DAJ: Data-Reweighted LLM Judge for Test-Time Scaling in Code Generation

Peijia Qin [1]   Ruiyi Zhang [1]   Qi Cao [1]   Pengtao Xie [1]

Abstract

Test-time scaling for code generation commonly relies on *Best-of-N* selection, in which multiple candidate solutions are sampled from a base model, and the best one is selected by an LLM judge. However, training reliable LLM judges is challenging due to severe distribution shifts, including imbalances between easy and hard problems, mismatches between training tasks and evaluation benchmarks, and trajectory mismatch arising from training data generated by cheaper models whose behavior differs from that of inference-time models. We propose DAJ, a reasoning-based LLM judge trained with verifiable rewards under a bi-level data-reweighted learning framework. The proposed framework learns data-importance weights (either domain-level or instance-level) to optimize generalization performance on a held-out meta set aligned with target benchmarks. To the best of our knowledge, this is the first application of data reweighting to LLM-as-a-Judge training for test-time scaling. Our approach automatically emphasizes hard problems, in-distribution samples, and trajectory-aligned data, without relying on hand-crafted heuristics. Empirically, DAJ achieves state-of-the-art performance on Live-CodeBench and BigCodeBench, outperforming strong test-time scaling baselines as well as leading proprietary models.

1. Introduction

Test-time scaling (TTS) refers to the use of additional inference-time computation to improve model outputs. It has recently emerged as an effective approach for enhancing large language model (LLM) performance on coding tasks (Zhang et al., 2025a; Snell et al., 2025; Jaech et al., 2024; Guo et al., 2025). Among existing TTS approaches, selection-based methods form an important class that gener-

ates multiple candidate outputs at test time and selects the best-performing sample, thereby improving prediction quality without modifying model parameters. A representative selection-based strategy is Best-of-N (Irvine et al., 2023), which selects among multiple sampled candidates based on a scalar score produced by a separate judge. Judge models are therefore a core component of selection-based TTS methods, and are typically implemented using listwise or pairwise comparisons to evaluate generated responses and assign scores. More recently, LLM-as-a-Judge approaches have leveraged large language models themselves as judges to evaluate candidate solutions (Zheng et al., 2023; Wu et al., 2024). By leveraging LLMs' reasoning capabilities, these methods can yield more interpretable evaluations and have demonstrated strong promise for selection-based test-time scaling.

Training LLM judges for code generation is challenging due to multiple forms of mismatch between training and target distributions. **First**, an easy–hard imbalance is common in training data, where easy problems are overrepresented relative to hard ones, leading to degraded judge performance on challenging test-time instances. **Second**, task distribution mismatch arises when the tasks used for training differ from those encountered during evaluation, limiting the ability of judges to generalize to unseen or differently distributed benchmarks. **Third**, trajectory mismatch occurs because training trajectories are often generated by cheaper or weaker models to reduce cost, whereas inference-time candidate solutions typically originate from stronger models, resulting in systematic misalignment between training and deployment settings. These challenges highlight the need for a principled, distribution-shift-aware training methodology that explicitly accounts for such discrepancies to achieve strong generalization to test-time tasks.

We propose DAJ, a **Da**ta-Reweighted LLM **J**udge, to address the challenges of training LLM judges for code generation through principled data reweighting (Figure 2). In DAJ, each training sample is assigned an importance weight with clear task-specific semantics that captures three complementary factors. **First**, instance difficulty is reweighted, where hard problems are upweighted while easy ones are downweighted. **Second**, task similarity is reweighted, where samples whose tasks are more similar to test-time scenarios are upweighted while those that are more different are down-

[1]University of California, San Diego. Correspondence to: Pengtao Xie <p1xie@ucsd.edu>.
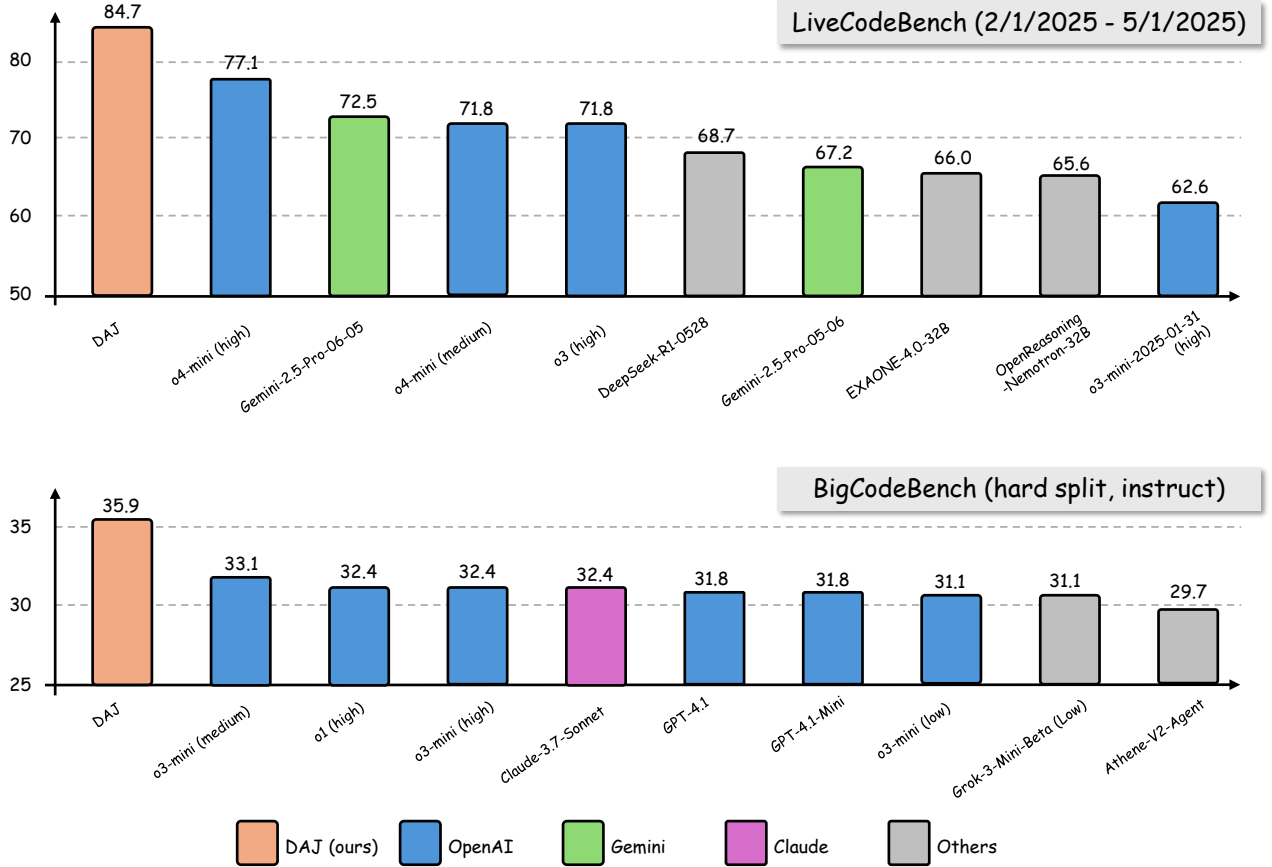
Figure 1. DAJ achieves first place on both LiveCodeBench and BigCodeBench, outperforming other top models. Results are taken from the official leaderboard.

