspective of logits-level control. LogitsCoder improves efficient reasoning search through two complementary mechanisms: Logits Preference Decoding adaptively calibrates local token expansion using public-test feedback, while Logits Rank Based Path Selection explores and selects compact high-confidence reasoning trajectories based on logit-level stability. Together with Thoughts Aggregation, LogitsCoder refines reasoning paths without relying on costly MCTS-style tree expansion. Extensive experiments demonstrate that LogitsCoder produces more efficient and higher-quality reasoning chains, leading to superior code generation performance compared to baseline methods.

1 Introduction

Large language models (LLMs) have demonstrated remarkable success across various domains, with code generation emerging as a particularly challenging task that requires both semantic understanding and executable reasoning (Fu et al., 2023; Jain et al., 2024; Zhang et al., 2025b). Unlike open-ended text generation, code generation admits execution feedback, making it possible to evaluate candidate solutions against test cases. Recent methods therefore increasingly rely on test-time reasoning

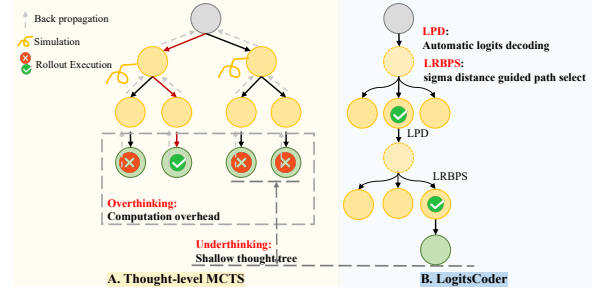


Figure 1: (A) Challenges faced by MCTS: shallow search trees causing underthinking, and excessive rollouts leading to overthinking and high computation cost. (B) LogitsCoder framework addressing these challenges with LPD and LRBPS for efficient and deeper reasoning paths.

strategies, such as iterative refinement and Chain-of-Thought (CoT) exploration, to search for better solution paths (Zhang et al., 2023; Zhong et al., 2024b).

However, CoT-based test-time scaling faces two practical failure modes in code generation. The first is **underthinking**: a model may follow a single shallow reasoning trajectory and miss necessary case analysis, algorithmic invariants, or edge conditions. The second is **overthinking**: increasing the amount of reasoning or rollouts may produce verbose, repetitive, or low-value thoughts, consuming substantial tokens without improving executable correctness. These two failure modes reveal a tension in efficient CoT search: the model must explore beyond a single linear path, but it must avoid spending computation on unproductive local expansions.

Structured search methods such as Monte Carlo Tree Search (MCTS) attempt to address underthinking by expanding multiple reasoning paths (Zhou et al., 2023; Zhang et al., 2024; Li et al., 2024b). However, as illustrated in Figure 1, practical MCTS-style reasoning still faces two practical bottlenecks: (1) it often produces *shallow trees* under

*Equal Contribution.

†Corresponding author.

limited rollout budgets, leaving complex algorithmic cases insufficiently explored; (2) when more rollouts are allocated, its repeated expansion, simulation, and backpropagation introduce substantial *computational overhead*, often amplifying redundant or weak local thoughts. These limitations suggest that efficient CoT search should not rely solely on heavier tree expansion, but should instead control both the quality of local reasoning expansions and the selection of global reasoning trajectories.

In this work, we propose LogitsCoder, a lightweight framework that revisits CoT search from the perspective of logits-level control. Instead of expanding a full search tree, LogitsCoder improves search efficiency by guiding local token expansion before trajectory generation and selecting confident reasoning paths after exploration. Motivated by this insight, LogitsCoder decomposes efficient CoT search into two complementary decisions: *which local reasoning tokens should be encouraged during expansion*, and *which global reasoning trajectory should be retained after exploration*. For the first decision, Logits Preference Decoding (LPD) converts public execution feedback from candidate reasoning-code trajectories into token-level logit preferences, steering generation away from vague or repetitive thoughts. For the second decision, Logits Rank Based Path Selection (LRBPS) explores multiple plausible CoT trajectories and selects a compact high-confidence path using logit-level stability, mitigating shallow single-path underthinking without constructing a costly MCTS-style search tree. In addition, Thoughts Aggregation combines complementary candidate paths when a single selected trajectory may miss useful information, improving robustness while preserving token efficiency.

The contributions of our work are summarized as follows:

- We propose LogitsCoder, a lightweight CoT search framework for code generation that revisits thought generation and thought refinement from the perspective of logits-level control, reducing the need for costly MCTS-style tree expansion.
- We introduce two complementary logits-control modules to mitigate inefficient CoT search: Logits Preference Decoding (LPD) calibrates local token expansions with public execution feedback to reduce overthinking, while Logits Rank Based Path Selection (LRBPS) explores and selects

compact high-confidence reasoning trajectories to alleviate underthinking.

- We evaluate LogitsCoder on different contest-level code generation tasks, achieving high-quality CoT trajectories with fewer tokens than strong baselines. This demonstrates LogitsCoder’s effectiveness and efficiency in complex code generation scenarios.

2 Related Work

LLMs for Code Generation LLMs have demonstrated strong code generation capabilities by modeling both natural language specifications and programming patterns. Existing methods mainly fall into two lines: training-based approaches that fine-tune LLMs on large-scale code corpora (Luo et al., 2023; Wang et al., 2023; Hui et al., 2024), and inference-time reasoning approaches that elicit intermediate reasoning through chain-of-thought (CoT) prompting (Wei et al., 2022). Early refinement methods improve code through iterative feedback (Zhang et al., 2023; Zhong et al., 2024b; Chen et al., 2025), but often remain limited in reasoning-path exploration (Wang et al., 2024). Recent plan- and search-based methods, including Monte Carlo Tree Search (MCTS) and structured planning, improve exploration by organizing candidate reasoning paths more explicitly (Li et al., 2024b,a). However, these methods still face efficiency challenges when deeper exploration requires many rollouts. Our work follows the inference-time reasoning line, but focuses on lightweight logits-level control for efficient CoT path search.

LLMs Logits Decoding Logits are unnormalized token scores that reflect model confidence during generation (Zhang et al., 2025a; Wang et al., 2025). Modifying logits provides a lightweight way to control LLM outputs without additional training; temperature scaling is the simplest example for adjusting generation diversity (Wang and Shu, 2025). Recent studies further regulate logits distributions to guide reasoning or improve generation quality (Tang et al., 2024). Contrastive Decoding compares logits from stronger and weaker distributions (Li et al., 2022a; Ding et al., 2024), while Guided Decoding uses external feedback to steer outputs toward task-specific constraints (Agrawal et al., 2023). Logit-induced implicit CoT has also been shown to improve complex reasoning over

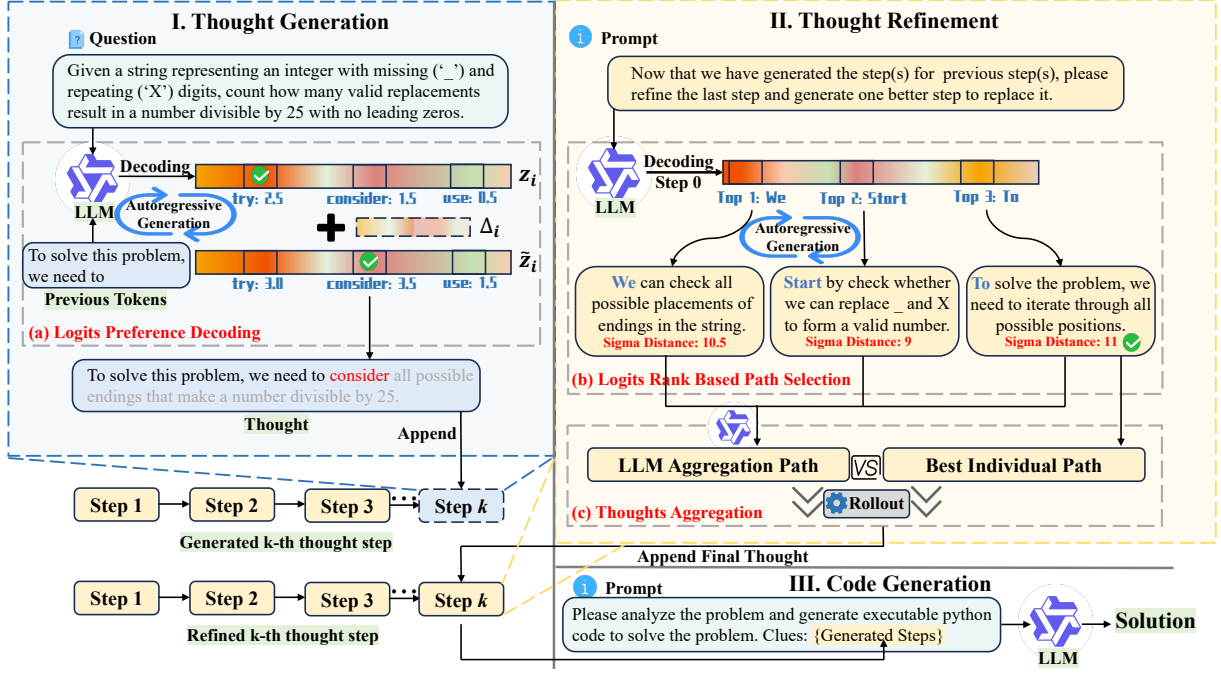


Figure 2: Framework overview. LogitsCoder follows a three-stage reasoning-to-code pipeline: Thought Generation, Thought Refinement, and Code Generation. LPD controls local token expansion during Thought Generation, while LRBPS and Thoughts Aggregation select and combine candidate reasoning trajectories during Thought Refinement.

greedy decoding (Wang and Zhou, 2024a). Different from these methods, LogitsCoder uses logits-level signals not only to calibrate local token expansion, but also to support path-level CoT selection for code generation.

3 Methodology

3.1 Overall Framework

As shown in Figure 2, LogitsCoder follows a three-stage reasoning-to-code pipeline: **Thought Generation**, **Thought Refinement**, and **Code Generation**. These stages describe the execution flow of the framework, while the proposed logits-control modules determine how reasoning is generated and selected within the flow. Specifically, LPD is applied during Thought Generation to calibrate local token expansion with public execution feedback; LRBPS is applied during Thought Refinement to select compact high-confidence reasoning trajectories; and Thoughts Aggregation further combines complementary candidate paths before final code generation.

Thought Generation Given an input problem description X and public tests D_{pub} , LogitsCoder first generates preliminary reasoning thoughts in natural language following the CoT paradigm. During this stage, exploratory reasoning-code trajec-

tries are evaluated on D_{pub} to estimate token-level logit preferences, which are then used by LPD to guide subsequent thought generation. This stage corresponds to local expansion control: it aims to encourage reasoning tokens supported by visible execution feedback and reduce vague or repetitive thoughts. Formally, preference-guided thought generation is written as:

$$p_{\text{LPD}}(T \mid X, \Delta) = \prod_{i=1}^n p_{\text{LPD}}(t_i \mid X, t_{<i}; \Delta),$$

where $T = (t_1, \dots, t_n)$ denotes a generated thought sequence, and Δ denotes the logit preference offsets estimated from public-test feedback.

Thought Refinement After initial thoughts are generated, LogitsCoder refines the reasoning process at the trajectory level. Rather than expanding a full MCTS-style tree, LRBPS explores multiple candidate CoT trajectories and ranks them according to logit-level stability. Thoughts Aggregation then optionally combines complementary candidate paths when a single selected trajectory may miss useful information. This stage corresponds to global path-level selection: it aims to retain a compact and reliable reasoning chain before code generation. We formulate the refined reasoning

chain as:

$$T' = \text{Refine}(X, \{T^{(k)}\}_{k=1}^K),$$

where $\{T^{(k)}\}_{k=1}^K$ denotes candidate thought trajectories generated during exploration, and T' is the refined reasoning chain selected or aggregated for final code generation.

Code Generation Given the refined reasoning chain T' , the code generation module synthesizes executable code conditioned on both the original problem and the refined thoughts:

$$p(C | X, T') = \prod_{i=1}^m p(c_i | X, T', c_{<i}).$$

The refined reasoning chain serves as structured context for final code generation.

3.2 Logits Preference Decoding

LPD is designed to control local token-level expansion during thought generation. In vanilla CoT generation, vague or repetitive tokens can accumulate into long but unproductive reasoning chains, leading to overthinking and token waste. To mitigate this, LPD converts public-test feedback from exploratory reasoning-code trajectories into token-level logit preferences, guiding subsequent thought generation toward execution-supported reasoning patterns rather than a manually fixed word list.

Concretely, LogitsCoder samples M exploratory trajectories and assigns each trajectory an execution reward $r_m \in [0, 1]$ based on its public-test pass rate. LPD then contrasts token occurrences in higher- and lower-reward trajectories. For each tokenizer token v , let $\text{cnt}(v, T_m)$ denote its count in trajectory T_m . We compute:

$$\begin{aligned} n^+(v) &= \sum_{m=1}^M r_m \cdot \text{cnt}(v, T_m), \\ n^-(v) &= \sum_{m=1}^M (1 - r_m) \cdot \text{cnt}(v, T_m). \end{aligned}$$

The raw preference score is computed with a smoothed log-odds ratio:

$$\begin{aligned} s(v) &= \log \frac{n^+(v) + \alpha}{\sum_u n^+(u) + \alpha|V|} \\ &\quad - \log \frac{n^-(v) + \alpha}{\sum_u n^-(u) + \alpha|V|}. \end{aligned}$$

The preference score is normalized and clipped before being injected into decoding:

$$\Delta(v) = \lambda \cdot \text{clip}(\text{Norm}(s(v)), -\tau, \tau),$$

where λ controls the strength of preference injection, τ bounds the maximum logit shift, α is a smoothing constant, and V is the model vocabulary. During subsequent thought generation, LPD adjusts the output logits as:

$$\tilde{z}_t(v) = z_t(v) + \Delta(v).$$

Here $z_t(v)$ is the original logit of token v at decoding step t , and $\tilde{z}_t(v)$ is the calibrated logit. This lightweight adjustment guides local token expansion toward execution-supported reasoning patterns while avoiding unrestricted logit shifts that could destabilize generation.