weighted. **Third**, trajectory alignment is reweighted, where samples whose solution candidates resemble inference-time candidates generated by stronger models are upweighted while those whose solution candidates rarely appear at test time are downweighted. DAJ is trained under a bi-level optimization framework that automatically learns these data weights. The upper-level objective optimizes generalization performance on a held-out meta dataset that more closely matches the test-time distribution, while the lower-level objective trains the judge on reweighted training data. This design enables adaptability to distribution shifts without relying on hand-crafted heuristics. For the judge architecture, we adopt a reasoning-based LLM-as-a-Judge paradigm (Wang et al., 2024; Kim et al., 2024; Li et al., 2024) that is particularly well suited for code evaluation (Section 3). The judge is trained using preference optimization or reinforcement learning objectives, where rewards and preference pairs are generated in a verifiable manner without human annotations: correct selections are assigned higher scores than incorrect selections or format errors. The overall training and inference pipelines are illustrated in

Figure 3.

Empirically, on LiveCodeBench (Jain et al., 2025) and BigCodeBench (Zhuo et al., 2024), DAJ achieves state-of-the-art performance (Figure 1), outperforming learning zero-shot methods as well as strong test-time scaling baselines (KRAFTON AI & SKT AI, 2025; He et al., 2024; Liu et al., 2025b). Our main contributions are:

- We introduce a novel bi-level optimization-based data-reweighted training framework for LLM-as-a-Judge, which improves Best-of-N selection in test-time scaling by automatically emphasizing important training examples across three complementary aspects.

- We conduct comprehensive experimental studies on data reweighting strategies and training paradigms, including preference optimization and reinforcement learning, evaluated across a diverse set of base policy models.

- We achieve state-of-the-art performance on LiveCodeBench and BigCodeBench, outperforming lead-
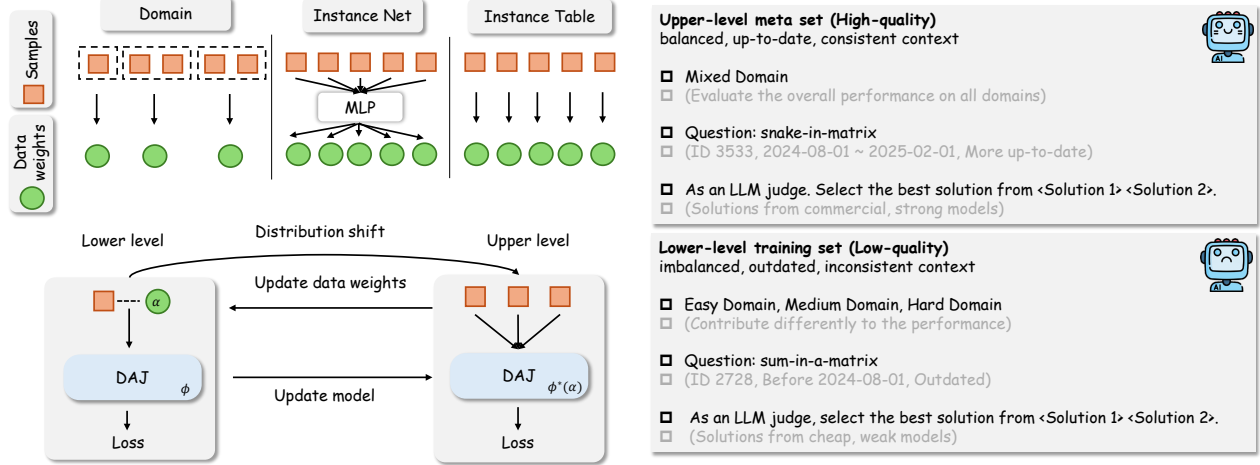
Figure 2. Bi-level optimization-based data-reweighted training. **Top left:** Three reweighting designs. Domain reweighting assigns weights to sample groups (e.g., different coding domains); the instance table assigns explicit learnable weights to each sample, while the instance net parameterizes these weights via a lightweight MLP. **Bottom left:** The bi-level optimization framework. Lower-level optimization updates the judge's parameters using weighted training data, whereas upper-level optimization evaluates the judge on a held-out meta dataset and updates data weights to maximize generalization. **Right:** A detailed comparison between the lower-level low-quality dataset and the upper-level high-quality dataset.

ing proprietary models and existing test-time scaling methods.

2. Related Work

LLM-as-a-Judge. Large language models are increasingly used as judges to evaluate open-ended generation, motivated by the scalability and flexibility of natural-language evaluation compared with traditional reference-based metrics. Early work on LLM-as-a-Judge demonstrated strong agreement between LLM judgments and human preferences in conversational settings (Zheng et al., 2023), establishing the feasibility of using LLMs as evaluators. Subsequent studies have shown that LLM judges are particularly effective for selection and reranking, where the goal is to select the best solution among multiple candidates. A notable application is large-scale sampling combined with filtering or selection in code generation, as demonstrated by AlphaCode (Li et al., 2022). Beyond outcome-level selection, step-level judging has been explored through process reward models that score intermediate reasoning steps, enabling reward-guided decoding and search procedures such as PRM-guided beam search (Lightman et al., 2023). More recently, unifying benchmarks such as JETTS have been proposed to systematically integrate reranking, step-level search, and critique-based refinement within a unified LLM-as-a-Judge framework (Zhou et al., 2025).

Test-time scaling. Test-time scaling (TTS) dynamically allocates inference-time computation based on input complexity, and can be broadly categorized into sequential and parallel approaches. Sequential methods extend a model's reasoning trace from chain-of-thought prompting (Wei et al., 2022; Nye et al., 2021; Kojima et al., 2022) to adaptive thinking horizons (Guo et al., 2025; Jaech et al., 2024) and self-reflection (Madaan et al., 2023; Saunders et al., 2022). Parallel test-time scaling generates multiple candidate solutions and aggregates them to improve performance (Brown et al., 2024; Snell et al., 2025; Zhang et al., 2025b). Representative strategies include self-consistency voting (Wang et al., 2022) and structured search over intermediate states (e.g., ToT (Yao et al., 2023), GoT (Besta et al., 2024)). In these settings, judges play a central role: Outcome Reward Models (ORMs) score complete responses (Cobbe et al., 2021; Liu et al., 2025a; Ouyang et al., 2022), while Process Reward Models (PRMs) provide step-level rewards to guide reasoning (Lightman et al., 2023; Wang et al., 2023; Zhang et al., 2025b; Luo et al., 2024; Dai et al., 2024; Zhang et al., 2026). Recent judge models further incorporate explicit reasoning before making judgments, improving interpretability and reliability (Wang et al., 2024; Kim et al., 2024; Li et al., 2024). Notable recent instantiations include Skywork-o1 Open PRM (He et al., 2024), which extends process reward modeling to code and mathematical reasoning, and DeepSeek GRM (Liu et al., 2025b), a generative reward model that applies inference-time scaling to reward modeling itself. Orthogonally, offline reinforcement learning approaches such as Offline GRPO (KRAFTON AI & SKT AI, 2025) improve reasoning quality at training time by learning from pre-collected trajectories, offering a complementary axis to test-time compute scaling.
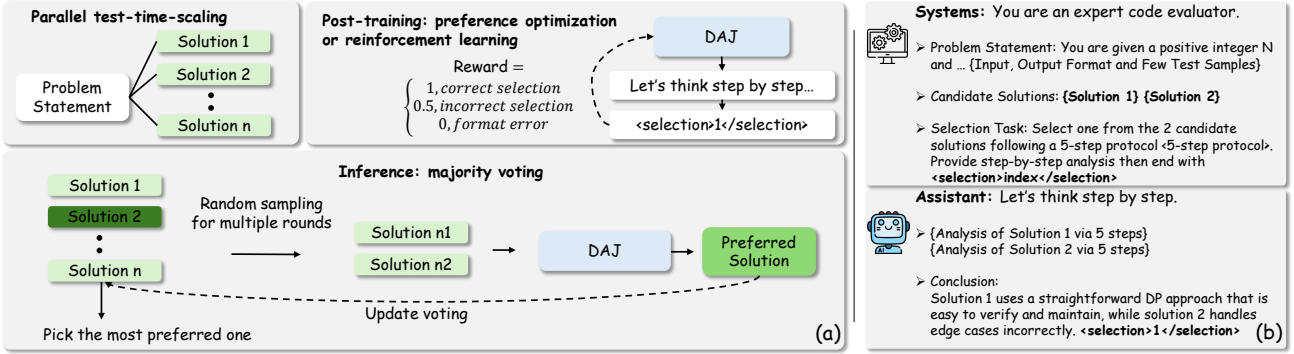
Figure 3. Overview of DAJ training and inference. **(a)** *Top left:* Given a coding problem, we sample n candidate solutions from the policy model for parallel test-time scaling. *Top right:* During training, DAJ performs step-by-step reasoning ("Let's think step by step...") before outputting a selection. The model receives a verifiable reward (Equation (1)), enabling preference optimization without human-annotated reasoning traces. *Bottom:* At inference time, multi-round pairwise voting selects the final output (Section 3). **(b)** A simplified example of inference process, where the judge is asked to select the preferred solution from two candidates, based on a 5-step reasoning pattern (Appendix D).

Data reweighting. Data reweighting aims to adjust the influence of heterogeneous training data to improve robust generalization. Early and foundational methods focused on adaptively learning explicit weighting functions to mitigate overfitting caused by biased data distributions, corrupted labels, and class imbalance (Shu et al., 2019). More recent work has explored data reweighting at scale in the context of large-scale pretraining and domain adaptation. DoReMi assigns domain-level weights through proxy-based optimization under distributionally robust objectives (Xie et al., 2023). DOGE further introduces a first-order bi-level optimization framework that updates mixture weights by aligning gradients between source and target domains (Fan et al., 2024). Complementarily, Data Mixing Laws provide analytical scaling laws that predict downstream performance under different data mixture strategies, offering theoretical guidance for dataset composition (Ye et al., 2025a). Beyond domain-level adaptation, data reweighting has also been applied to reasoning-centric training paradigms. In particular, recent work adapts reweighting techniques to process supervision, enabling better balance across reasoning trajectories of varying quality (Cao et al., 2025). From a theoretical perspective, the convergence properties of related gradient-based bi-level optimization algorithms have been studied extensively (Pedregosa, 2016; Rajeswaran et al., 2019), providing formal support for the stability and soundness of such training procedures, including this work.

3. Preliminary: LLM-as-a-Judge for Code Evaluation

Reasoning-based judging. Unlike conventional reward models that directly output scalar scores, DAJ belongs to the LLM-as-a-Judge paradigm, which generates explicit reasoning traces before producing a final judgment on pair-wise/listwise comparison. While CoT prompting has been explored for LLM judges in general evaluation settings (Liu et al., 2025b; Ye et al., 2025b), our work extends this paradigm to code evaluation. This formulation is particularly well aligned with program verification, as assessing code quality naturally involves structured reasoning such as time and space complexity analysis, correctness on edge cases, and verification of algorithmic soundness.

As illustrated in Figure 3 (b), the judging task is formulated as a chat completion problem. Given a query and the corresponding candidate responses, DAJ autoregressively generates a thinking process followed by a final judgment that identifies the preferred candidate. Although DAJ can accept two or more candidate responses as input, we observe low sensitivity to the number of candidates in practice. We therefore adopt pairwise comparison to reserve sufficient output length for detailed reasoning. When more than two candidates are available, we repeat the following for R rounds: two candidates are sampled uniformly at random (with replacement across rounds), the judge compares them, and the preferred candidate receives one vote. After all rounds, the candidate with the highest cumulative vote count is selected as the final output, with ties broken at random. The full prompt template used for reasoning-based judging is provided in Appendix D.