3.3 Logits Rank Based Path Selection

While LPD improves local token expansion, a single CoT trajectory may still miss important reasoning branches, leading to underthinking. To address this, we introduce **Logits Rank Based Path Selection** (LRBPS), a lightweight path-level selection mechanism that explores multiple plausible trajectories and retains a compact high-confidence path without constructing a full MCTS-style search tree.

Given an input X , the model first computes the logits distribution for the initial reasoning token:

$$z^{(0)} = f_{\text{LM}}(X), \quad z^{(0)} \in \mathbb{R}^{|V|}.$$

Instead of greedily following only the top token, LRBPS selects the top- K initial tokens:

$$\{t^{(k)}\}_{k=1}^K = \text{TopK}(z^{(0)}).$$

Each selected token seeds one candidate CoT trajectory:

$$p(T^{(k)} | X, t^{(k)}) = \prod_{j=2}^n p_{\theta}(t_j | X, t^{(k)}, t_{<j}),$$

where $T^{(k)}$ denotes the k -th candidate reasoning trajectory. This provides lightweight multi-path exploration while avoiding repeated tree expansion and rollout.

To select among candidate trajectories, LRBPS uses sigma distance as a lightweight confidence proxy based on logit-level stability. This choice is inspired by Top- $n\sigma$ (Tang et al., 2024), which shows that the standardized separation between

a high-confidence logit and the background logit distribution can distinguish informative decisions from noisy alternatives. LRBPS adapts this token-level intuition to the path level: a reasoning path is preferred when it contains clearer high-confidence token decisions relative to the logit variation observed along that path. For a trajectory $T^{(k)}$, let $\{s_i^{(k)}\}_{i=1}^n$ denote the logits of its generated tokens. We compute:

$$\bar{s}^{(k)} = \frac{1}{n} \sum_{i=1}^n s_i^{(k)},$$

$$\text{std}^{(k)} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (s_i^{(k)} - \bar{s}^{(k)})^2}.$$

The sigma distance is defined as:

$$D_\sigma(T^{(k)}) = \frac{\max_i s_i^{(k)} - \bar{s}^{(k)}}{\text{std}^{(k)} + \epsilon}.$$

A larger D_σ indicates that the trajectory contains stronger high-confidence token decisions relative to its overall logit variation. We select the trajectory with the highest sigma distance:

$$T^* = \arg \max_k D_\sigma(T^{(k)}).$$

In this way, LRBPS mitigates shallow single-path reasoning by exploring multiple candidate trajectories, while limiting redundant search by retaining only a compact high-confidence path.

3.4 Thoughts Aggregation

LRBPS retains a compact high-confidence trajectory, but a single selected path may discard useful reasoning clues from other candidates. To balance path confidence and information coverage, we introduce Thoughts Aggregation as a lightweight complementary step.

Given candidate trajectories $\{T^{(k)}\}_{k=1}^K$ and their sigma-distance scores $\{S^{(k)}\}_{k=1}^K$, where $S^{(k)} = D_\sigma(T^{(k)})$, we construct two reasoning candidates. The first is an aggregated path, where an LLM summarizes complementary insights across trajectories:

$$T_{\text{agg}} = \text{Aggregator}\left(\{(T^{(k)}, S^{(k)})\}_{k=1}^K\right).$$

The second is the best individual path selected by LRBPS:

$$T_{\text{best}} = \arg \max_k D_\sigma(T^{(k)}).$$

We then validate both candidates by rolling out code and executing it on public tests, and choose the candidate with the higher validation score:

$$T_{\text{final}} = \arg \max_{T \in \{T_{\text{agg}}, T_{\text{best}}\}} \text{Score}_{\text{val}}(T).$$

By integrating diverse reasoning signals with public-test validation, Thoughts Aggregation selects a more reliable reasoning chain for subsequent code generation.

4 Experiment

In this section, a series of experiments are conducted to answer the following research questions (RQs):

RQ1 How does our proposed LogitsCoder perform against the baselines?

RQ2 Does the proposed LogitsCoder perform an efficient and lightweight CoT path search?

RQ3 Does each component of LogitsCoder bring a performance gain?

RQ4 How does LogitsCoder scale with increasing rollout budgets?

RQ5 How does LogitsCoder affect the generated reasoning chains?

4.1 Experimental Setup

4.1.1 Datasets and Metrics

We evaluate LogitsCoder on two widely used code generation benchmarks: APPS (Hendrycks et al., 2021) and CodeContest (Li et al., 2022b).

- **APPS** contains three difficulty levels: introductory, interview, and competition. We evaluate all methods on 100 problems from each difficulty level.
- **CodeContest** categorizes problems into basic and advanced levels according to difficulty tags. We evaluate on 73 basic problems and 43 advanced problems.

We use *pass rate* and *pass@1* as evaluation metrics. *Pass rate* measures the average percentage of private test cases passed by generated programs, while *pass@1* measures the proportion of problems for which the generated program passes all private test cases.

Model		APPS						CodeContest			
		Pass Rate			Pass@1			Pass Rate		Pass@1	
		Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Basic	Advanced	Basic	Advanced
Base	Zero-shot	0.3802	0.3157	0.1950	0.1800	0.1200	0.0700	0.3438	0.2649	0.1918	0.1395
	self-play	0.5146	0.4876	<u>0.3017</u>	0.3100	0.2100	<u>0.1700</u>	0.4356	0.4459	0.2877	<u>0.3023</u>
Reflection-based Models	Reflexion	0.4538	0.4133	0.2450	0.2700	0.2000	0.1200	0.3432	0.2793	0.2055	0.1628
	RAP	0.4182	0.3632	0.2483	0.2400	0.1300	0.0700	0.2954	0.2056	0.1644	0.1163
	LDB	0.4337	0.3761	0.2367	0.2400	0.2700	0.1000	0.4139	0.3087	0.2740	0.1628
	LATS	0.5321	0.4442	0.2386	<u>0.3500</u>	0.2100	0.1300	0.4412	0.3351	0.2603	0.1860
Search-guided Reasoning Models	MCTS	0.5627	0.5667	0.2733	0.3200	0.2300	0.1300	0.4114	<u>0.4590</u>	0.2329	0.2326
	RethinkMCTS	<u>0.5995</u>	<u>0.5888</u>	0.2700	0.3100	<u>0.2900</u>	0.1300	0.4003	0.4238	0.2381	0.2587
Decoding-based Models	Contrastive Decoding	0.5623	0.5463	0.2883	<u>0.3500</u>	0.2500	0.1300	<u>0.4512</u>	0.4286	<u>0.2883</u>	0.2466
	Guided Decoding	0.5615	0.5527	0.3150	0.3200	0.2500	<u>0.1700</u>	0.4360	0.3996	0.2603	0.2326
	LogitsCoder (Ours)	0.6221	0.6130	0.3500	0.3800	0.3400	0.2000	0.4999	0.5040	0.3151	0.3256

Table 1: Major results. The best result(s) are marked in bold, and the second best result(s) are underlined. All methods use the same backbone and rollout budget.

Method	APPS									CodeContest					
	Input (k tokens)			Output (k tokens)			Efficiency (k tokens)			Input (k tokens)			Output (k tokens)		
	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Basic	Adv.	Basic	Adv.	Basic	Adv.
RethinkMCTS	13072	12900	15163	1922	1966	2414	282	286	740	12311	8184	1766	1166	396	248
Base+Decoding+Best	10187	9130	10672	971	129	1369	201	196	430	11958	6068	1291	649	331	174
Base+Decoding+Agg.	12748	11355	12287	924	1037	1420	253	237	422	9700	6113	948	630	261	173
LogitsCoder	12835	12777	12403	1140	207	1733	243	248	453	11561	6068	124	649	250	158

Table 2: Token usage and token-cost proxy for different methods on APPS and CodeContest. Efficiency is measured in weighted thousand tokens per pass-rate percentage point.

4.1.2 Baselines

We compare LogitsCoder with representative baselines from four categories: **self-reflection-based methods**, including self-play (Madaan et al., 2022), Reflexion (Shinn et al., 2024), and RAP (Kagaya et al., 2024); **code debugging methods**, including LDB (Zhong et al., 2024a); **search-guided reasoning methods**, including MCTS (Zhu et al., 2022), RethinkMCTS (Li et al., 2024b), and LATS (Zhou et al., 2023); and **decoding-based methods**, including Contrastive Decoding (Li et al., 2023) and Guided Decoding (Wang and Zhou, 2024b).

4.1.3 Implementation

We use Qwen2.5-14B-Instruct (Yang et al., 2024) as the backbone model for the main experiments. Closed-source models such as GPT-4 (Jaech et al., 2024) are not included because our method requires token-level logits access during decoding. For fair comparison, all baselines and LogitsCoder use the same maximum budget of 20 code execution rollouts. Following prior work (Li et al., 2024b; Chen et al., 2022), we split test cases into public tests for execution feedback and private tests for final evaluation. In LogitsCoder, LPD constructs token preferences only from trajectories evaluated

on public tests, and private tests are never used for preference estimation, path selection, or model feedback. We additionally reproduce the comparison with DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024); the cross-backbone results are provided in Appendix E.

4.2 Overall Performance (RQ1)

In this section, we compare LogitsCoder with baseline methods for code generation, with results shown in Table 1. From these results, we observe that (1) LogitsCoder achieves the best performance across all APPS and CodeContest settings under the same backbone and rollout budget, showing its effectiveness on contest-level code generation tasks. (2) Search-guided methods, such as MCTS and RethinkMCTS, generally outperform reflection-based methods, suggesting that exploring multiple reasoning paths is important for complex code generation. LogitsCoder further improves over these search-guided baselines without relying on full MCTS-style tree expansion, indicating the benefit of lightweight logits-level control. (3) Compared with decoding-based baselines, LogitsCoder combines local token-level calibration with global path-level selection, suggesting that LPD and LRBPS

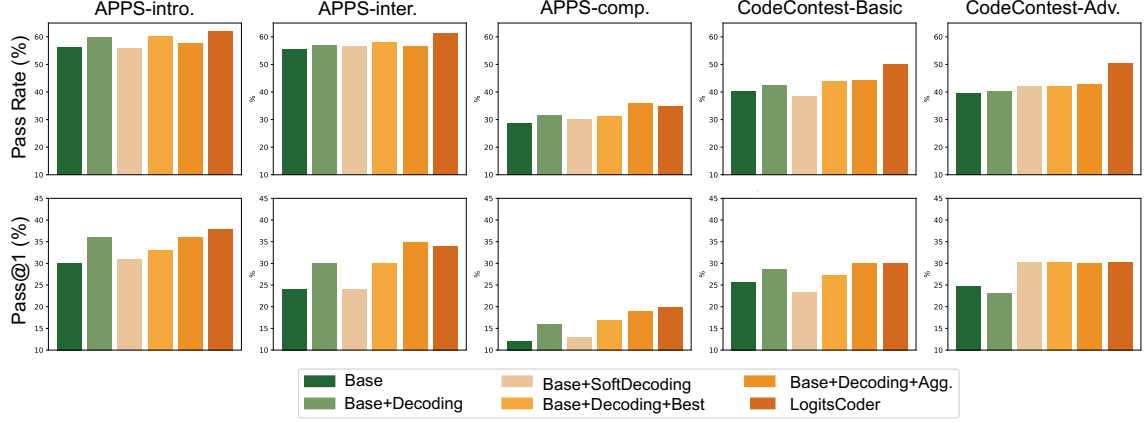


Figure 3: Ablation study: Performance of variants of LogitsCoder on APPS and CodeContest datasets.

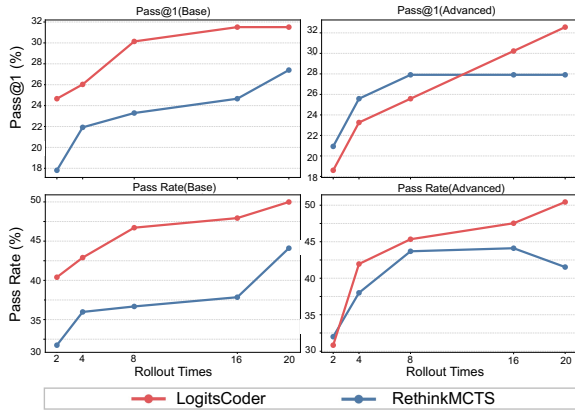


Figure 4: Test-time scaling performance of LogitsCoder and RethinkMCTS with rollout budgets from 2 to 20.

provide complementary control over CoT search and lead to more reliable code generation across different difficulty levels.

4.3 Token Efficiency (RQ2)

To evaluate the cost-effectiveness of CoT search, we compare LogitsCoder with two simplified variants and the search-based baseline RethinkMCTS under the same rollout budget:

- **Base+Decoding+Agg.** applies LPD during thought generation and aggregates multiple reasoning paths through LLM summarization.
- **Base+Decoding+Best** applies LPD and selects the single best path according to sigma distance without summarization.

These variants isolate the effects of path aggregation and best-path selection in the proposed framework.

We report input and output token consumption and use the following token-cost proxy:

$$\text{Efficiency} = \frac{(\text{Input} + 2 \times \text{Output})/1000}{\text{Pass Rate (\%)}}.$$

A lower value indicates fewer weighted thousand tokens required per percentage point of pass rate. Following common inference-cost considerations, output tokens are weighted more heavily because autoregressive decoding is typically more expensive than processing input tokens (Pope et al., 2022; Shazeer, 2019). This metric is intended as a lightweight proxy for token cost rather than a replacement for wall-clock or monetary cost. We set the output-token weight to 2 as a simple approximation of higher decoding cost and include raw input/output token counts for transparency. Additional model-call and wall-clock measurements are reported in Appendix E.

As shown in Table 2, LogitsCoder and its variants generally require fewer weighted tokens than RethinkMCTS while achieving comparable or higher pass rates. This supports our motivation that logits-level control can reduce redundant reasoning expansion and improve the cost-effectiveness of CoT search.

4.4 Ablation Study (RQ3)

To analyze the contribution of the main components in LogitsCoder, we conduct incremental ablation studies. Figure 3 reports Pass@1 and Pass Rate of different variants on APPS and CodeContest.