Verifiable rewards. We consider a dataset $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N}$, where X_i denotes a problem statement paired with candidate solutions, and Y_i represents the judge's selection along with a generated reasoning trajectory. Because each candidate can be executed against test cases, selection correctness is automatically verifiable without hu-

man annotations. We define the reward function as:

$$\mathcal{R}(X_i, Y_i) = \begin{cases} 1, & \text{DAJ selects correctly} \\ 0.5, & \text{DAJ selects incorrectly} \\ 0, & \text{format error} \end{cases} \quad (1)$$

The reward $\mathcal{R}$ evaluates selection correctness rather than the quality of the candidate solutions themselves. This formulation follows the reinforcement learning with verifiable rewards (RLVR) paradigm (Guo et al., 2025), in which rule-based rewards supervise the model to develop reasoning patterns that lead to correct final selections, without requiring labeled reasoning traces.

Training objectives. Building on the above verifiable reward, we consider two training formulations. In the first step, optimization is performed using Group Relative Policy Optimization (GRPO), yielding the loss $\mathcal{L}_{\text{RL}}(X_i, Y_i; \phi)$, where ϕ denotes the parameters of the judge model. As an alternative, we cast judge training as a preference optimization problem. Preference pairs are constructed by treating correct selections as chosen responses and incorrect selections or format errors equally as rejected responses, collapsing the three-level reward into a binary preference signal. Specifically, we define $X_i = (x_i, y_i^a, y_i^b)$ with corresponding labels $Y_i = l_i$, where x_i is the input prompt including the problem statement and candidate solutions, (y_i^a, y_i^b) are two sampled judge responses, and $l_i \in \{a, b\}$ indicates which response is preferred. The associated training loss is denoted by $\mathcal{L}_{\text{PO}}(X_i, Y_i; \phi)$. For simplicity, we use $\mathcal{L}(X_i, Y_i; \phi)$ to denote the training loss, encompassing both formulations.

4. Data-Reweighted Training of LLM Judges

Overview. As illustrated in Figure 2, we optimize judge parameters at the lower level and trainable data weights at the upper level. The key insight is that data weights should be optimized to improve performance on a held-out meta dataset, ensuring the judge generalizes well despite the distribution shift between training and test data. In practice, $\mathcal{D}_{\text{tr}}$ corresponds to a large lower-level pool of training problems with candidate solutions generated by cost-efficient models, while $\mathcal{D}_{\text{meta}}$ is a smaller upper-level dataset constructed from problems and trajectories that more closely reflect the test-time setting—using temporally or difficulty-wise closer tasks and candidates generated by stronger policy models (see Section 5.1 for details).

Data reweighting methods. We propose three reweighting methods at different granularities:

- **Domain reweighting.** We split the dataset into K subsets from distinct domains (e.g., platform $\times$ difficulty

level), yielding training pools $\{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$. Each domain is assigned a weight α_k, offering coarse-grained control with strong prior knowledge of inter-domain imbalance.

- **Instance reweighting with instance table.** We maintain a lookup table that assigns a weight α_i to each sample (X_i, Y_i). This provides fine-grained per-instance control but requires parameters that scale with dataset size.

- **Instance reweighting with instance net.** We parameterize instance weights via a lightweight MLP that predicts weights based on sample loss:

$$\alpha_i = \text{MLP}_\theta(\mathcal{L}(X_i, Y_i; \phi)), \quad (2)$$

where θ are learnable parameters. This maintains a fixed parameter count independent of dataset size while providing better generalization.

We unify the notation as $\alpha_{(X_i, Y_i)}$ for all three methods. We slightly abuse notation for simplicity: when α is updated in the instance net case, we actually mean updating the instance net parameters θ.

Lower-level optimization. At the lower level, we optimize the judge parameters ϕ on the weighted training data. The training objective is a weighted sum of per-sample losses, allowing each sample's contribution to be adjusted:

$$\mathcal{L}_{\text{tr}}(\mathcal{D}_{\text{tr}}, \phi, \alpha) = \sum_{(X_i, Y_i) \in \mathcal{D}_{\text{tr}}} \alpha_{(X_i, Y_i)} \mathcal{L}(X_i, Y_i; \phi) \quad (3)$$

The optimal values of the judge parameters ϕ^* are obtained by optimizing the following objective:

$$\phi^*(\alpha) = \arg\min_{\phi} \mathcal{L}_{\text{tr}}(\mathcal{D}_{\text{tr}}, \phi, \alpha) \quad (4)$$

Note that only ϕ is optimized at this level while α remains fixed; the resulting ϕ^* is thus a function of α.

Upper-level optimization. At the upper level, we optimize the data weights α to minimize the loss on a held-out meta dataset $\mathcal{D}_{\text{meta}}$, using the judge $\phi^*(\alpha)$ from the lower level:

$$\mathcal{L}_{\text{meta}}(\mathcal{D}_{\text{meta}}, \phi^*(\alpha)) = \sum_{(X_i, Y_i) \in \mathcal{D}_{\text{meta}}} \mathcal{L}(X_i, Y_i; \phi^*(\alpha)) \quad (5)$$

The upper-level optimization problem is:

$$\alpha^* = \arg\min_{\alpha} \mathcal{L}_{\text{meta}}(\mathcal{D}_{\text{meta}}, \phi^*(\alpha)) \quad (6)$$

5

Algorithm 1 Data-Reweighted LLM Judge (DAJ)

Require: Training set $\mathcal{D}_{\text{tr}}$, meta set $\mathcal{D}_{\text{meta}}$, learning rates β_1, β_2
Ensure: Optimized judge parameters ϕ^*, data weights α^*
1: $\phi \leftarrow$ pretrained LLM checkpoint
2: **if** domain reweighting **then**
3:     $\alpha_k \leftarrow 1/K$ for all domains $k \in \{1, \dots, K\}$
4: **else if** instance table **then**
5:     $\alpha_i \leftarrow 1$ for all samples i
6: **else**
7:     $\alpha_\theta \leftarrow$ random initialization for instance net, where θ is the weights of instance net.
8: **end if**
9: **while** not converged **do**
10:     $\phi^{(t+1)} = \phi^{(t)} - \beta_1 \nabla_\phi \mathcal{L}_{\text{tr}}(\mathcal{D}_{\text{tr}}, \phi, \alpha)$ $\triangleright$ Lower-level
11:     $\alpha^{(t+1)} = \alpha^{(t)} - \beta_2 \nabla_\alpha \mathcal{L}_{\text{meta}}(\mathcal{D}_{\text{meta}}, \phi^*(\alpha))$ $\triangleright$ Upper-level
12: **end while**
13: **return** ϕ^*, α^*