We start from *Base*, which performs sequential thought generation, thought refinement, and code generation without logits-level control or path-level selection. *Base+Decoding* adds LPD to calibrate local token expansion. *Base+SoftDecoding*

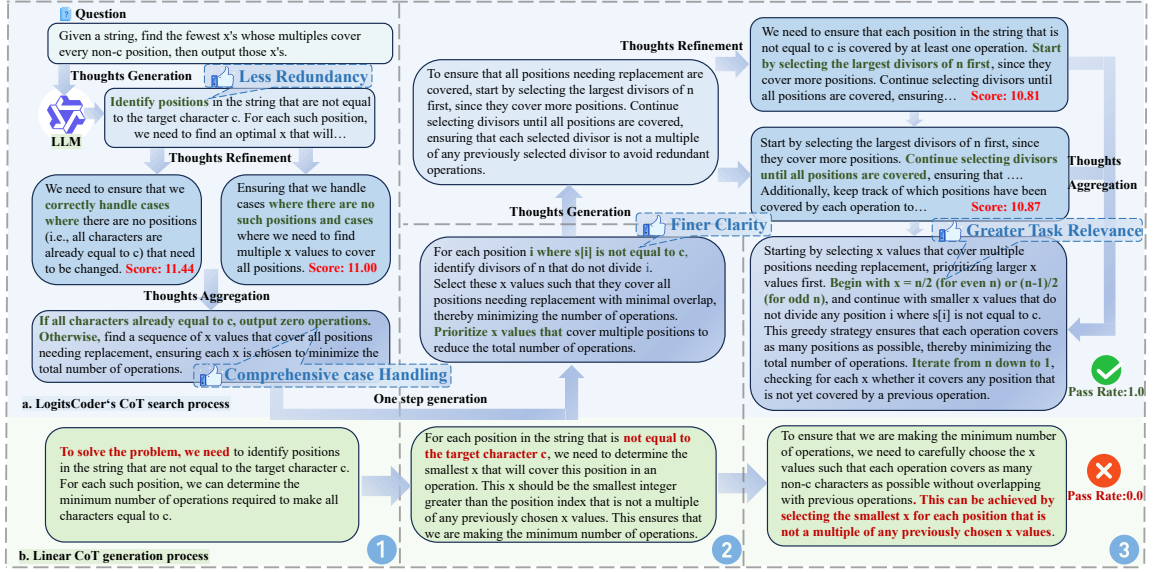


Figure 5: Case study comparing the reasoning chain generated by LogitsCoder and linear CoT on a programming task.

uses a softer variant of LPD, where preference weights are derived from reward-weighted statistics. *Base+Decoding+Best* further applies LRBPS to select the best individual trajectory according to sigma distance, while *Base+Decoding+Agg.* aggregates multiple candidate trajectories through LLM summarization. The full LogitsCoder combines LPD, LRBPS, and Thoughts Aggregation with validation-based selection.

From the results, we observe the following:

- **Effect of LPD.** Compared with the Base model, *Base+Decoding* brings clear performance gains, indicating that token-level calibration is useful for guiding the model toward better reasoning paths.
- **Effect of path-level selection and aggregation.** Both *Base+Decoding+Best* and *Base+Decoding+Agg.* enhance performance by making better use of multiple reasoning trajectories. Best-path selection retains a compact high-confidence path, while aggregation preserves complementary information across paths.
- **Overall effect.** The full LogitsCoder achieves the strongest overall performance among the evaluated variants, suggesting that local expansion control, path-level selection, and aggregation provide complementary benefits for CoT search.

4.5 Test Time Scaling Performance (RQ4)

We conduct experiments on test-time scaling performance to address two questions: (1) whether Log-

itsCoder can continuously benefit from increased rollout budgets? (2) Does LogitsCoder possess TTS capability comparable to or better than RethinkMCTS?

To answer these questions, we vary the number of rollouts from 2 to 20 for both LogitsCoder and RethinkMCTS, and evaluate them on the basic and advanced subsets of CodeContest. From Figure 4, we observe that (1) LogitsCoder improves as the rollout budget increases, reaching Pass Rate scores of 49.99% on basic problems and 50.44% on advanced problems at 20 rollouts. This indicates that LogitsCoder can benefit from additional test-time computation. (2) Compared with RethinkMCTS, LogitsCoder shows stronger performance at larger rollout budgets, suggesting that lightweight trajectory exploration and logits-based path selection can make effective use of additional rollouts without relying on costly MCTS-style tree expansion.

4.6 Case Study (RQ5)

We further conduct a qualitative case study to examine how LogitsCoder affects the generated reasoning chain. Figure 5 compares LogitsCoder with conventional linear CoT on a representative CodeContest problem. The comparison shows the following differences.

Less Redundancy LogitsCoder reduces verbose or repetitive statements commonly observed in vanilla CoT. Generic remarks are replaced by more focused problem-decomposition steps.

More Complete Case Handling LogitsCoder considers special and edge cases more explicitly, such as when all characters are already the target c or when certain positions require no change.

Finer Granularity The reasoning chain is organized into clearer sub-tasks, making intermediate decisions more actionable for final code generation.

Greater Task Relevance The generated thoughts are more directly tied to the final solution, whereas the baseline may include steps that are too abstract or weakly related to executable code.

Overall, the case study illustrates how logits-guided local expansion and path-level selection can reduce redundant reasoning while improving coverage of task-specific solution details.

5 Conclusion

In this paper, we propose LogitsCoder, a lightweight framework for efficient CoT search in code generation. LogitsCoder uses logits-level control to improve both thought generation and thought refinement: LPD calibrates local token expansion with public-test feedback, while LRBPS selects compact high-confidence reasoning paths without costly MCTS-style tree expansion. Experiments on APPS and CodeContest show that LogitsCoder improves both code generation accuracy and token efficiency under the same backbone and rollout budget.

Acknowledgements

The Shanghai Jiao Tong University team is partially supported by National Natural Science Foundation of China 62502310.

Limitations

Despite the substantial performance improvements demonstrated by LogitsCoder, this work has several limitations that warrant future investigation:

Limited Generalizability of Experimental Evaluation Our experiments focus on the competitive benchmarks, which consist primarily of self-contained, algorithmic problems typical of programming competitions, following previous works (Li et al., 2024b; Chen et al., 2025). While these datasets effectively showcase LogitsCoder’s strengths in complex algorithmic reasoning, they do not fully reflect the challenges of real-world

software development—such as managing complex dependencies and interacting with external libraries. More realistic benchmarks present these challenges, remain unexplored in our study.

Limited Backbone and Scale Coverage We evaluate Qwen2.5-14B-Instruct and additionally reproduce our main comparison with DeepSeek-Coder-V2-Lite-Instruct. The consistent trend across these two open-weight model families provides initial cross-backbone evidence, but does not constitute a systematic study across model scales. Moreover, closed-source state-of-the-art models (e.g., GPT-4, Gemini, and Claude) generally do not expose the token-level logits required by LogitsCoder. Evaluating a broader range of open-weight model families and parameter scales remains important future work.

These limitations highlight areas where LogitsCoder can evolve. We believe that addressing them could further enhance the framework’s adaptability and effectiveness across various structured reasoning tasks.