Optimization algorithm. Solving the bi-level optimization problem in Equation (6) directly can be computationally prohibitive due to its nested structure. Following previous work (Choe et al., 2023), we use an approximate algorithm with a few unrolling steps. For example, under one-step unrolling, the update of the judge's weights can be expressed as:

$$\phi^{(t+1)} = \phi^{(t)} - \beta_1 \nabla_\phi \mathcal{L}_{\text{tr}}(\mathcal{D}_{\text{tr}}, \phi, \alpha), \qquad (7)$$

where β_1 is the learning rate in lower-level optimization. After obtaining the updated judge parameter $\phi^{(t+1)}$, we use it as an approximation to $\phi^*(\alpha)$ and update the reweighting parameter α as follows:

$$\alpha^{(t+1)} = \alpha^{(t)} - \beta_2 \nabla_\alpha \mathcal{L}_{\text{meta}}(\mathcal{D}_{\text{meta}}, \phi^*(\alpha)), \qquad (8)$$

where β_2 is the learning rate for upper-level optimization. The two optimization steps are conducted iteratively until convergence to obtain the optimal judge weights ϕ^* and the optimal reweighting parameter α^*. Algorithm 1 summarizes this process.

Discussion. The proposed framework establishes a closed-loop feedback between model learning and data reweighting. Specifically, the judge parameters ϕ are optimized on the reweighted training data, while the data weights α are updated based on performance on a held-out meta dataset. This feedback loop enables the data weighting strategy to co-evolve with the judge as training progresses. By explicitly maximizing performance on the meta set, the bi-level optimization framework automatically identifies training samples that transfer most effectively to the target distribution (see Appendix C for details).

Table 1. Performance comparison on BigCodeBench.

Zero-shot Methods		Test-time Scaling Methods	
	Ovr.		Ovr.
o3-mini (medium)	**33.1**	Skywork-o1 PRM	35.2
o1 (high)	32.4	DeepSeek GRM	33.8
o3-mini (high)	32.4	DAJ	**35.9**
Claude-3.7-Sonnet	32.4		

Table 2. Performance comparison on LiveCodeBench.

	Overall	Easy	Medium	Hard
Zero-shot Methods				
o4-mini (high)	**77.1**	**100.0**	**89.7**	**57.4**
Gemini-2.5-Pro-06-05	72.5	**100.0**	82.1	52.5
o4-mini (medium)	71.8	96.8	79.5	54.1
Test-time Scaling Methods				
Random	76.0	**100.0**	89.7	54.9
Skywork-o1 PRM	68.7	**100.0**	84.6	42.6
DeepSeek GRM	83.2	**100.0**	**94.9**	67.2
DAJ	**84.7**	**100.0**	89.7	**73.8**

Table 3. Performance comparison across policy models on LiveCodeBench.

	Overall	Easy	Medium	Hard
o4-mini (high)				
Random	75.8	98.4	80.8	61.1
Skywork-o1 PRM	74.1	96.8	**82.1**	57.4
DeepSeek GRM	75.6	**100.0**	79.5	60.7
DAJ	**76.3**	**100.0**	79.5	**62.3**
DeepSeek V3.2 Speciale				
Random	76.0	**100.0**	89.7	54.9
Skywork-o1 PRM	68.7	**100.0**	84.6	42.6
DeepSeek GRM	83.2	**100.0**	**94.9**	67.2
DAJ	**84.7**	**100.0**	89.7	**73.8**
Qwen2.5-Coder-32B				
Random	23.3	75.4	14.4	**2.5**
Skywork-o1 PRM	**25.2**	**80.7**	**20.5**	0.0
DeepSeek GRM	**25.2**	77.4	**23.1**	0.0
DAJ	**25.2**	80.6	17.9	1.6
Average over all policy models				
Random	58.4	91.3	61.6	39.5
Skywork-o1 PRM	56.0	92.5	62.4	33.3
DeepSeek GRM	61.3	92.5	**65.8**	42.6
DAJ	**62.1**	**93.5**	62.4	**45.9**

6

5. Results

5.1. Experimental Settings

We evaluate DAJ on LiveCodeBench (LCB, Jain et al. (2025)) and BigCodeBench (BCB, Zhuo et al. (2024)), two comprehensive benchmarks for code generation. We use temporal split for LiveCodeBench and difficulty-based split for BigCodeBench to construct the training and testing sets. More details are available in Appendix A. Domains are defined according to benchmark-specific characteristics to capture meaningful sources of distributional variation. On LiveCodeBench, we construct six domains based on the cross product of platform and difficulty level: AtCoder-easy, AtCoder-medium, AtCoder-hard, LeetCode-easy, LeetCode-medium, and LeetCode-hard. For BigCodeBench, we define seven domains according to task categories: Computation, Visualization, System, Time, Network, Cryptography, and General.

We generate reasoning trajectories and candidate solutions using Qwen3-Coder-30B-A3B (Yang et al., 2025) and Qwen2.5-Coder-32B (Hui et al., 2024) as the lower-level dataset. For the upper-level optimization, we employ o4-mini (high) (OpenAI, 2025) and DeepSeek V3.2 Speciale (DeepSeek-AI, 2025) to provide higher-quality data. This design ensures that the upper-level tasks align more closely with the test-time distribution. We fine-tune Qwen2.5-Coder-14B (Hui et al., 2024) for preference optimization and Qwen3-1.7B (Yang et al., 2025) for reinforcement learning. Full implementation details are included in Appendix B.

We compare DAJ against two categories of baselines: zero-shot methods and test-time scaling approaches. Zero-shot baselines consist of leading large language models evaluated on LiveCodeBench without applying test-time scaling. These include o4-mini variants and Gemini-2.5-Pro. Reported results are taken directly from the official Live-CodeBench and BigCodeBench leaderboards. We additionally compare against representative test-time scaling baselines. **Random** selects a candidate solution uniformly at random, reflecting the average performance of sampled candidates. **Skywork-o1 PRM** (He et al., 2024) is a process reward model that assigns step-level scores during generation [1]. **DeepSeek GRM** (Liu et al., 2025b) serves as a generative reward model baseline that does not incorporate data reweighting [2]. Best performance is highlighted in **bold** in all tables. pass@1 percentages are reported. For Live-CodeBench, results are broken down by Easy, Medium, and Hard difficulty levels; for BigCodeBench, overall performance is reported. Unless otherwise noted, DeepSeek V3.2 Speciale is used as the base policy model.