References

- Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2023. Monitor-guided decoding of code lms with static analysis of repository context. *Advances in Neural Information Processing Systems*, 36:32270–32298.
- 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*.
- Jizheng Chen, Kounianhua Du, Xinyi Dai, Weiming Zhang, Xihuai Wang, Yasheng Wang, Ruiming Tang, Weinan Zhang, and Yong Yu. 2025. Debatecoder: Towards collective intelligence of llms via test case driven llm debate for code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12055–12065.
- Liang Ding, Li Shen, Wei Yu, Dacheng Tao, and 1 others. 2024. Usdc: Improving code generation of llms by uncertainty-aware selective contrastive decoding.
- Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, and 1 others. 2023. Codeapex: A bilingual programming evaluation benchmark for large language models. *arXiv preprint arXiv:2309.01940*.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns,

- Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. 2024. [Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents](#). *Preprint*, arXiv:2402.03610.
- Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024a. Code-tree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*.
- Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2024b. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*.
- Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. 2022a. Contrastive decoding: Open-ended text generation as optimization. *arXiv preprint arXiv:2210.15097*.
- Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. 2023. [Contrastive decoding: Open-ended text generation as optimization](#). *Preprint*, arXiv:2210.15097.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022b. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.
- Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.
- Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. [Efficiently scaling transformer inference](#). *Preprint*, arXiv:2211.05102.
- Noam Shazeer. 2019. [Fast transformer decoding: One write-head is all you need](#). *Preprint*, arXiv:1911.02150.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.
- Chenxia Tang, Jianchun Liu, Hongli Xu, and Liusheng Huang. 2024. [Top- \$n\sigma\$: Not all logits are you need](#). *Preprint*, arXiv:2411.07641.
- Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*.
- Haoran Wang and Kai Shu. 2025. Make every token count: A systematic survey on decoding methods for foundation models.
- Xuezhi Wang and Denny Zhou. 2024a. Chain-of-thought reasoning without prompting. *Advances in Neural Information Processing Systems*, 37:66383–66409.
- Xuezhi Wang and Denny Zhou. 2024b. [Chain-of-thought reasoning without prompting](#). *Preprint*, arXiv:2402.10200.
- Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.
- Yue Wang, Qiuzhi Liu, Jiahao Xu, Tian Liang, Xingyu Chen, Zhiwei He, Linfeng Song, Dian Yu, Juntao Li, Zhuosheng Zhang, and 1 others. 2025. Thoughts are all over the place: On the underthinking of o1-like llms. *arXiv preprint arXiv:2501.18585*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits its reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.
- Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772.
- Jinghan Zhang, Xiting Wang, Fengran Mo, Yeyang Zhou, Wanfu Gao, and Kunpeng Liu. 2025a. Entropy-based exploration conduction for multi-step reasoning. *arXiv preprint arXiv:2503.15848*.
- Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.
- Weiming Zhang, Qingyao Li, Xinyi Dai, Jizheng Chen, Kounianhua Du, Weinan Zhang, Weiwen Liu, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025b. Nl-debugging: Exploiting natural language as an intermediate representation for code debugging. *arXiv preprint arXiv:2505.15356*.
- Li Zhong, Zilong Wang, and Jingbo Shang. 2024a. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.
- Lily Zhong, Zilong Wang, and Jingbo Shang. 2024b. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.
- Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.
- Xinyu Zhu, Junjie Wang, Lin Zhang, Yuxiang Zhang, Ruyi Gan, Jiaying Zhang, and Yujiu Yang. 2022. Solving math word problem via cooperative reasoning induced language models. *arXiv preprint arXiv:2210.16257*.
- Yuqi Zhu, Ge Li, Xue Jiang, Jia Li, Hong Mei, Zhi Jin, Yihong Dong, and Qibin Zheng. 2025. Uncert-cot: Uncertainty-aware chain-of-thought for code generation with large language model. In *International Conference on Intelligent Computing*, pages 426–437. Springer.

Appendix

A Algorithm

Algorithm 1 summarizes the inference procedure of LogitsCoder. The LPD preparation stage only constructs logits preferences from visible public tests, while the main procedure performs iterative thought generation and refinement for each problem. Here M denotes the number of exploratory trajectories for LPD, K denotes the number of candidate continuations at each step, and L denotes the maximum number of thought steps. Private tests are excluded from all steps and used only for final reporting.

Algorithm 1 Inference Procedure of LogitsCoder

Require: Problem X ; visible public tests D_{pub} ; model f_θ ; budgets M, K, L

Ensure: Final generated program C^*

```
1: Comment:  $D_{\text{pub}}$  is visible pre-test feedback; private tests are never used below.
2: LPD preparation
3: Sample exploratory trajectories  $\mathcal{Y} = \{(T_m, C_m)\}_{m=1}^M$ 
4: for each  $(T_m, C_m) \in \mathcal{Y}$  do
5:   Execute  $C_m$  on  $D_{\text{pub}}$  and obtain public reward  $r_m$ 
6: end for
7: Estimate token-level offsets  $\Delta$  from  $\{(T_m, r_m)\}_{m=1}^M$ 
8: Iterative thought generation and refinement
9: Initialize thought chain  $T \leftarrow \emptyset$ 
10: for  $\ell = 1$  to  $L$  do
11:   Generate  $K$  candidate next-step continuations  $\{T_\ell^{(k)}\}_{k=1}^K$  from  $(X, T)$ 
12:   using calibrated logits  $\tilde{z}_t = z_t + \Delta$ 
13:   for  $k = 1$  to  $K$  do
14:     Record generated-token logits of  $T_\ell^{(k)}$ 
15:     Compute sigma distance  $D_\sigma(T_\ell^{(k)})$ 
16:   end for
17:   Select  $T_{\ell, \text{best}} = \arg \max_k D_\sigma(T_\ell^{(k)})$ 
18:   Aggregate ranked candidates into  $T_{\ell, \text{agg}}$ 
19:   Generate code from  $T \oplus T_{\ell, \text{best}}$  and  $T \oplus T_{\ell, \text{agg}}$ 
20:   Execute both programs on  $D_{\text{pub}}$ 
21:   Set  $T_\ell^*$  to the candidate with higher public-test score
22:   Update thought chain  $T \leftarrow T \oplus T_\ell^*$ 
23:   if the thought chain is complete then
24:     break
25:   end if
26: end for
27: Code generation
28: Generate final code  $C^*$  conditioned on  $X$  and refined thoughts  $T$ 
29: return  $C^*$ 
```

B Dataset Details

We assess the performance of our LogitsCoder framework and all the baselines using two commonly used code generation datasets: APPS (Hendrycks et al., 2021) and CodeContest (Li et al., 2022b). The APPS dataset contains 5000 programming challenges split into three

difficulty categories—introductory, interview, and competition—with 5000 problems for training and another 5000 for testing. The CodeContest dataset, sourced from the online competitive programming platform, includes problems of various difficulty levels, categorized by "ratings". For evaluation, we focus on problems rated below and above 15, dividing them into basic and advanced sets.

For APPS dataset, we select 100 problems from each difficulty level to ensure a balanced evaluation, and for CodeContest dataset, we have 73 problems for basic and 43 problems for advanced set. Each dataset uses the same set of public test cases for algorithm optimization and private test cases for performance evaluation.

C Baseline Details

To give a thorough insight into our experimental setup, we provide an in-depth description of each baseline code generation method used for comparison with LogitsCoder. All baselines utilize code execution feedback to progressively refine the code.

Self-play (Madaan et al., 2022) This method improves the output of LLMs by using an iterative feedback and refinement process. After an initial output is generated by the model, the same model provides feedback on its own output. This feedback is then used to refine the output, repeating the process for multiple iterations until a stopping condition is met. This method does not require additional training or external supervision, instead relying on a single LLM to generate, critique, and refine its own outputs.

Reflexion (Shinn et al., 2024) Reflexion enables language agents to improve through code execution traces. After each trial, the agent generates natural language feedback about its own mistakes and stores it in memory. This feedback is then used to guide future decisions, acting as a semantic signal for learning.

RAP (Kagaya et al., 2024) RAP enhances language agents by combining retrieval-augmented memory with task planning. The agent retrieves relevant past trajectories through code execution from an external memory based on the current programming problem, then conditions its planning and actions on this retrieved experience.

LDB (Zhong et al., 2024a) LDB breaks down programs into fundamental code blocks and con-

ducts execution-level analysis on each one. Its reasoning approach is detailed and structurally informed, yet remains confined to the scope of individual code blocks.

LATS (Zhou et al., 2023) LATS unifies reasoning, acting, and planning in language models by integrating Monte Carlo Tree Search (MCTS) with large language models (LLMs). In LATS, the LLM acts as both an agent and a value function. LATS samples multiple candidate solutions, evaluates them using test feedback (e.g., assert statements), and selects the best-performing one via tree search.

MCTS (Zhu et al., 2022) Monte Carlo Tree Search (MCTS) treats code generation as a thought-level sequential decision-making problem, where each thought step corresponds to an action. It incrementally builds a tree of partial CoT sequences, simulating code completions to evaluate possible next thought steps. Using techniques like UCT to balance exploration and exploitation, MCTS selects promising paths and updates node values based on code quality metrics or execution results.

RethinkMCTS (Li et al., 2024b) RethinkMCTS introduces a "rethink" operation, where erroneous thoughts are corrected based on this feedback, improving search quality and efficiency, to the MCTS process.

RethinkMCTS enhances CoT reasoning via Monte Carlo Tree Search with path-level rethink operations, but incurs high token and computation costs; in contrast, LogitsCoder achieves efficient CoT exploration through logits-based decoding, ranking, and aggregation without full tree traversal.

Contrastive Decoding (Li et al., 2023) Contrastive Decoding (CD) generates text by selecting tokens that maximize the difference between the log-likelihoods derived from two logits, denoted as z^+ and z^- , while simultaneously enforcing a plausibility constraint to maintain naturalness and fluency. Specifically, we define z^+ as the statistical distribution of tokens obtained from problems with satisfactory performance (accuracy > 0.5), and z^- as the corresponding distribution from problems with unsatisfactory performance (accuracy ≤ 0.5).

Guided Decoding (Wang and Zhou, 2024b) Guided decoding steers code generation by incorporating guidance function into the token selection process during inference to adjust the model’s logits distributions. In this paper, for a fair comparison,

we use sigma distance as the guidance function.

D Implementation Details

We use Qwen2.5-14B-Instruct as the backbone model for the main experiments, since LogitsCoder requires access to token-level logits during decoding. The additional backbone and efficiency analyses in Appendix E use DeepSeek-Coder-V2-Lite-Instruct under the same protocol. Following the setting in the main experiments, all methods use the same maximum budget of 20 public-test execution rollouts. For LPD, we run two exploration turns on the public split of each dataset to collect public-test-scored reasoning-code trajectories and estimate token-level preference offsets. These offsets are constructed only from visible public-test feedback. Private tests are never used for preference estimation, path selection, aggregation, refinement, or any other inference-time decision; they are used only for final evaluation.

During iterative thought refinement, LogitsCoder maintains a partial reasoning chain and updates it step by step. At each step, candidate continuations are generated with LPD-calibrated logits, ranked by LRBPS using sigma distance, and optionally summarized by Thoughts Aggregation. The best individual path and the aggregated path are both validated through public-test execution, and the path with the higher public-test score is retained. We set the maximum reasoning-path length to $L = 10$. Unless otherwise specified, we use temperature 0.7, top- $p = 0.95$, smoothing coefficient $\alpha = 1.0$, logit-offset strength $\lambda = 0.5$, clipping threshold $\tau = 2.0$, and numerical stability constant $\epsilon = 10^{-6}$.

E Additional Experimental Analysis

We provide a concise analysis of the path selector, backbone generalization, and end-to-end efficiency.

E.1 Ablation of Path-Selection Metrics

Following Top- $n\sigma$ (Tang et al., 2024), sigma distance measures the standardized separation of a high-confidence logit from the background variation. LRBPS applies this intuition at the path level. We fix the DeepSeek-Coder-V2-Lite-Instruct backbone, prompts, candidate pool, and rollout budget, and replace only the selector with random selection or the Probability Differential score from UnCert-CoT (Zhu et al., 2025).

Path selector	APPS						CodeContest			
	Pass Rate			Pass@1			Pass Rate		Pass@1	
	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Basic	Advanced	Basic	Advanced
LogitsCoder-UnCert-CoT	0.6219	0.6286	0.3217	0.41	0.33	0.16	0.4339	0.4207	0.2877	0.2326
LogitsCoder-Random	0.3050	0.6033	0.3000	0.15	0.31	0.15	0.4289	0.4283	0.2740	0.2791
LogitsCoder (Sigma distance)	0.6453	0.6234	0.3150	0.42	0.34	0.18	0.4522	0.4410	0.3151	0.2791

Table 3: Controlled ablation of path-selection metrics with DeepSeek-Coder-V2-Lite-Instruct. Values are proportions; bold and underlined denote the best and second-best results.

Method	APPS						CodeContest			
	Pass Rate			Pass@1			Pass Rate		Pass@1	
	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Basic	Advanced	Basic	Advanced
LDB	0.5264	0.4442	0.2117	0.33	0.21	0.08	0.3682	0.3304	0.2192	0.1389
Reflexion	0.5696	0.4559	0.2317	0.33	0.23	0.08	0.3101	0.3417	0.1918	0.1860
MCTS	0.6118	0.6110	0.3350	0.41	0.33	0.12	0.4386	0.4013	0.3014	0.2326
RAP	0.4436	0.4343	0.2217	0.27	0.20	0.06	0.3187	0.2526	0.1918	0.1395
LATS	0.5766	0.5029	0.1483	0.39	0.24	0.12	0.4137	0.3215	0.2900	0.1351
RethinkMCTS	0.6025	0.6137	0.3366	0.40	0.32	0.16	0.4519	0.4013	0.3151	0.2326
LogitsCoder (Ours)	0.6380	0.6234	0.3150	0.42	0.34	0.18	0.4522	0.4410	0.3151	0.2791

Table 4: Results with DeepSeek-Coder-V2-Lite-Instruct. Values are proportions; bold and underlined denote the best and second-best results.