---

[1] https://huggingface.co/Skywork/Skywork-o1-Open-PRM-Qwen-2.5-7B

[2] https://huggingface.co/BBQGOD/DeepSeek-GRM-16B

5.2. Main Results

The overall performance of DAJ and all baseline methods is summarized in Table 2. On LiveCodeBench, DAJ achieves an overall accuracy of 84.7%, outperforming the strongest baseline, o4-mini (high), by 7.6 points. DAJ outperforms or match existing test-time scaling methods in most cases. The largest performance gains are observed on hard problems, where DAJ attains 73.8% compared to 67.2% for the strongest baseline. These results highlight the effectiveness of explicit, reasoning-based judgment in complex coding tasks. The substantial performance gap between DAJ and the random selection baseline demonstrates that the learned judge provides meaningful and informative selection signals, rather than relying on chance-level improvements from sampling. On BigCodeBench (Table 1), DAJ achieves 35.9%, outperforming all test-time scaling baselines and improving upon the best competing method, which attains 35.2% by Skywork-o1 PRM. Overall, DAJ exhibits comparable or superior performance relative to other test-time scaling approaches across different base policy models.

5.3. Results on Different Policy Models

We evaluate DAJ with a diverse set of policy models, including Qwen3-Coder-30B-A3B (Yang et al., 2025), o4-mini (high) (OpenAI, 2025), Qwen2.5-Coder-32B (Hui et al., 2024), and DeepSeek V3.2 Speciale (DeepSeek-AI, 2025). Candidate generation and selection details are provided in Appendix B.

As reported in Table 3, across all evaluated policy models, DAJ consistently outperforms or matches the performance of the strongest baseline methods. *This is noteworthy, as the reported results are with a backbone of only 1.7B parameters (Qwen3-1.7B (Yang et al., 2025)), which is significantly smaller than the previous state-of-the-art method, for example, the Skywork-o1 PRM has 7B parameters while DeepSeek GRM has 16B parameters.* On average, DAJ yields a +2.9 improvement over random selection. The largest gain is observed for DeepSeek V3.2 Speciale, where accuracy improves from 76.0% to 84.7%. These results demonstrate strong cross-policy generalization, indicating that DAJ generalizes well to the policy models.

5.4. Ablation Study

We conduct an ablation study to analyze the impact of training objectives and data reweighting on the performance of DAJ. We compare preference optimization objectives—DPO (Rafailov et al., 2023), KTO (Ethayarajh et al., 2024), ORPO (Hong et al., 2024), and reinforcement learning-GRPO (RLVR, Guo et al. (2025)). We evaluate four data reweighting strategies: no reweighting, domain reweighting, instance table, and instance net.
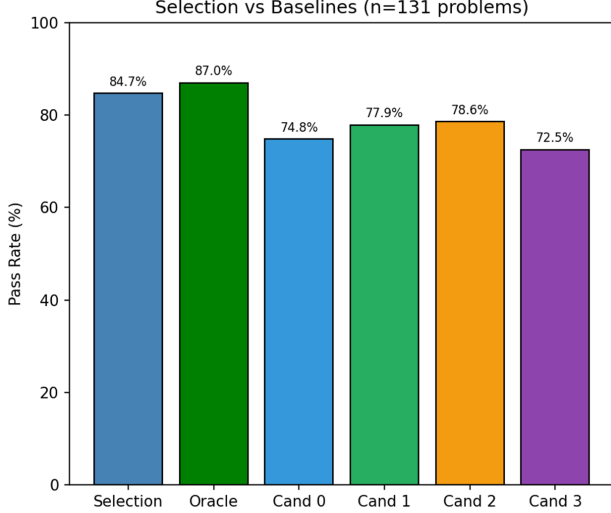
Figure 4. Comparison with candidate solutions and oracle judge on LiveCodeBench.

Comparison of training objectives and reweighting methods. The results are summarized in Table 4. Across all preference optimization methods, every reweighting strategy consistently outperforms or matches the no reweighting baseline. Performance gains are most pronounced on hard problems, where data imbalance is most severe. Among the training objectives, GRPO achieves the best overall performance, while KTO performs competitively even without paired preference data. Among reweighting strategies, domain reweighting provides strong domain-level flexibility, instance table enables fine-grained per-instance control, and instance net maintains a fixed parameter size while achieving improved generalization. Overall, the instance net achieves the best performance among all reweighting strategies, while all data reweighting strategies outperform the no reweighting baseline overall.

Comparison with candidate solutions and oracle judge. Figure 4 compares DAJ with individual candidate solutions as well as an oracle judge. The oracle judge is the theoretical upper bound of best-of-N, where a correct response is always selected, if possible. DAJ performs close to the oracle judge and significantly outperforms all individual candidate solutions, highlighting the quality of the learned judge.

6. Conclusion

We propose DAJ, a reasoning-based LLM judge trained with verifiable rewards under a bi-level data-reweighted training framework, designed to improve test-time scaling for code generation. Our method explicitly targets three major sources of distribution shift that hinder reliable judge

training, and automatically learns either domain-level or instance-level importance weights by optimizing generalization performance on a held-out meta dataset aligned with target benchmarks. Through this process, the model upweights important data, which encourages generalization. DAJ adopts an LLM-as-a-Judge paradigm with explicit reasoning, enabling detailed static code verification. Empirically, DAJ achieves state-of-the-art performance on LiveCodeBench and BigCodeBench, outperforming leading models and existing test-time scaling baselines. Empirical results demonstrate the effectiveness of combining principled data reweighting with reasoning-based LLM-as-a-Judge for robust code generation.

7. Impact Statements

This paper presents work aimed at advancing the field of machine learning. There are many potential societal consequences of our work, none of which we consider necessary to highlight here.

Table 4. Ablation study on training objectives and data reweighting.

(a) Preference Optimization

	Overall	Easy	Medium	Hard
DPO				
No reweighting	81.7	**100.0**	92.3	65.6
Domain	**83.2**	**100.0**	**94.9**	67.2
Instance table	82.4	**100.0**	**94.9**	65.6
Instance net	**83.2**	**100.0**	92.3	**68.9**
KTO				
No reweighting	83.2	**100.0**	**94.9**	67.2
Domain	83.2	**100.0**	92.3	68.9
Instance table	**84.0**	**100.0**	92.3	**70.5**
Instance net	**84.0**	**100.0**	**94.9**	68.9
ORPO				
No reweighting	81.7	**100.0**	92.3	65.6
Domain	82.4	**100.0**	89.7	68.9
Instance table	**83.2**	**100.0**	89.7	**70.5**
Instance net	**83.2**	**100.0**	**94.9**	67.2
Average over all preference optimization methods				
No reweighting	82.2	**100.0**	93.2	66.1
Domain	82.9	**100.0**	92.3	68.3
Instance table	83.2	**100.0**	92.3	**68.9**
Instance net	**83.5**	**100.0**	**94.0**	68.3

(b) Reinforcement Learning

	Overall	Easy	Medium	Hard
GRPO				
No reweighting	84.0	**100.0**	**89.7**	72.1
Domain	**84.7**	**100.0**	**89.7**	73.8

References

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 17682–17690, 2024.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Cao, Q., Wang, R., Zhang, R., Somayajula, S. A., and Xie, P. DreamPRM: Domain-reweighted process reward model for multimodal reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=ZyiBk1ZinG.

Choe, S. K., Neiswanger, W., Xie, P., and Xing, E. Betty: An automatic differentiation library for multilevel optimization. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=LV_MeMS38Q9.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Dai, N., Wu, Z., Zheng, R., Wei, Z., Shi, W., Jin, X., Liu, G., Dun, C., Huang, L., and Yan, L. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*, 2024.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025.

Ethayarajh, K., Xu, W., Muennighoff, N., Jurafsky, D., and Kiela, D. KTO: Model alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*, 2024.

Fan, S., Pagliardini, M., and Jaggi, M. DOGE: Domain reweighting with generalization estimation. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 12895–12915. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/fan24e.html.

Guo, D., Yang, D., Zhang, H., et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

He, J., Wei, T., Yan, R., Liu, J., Wang, C., Gan, Y., Tu, S., Liu, C. Y., Zeng, L., Wang, X., Wang, B., Li, Y., Zhang, F., Xu, J., An, B., Liu, Y., and Zhou, Y. Skywork-o1 open series, November 2024. URL https://doi.org/10.5281/zenodo.16998085.

Hong, J., Lee, N., and Thorne, J. ORPO: Monolithic preference optimization without reference model. *arXiv preprint arXiv:2403.07691*, 2024.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Irvine, R., Boubert, D., Raina, V., Liusie, A., Zhu, Z., Mudupalli, V., Korshuk, A., Liu, Z., Cremer, F., Assassi, V., et al. Rewarding chatbots for real-world engagement with millions of users. *arXiv preprint arXiv:2303.06135*, 2023.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Live-CodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Kim, S., Suk, J., Longpre, S., Lin, B. Y., Shin, J., Welleck, S., Neubig, G., Lee, M., Lee, K., and Seo, M. Prometheus 2: An open source language model specialized in evaluating other language models. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 4334–4353, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.248. URL https://aclanthology.org/2024.emnlp-main.248/.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL https://api.semanticscholar.org/CorpusID:6628106.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

KRAFTON AI and SKT AI. Continual post-training of LLMs via offline GRPO for mathematical reasoning, 2025. URL https://krafton-ai.github.io/blog/llm_post_training_en/.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Li, J., Sun, S., Yuan, W., Fan, R.-Z., Zhao, H., and Liu, P. Generative judge for evaluating alignment. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=gtkFw6sZGS.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode, 2022. URL https://arxiv.org/abs/2203.07814.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Liu, Y., Yao, Z., Min, R., Cao, Y., Hou, L., and Li, J. Pairwise RM: Perform best-of-n sampling with knockout tournament. *arXiv preprint arXiv:2501.13007*, 2025a.

Liu, Z., Wang, P., Xu, R., Ma, S., Ruan, C., Li, P., Liu, Y., and Wu, Y. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*, 2025b.

Luo, L., Liu, Y., Liu, R., Phatale, S., Guo, M., Lara, H., Li, Y., Shu, L., Zhu, Y., Meng, L., Sun, J., and Rastogi, A. Improve mathematical reasoning in language models by automated process supervision, 2024. URL https://arxiv.org/abs/2406.06592.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

OpenAI. OpenAI o3 and o4-mini system card. https://openai.com/index/o3-o4-mini-system-card/, 2025. Accessed: 2025-12-15.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

Pedregosa, F. Hyperparameter optimization with approximate gradient. In Balcan, M. and Weinberger, K. Q. (eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 737–746. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/pedregosa16.html.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

Rajeswaran, A., Finn, C., Kakade, S. M., and Levine, S. Meta-learning with implicit gradients. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 113–124, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/072b030ba126b2f4b2374f342be9ed44-Abstract.html.

Saunders, W., Yeh, C., Wu, J., Bills, S., Ouyang, L., Ward, J., and Leike, J. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*, 2022.

Shu, J., Xie, Q., Yi, L., Zhao, Q., Zhou, S., Xu, Z., and Meng, D. Meta-weight-net: Learning an explicit mapping for sample weighting. *Advances in neural information processing systems*, 32, 2019.

Snell, C. V., Lee, J., Xu, K., and Kumar, A. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. Math-Shepherd: A label-free step-by-step verifier for LLMs in mathematical reasoning. *arXiv preprint arXiv:2312.08935*, 3, 2023.

Wang, T., Kulikov, I., Golovneva, O., Yu, P., Yuan, W., Dwivedi-Yu, J., Pang, R. Y., Fazel-Zarandi, M., Weston, J., and Li, X. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2022. URL https://arxiv.org/abs/2203.11171.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-Thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

Wu, T., Yuan, W., Golovneva, O., Xu, J., Tian, Y., Jiao, J., Weston, J., and Sukhbaatar, S. Meta-rewarding language models: Self-improving alignment with LLM-as-a-meta-judge. *arXiv preprint arXiv:2407.19594*, 2024.

Xie, S. M., Pham, H., Dong, X., Du, N., Liu, H., Lu, Y., Liang, P., Le, Q. V., Ma, T., and Yu, A. W. DoReMi: Optimizing data mixtures speeds up language model pretraining. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=lXuByUeHhd.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Ye, J., Liu, P., Sun, T., Zhan, J., Zhou, Y., and Qiu, X. Data mixing laws: Optimizing data mixtures by predicting language modeling performance. In *The Thirteenth*

International Conference on Learning Representations, 2025a. URL https://openreview.net/forum?id=jjCB27TMK3.