Sigma distance improves Pass@1 over UnCert-CoT in all five settings (1.0–4.65 points), wins eight of ten comparisons against UnCert-CoT, and wins nine against random selection with one tie.

E.2 Performance on Different Backbone Models

We repeat the main comparison with DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024) under the same datasets, prompts, rollout budget, and evaluation protocol.

LogitsCoder obtains the best Pass@1 on four of five splits and ties RethinkMCTS on CodeContest Basic. It has the best Pass Rate on four splits; APPS Competition is the exception.

E.3 More Details about Efficiency

We profile RethinkMCTS and LogitsCoder with DeepSeek-Coder-V2-Lite-Instruct using the same hardware/software configuration and a maximum of 20 code-execution rollouts per problem. The reported LogitsCoder totals include one-time LPD preparation and Thoughts Aggregation. Efficiency Ratio is $(\text{input} + 2 \times \text{output})/1000$ divided by pass rate in percentage points.

LogitsCoder reduces model calls by 4.4%–44.4% and runtime by 49.3%–65.1%, corresponding to $1.97 \times$ – $2.87 \times$ speedups over RethinkMCTS.

F Details of Logits Preference Decoding

LPD is a local token-level calibration mechanism for thought generation. Its goal is to reduce vague, repetitive, or low-value reasoning expansions by converting visible public-test feedback into logit offsets. Importantly, LPD is not a fixed decoding setting or a manually fixed global word list. Instead, it is adaptive to the evaluation environment: for each dataset setting, LogitsCoder first performs lightweight exploration on the corresponding public split, collects public-test-scored reasoning-code trajectories, and estimates tokenizer-level preferences from these trajectories. Private tests are never used for preference estimation.

Exploration and scoring Let $\mathcal{Y} = \{(T_j, C_j)\}_{j=1}^{|\mathcal{Y}|}$ denote the collected exploratory trajectories, where T_j is the reasoning chain and C_j is the generated code. Each trajectory is scored by executing C_j on public tests:

$$r_j = \frac{1}{|D_{\text{pub}}|} \sum_{d \in D_{\text{pub}}} \mathbb{I}[C_j(d) \text{ passes}],$$

where $r_j \in [0, 1]$ is used only as visible execution feedback.

Token-level preference estimation LPD estimates preferences at the tokenizer-token level. For each token v , let $\text{cnt}(v, T_j)$ denote the number of

Method	Setting	Input Tokens	Output Tokens	Efficiency Ratio	Model Calls	Runtime (s)
RethinkMCTS	APPS Intro.	11,097,423	1,223,521	229.10	6,161	47,235.56
LogitsCoder	APPS Intro.	10,896,419	896,597	198.90	3,659	18,432.38
RethinkMCTS	APPS Inter.	10,107,886	1,352,870	208.79	6,237	47,591.38
LogitsCoder	APPS Inter.	9,910,477	1,112,964	194.68	3,654	22,946.08
RethinkMCTS	APPS Comp.	13,004,924	1,834,424	495.36	6,295	61,753.45
LogitsCoder	APPS Comp.	11,066,943	1,198,051	427.40	3,498	21,530.82
RethinkMCTS	CodeContest Basic	10,448,963	881,219	270.22	2,853	35,698.96
LogitsCoder	CodeContest Basic	9,577,379	871,335	250.33	2,727	18,111.65
RethinkMCTS	CodeContest Advanced	6,299,614	759,961	194.86	2,076	27,485.40
LogitsCoder	CodeContest Advanced	6,345,355	602,760	171.22	1,865	12,579.86

Table 5: End-to-end efficiency comparison with DeepSeek-Coder-V2-Lite-Instruct. LogitsCoder includes one-time LPD preparation and Thoughts Aggregation.

times v appears in reasoning chain T_j . We compute reward-weighted and failure-weighted counts:

$$n^+(v) = \sum_{j=1}^{|\mathcal{D}|} r_j \cdot \text{cnt}(v, T_j),$$

$$n^-(v) = \sum_{j=1}^{|\mathcal{D}|} (1 - r_j) \cdot \text{cnt}(v, T_j).$$

The raw preference score is computed with a smoothed log-odds ratio:

$$s(v) = \log \frac{n^+(v) + \alpha}{\sum_u n^+(u) + \alpha|V|} - \log \frac{n^-(v) + \alpha}{\sum_u n^-(u) + \alpha|V|},$$

where α is the smoothing coefficient and V is the model vocabulary. We normalize $s(v)$ over observed tokens and clip the result:

$$\Delta(v) = \lambda \cdot \text{clip}(\text{Norm}(s(v)), -\tau, \tau).$$

Tokens unseen in the exploration stage receive $\Delta(v) = 0$, preventing LPD from imposing arbitrary preferences beyond the collected public feedback.

Preference-guided decoding During thought generation, LPD adjusts the model logits by:

$$\tilde{z}_t(v) = z_t(v) + \Delta(v),$$

where $z_t(v)$ is the original logit of token v at decoding step t and $\tilde{z}_t(v)$ is the calibrated logit. The strength and range of this adjustment are controlled by λ and τ . This calibration encourages tokens associated with higher public-test rewards while suppressing tokens associated with lower-reward exploratory trajectories.

G Prompts

We present prompts that represent the entire framework’s operation process, including steps for thought generation, thought refinement, thoughts aggregation and code generation.

G.1 Prompt for Thought Generation

You will be given a problem description. Please analyze the problem and generate step-by-step clues of how to solve the problem with executable python code. The problem:
{Problem Description}
An example of step-by-step clues for finding the longest palindrome substring is displayed below:
{Example thought step}
You can refer to the above example and generate step-by-step clues for the given problem.
Now, please first generate the first clue for step 1. (If no step has been generated.)
Now that we have generated the clue(s) for previous step(s). Please follow the clues and generate one clue for the next Step . (If previous step have been generated.)

G.2 Prompt for Thought Refinement

Now that we have generated the clue(s) for previous step(s), please refine the last clue of Step and generate one better clue to replace it. Current steps:
{Current Thought Steps}
Here is the current code.
{Current Code Implementation}
Here is the current execution feedback: {Code Execution Feedback}
Please wrap your response into a JSON object that contains keys 'Clue of Step '
An example is given below. Please format your json output in a ****LIST**** form.
{Example thought step}

G.3 Prompt for LLM Thoughts Aggregation

Now that we have generated the clue(s) for previous step(s), we have some candidate clues along with their confidence score to improve clue of Step to make it more concise and detailed. Please give me one better clue that summarizes the candidate clues and contains detailed instruction on solving the problem.
Here are the candidate clues: {Candidate Step with Sigma Distance Score}
Please wrap your response into a JSON object that contains keys 'Clue of Step '
An example is given below. Please format your json output in a ****LIST**** form.
{Example thought step}

G.4 Prompt for Code Generation

You are a professional Python engineer. You will be given a problem description.
Please analyze the problem and generate executable python code to solve the problem.
The problem:
{Problem Description}
——Clues——
{Current Thought Steps}
——Instruction——
Now we have the clues above. Please refer to the clues and solve the problem. Please only output the code ONLY, no example usage or other explanation.