Ye, Z., Li, X., Li, Q., Ai, Q., Zhou, Y., Shen, W., Yan, D., and LIU, Y. Learning LLM-as-a-judge for preference alignment. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=HZVIQE1MsJ.

Zhang, Q., Lyu, F., Sun, Z., Wang, L., Zhang, W., Hua, W., Wu, H., Guo, Z., Wang, Y., Muennighoff, N., et al. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*, 2025a.

Zhang, R., Qin, P., Cao, Q., Xue, E., and Xie, P. FunPRM: Function-as-step process reward model with meta reward correction for code generation, 2026. URL https://github.com/ruz048/FunPRM/blob/main/FunPRM.pdf.

Zhang, Z., Zheng, C., Wu, Y., Zhang, B., Lin, R., Yu, B., Liu, D., Zhou, J., and Lin, J. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*, 2025b.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=uccHPGDlao.

Zhou, Y., Xu, A., Wang, P., Xiong, C., and Joty, S. Evaluating judges as evaluators: The JETTS benchmark of LLM-as-judges as test-time scaling evaluators, 2025. URL https://arxiv.org/abs/2504.15253.

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A. Datasets and Benchmarks

LiveCodeBench. LiveCodeBench (Jain et al., 2025) is a contamination-free benchmark designed to evaluate large language models on code-related tasks. It continuously collects problems from LeetCode, AtCoder, and Codeforces, each annotated with release dates to enable temporal evaluation. The benchmark covers four coding scenarios: code generation, self-repair, code execution, and test output prediction, with problems categorized into easy, medium, and hard difficulty levels. We use the code-generation task, excluding Codeforces problems due to insufficient samples. To measure generalization to unseen problems and prevent data contamination, evaluation is restricted to tasks released after the model's training cutoff. In our setup, 601 problems published before 2025-02-01 are used for training, while 131 problems published between 2025-02-01 and 2025-05-01 are reserved for testing. We further construct a hierarchical training split, in which the lower-level dataset comprises problems released before 2024-08-01, and the upper-level dataset comprises problems released between 2024-08-01 and 2025-02-01, thereby providing stronger alignment with the target test distribution. Note that no problems are released exactly on the boundary dates (2024-08-01 and 2025-02-01), so the inclusive/exclusive distinction does not arise.

BigCodeBench. BigCodeBench (Zhuo et al., 2024) evaluates LLMs on practical programming tasks that require tool use and complex instruction following. The benchmark contains 1,140 function-level tasks involving the composition of multiple function calls drawn from 139 libraries. Each task is associated with an average of 5.6 test cases and achieves approximately 99% branch coverage. BigCodeBench includes two benchmark variants: BigCodeBench-Complete for code completion, BigCodeBench-Instruct for instruction following. BigCodeBench-Hard is a subset of BigCodeBench comprising 148 particularly challenging real-world tasks. In our experiments, we evaluate on BigCodeBench-Instruct using the hard split, while the remaining tasks are used for training. We split the training data into two subsets based on difficulty, measured by the pass rates of four policy models (Qwen3-Coder-30B-A3B, o4-mini (high), Qwen2.5-Coder-32B, and DeepSeek V3.2 Speciale). The hardest 20% of the tasks are used for the upper-level dataset, and the remaining 80% are used for the lower-level dataset.

B. Implementation Details

The proposed bi-level optimization–based data-reweighted training framework is implemented using the Betty library (Choe et al., 2023). Dataset preparation and evaluation strictly follow the official repositories for LiveCodeBench (Jain et al., 2025)[3] and BigCodeBench (Zhuo et al., 2024)[4]. We use Adam (Kingma & Ba, 2014) as the optimizer, with a learning rate of 10^{-6} for the lower-level optimization and a meta-learning rate of 10^{-4} for the upper-level optimization. Weight decay is set to 0, and the batch size is 1. Gradient accumulation is applied with 16 steps for lower-level optimization and 8 steps for upper-level optimization. All models are trained for 20,000 iterations. Preference optimization experiments are conducted using a single NVIDIA A100 GPU, while reinforcement learning experiments use two NVIDIA A100 GPUs, with one GPU dedicated to rollout and the other to training. All experiments are performed using bfloat16 (bf16) precision. The rollout is conducted with VLLM (Kwon et al., 2023) as the efficient inference engine. For candidate generation, we sample four candidates per problem for DeepSeek V3.2 Speciale and o4-mini (high), and eight candidates per problem for Qwen2.5-Coder-32B and Qwen3-Coder-30B-A3B. All candidates are decoded with temperature 0.7, top-p 0.8, top-k 20, repetition penalty 1.05, and a maximum output length of 8,192 tokens. Candidate selection is performed using the pairwise voting procedure described in Section 3 with $R = 8$ rounds. We adopt LoRA (Hu et al., 2022) for parameter-efficient fine-tuning, with rank $r = 32$, scaling factor $\alpha = 64$, and dropout rate set to 0.0. LoRA adapters are applied to the query, key, value, and output projection layers, as well as the gate, up, and down projection layers of the transformer. For instance-level reweighting, we employ a lightweight multilayer perceptron (MLP) with two hidden layers, each containing 10 hidden units.

We consider several preference optimization approaches for training the judge. Direct Preference Optimization (DPO, Rafailov et al. (2023)) directly optimizes preferences without relying on an explicit reward model; in our experiments, we set the temperature parameter to $\beta = 0.1$. Kahneman–Tversky Optimization (KTO, Ethayarajh et al. (2024)) is an alignment method that maximizes generation utility without requiring paired preference data. For KTO, we use $\beta = 0.1$, and the weights of the chosen and rejected samples are balanced. Odds Ratio Preference Optimization (ORPO, Hong et al. (2024)) is a reference-model-free, monolithic preference optimization approach that removes the need for a separate preference alignment stage. We set $\beta = 0.1$ for ORPO to balance the supervised fine-tuning loss and the preference loss. In addition to

---

[3] https://github.com/LiveCodeBench/LiveCodeBench
[4] https://github.com/bigcode-project/bigcodebench

preference optimization, we train the judge using Group Relative Policy Optimization (GRPO, Guo et al. (2025)), an online reinforcement learning method with verifiable rewards (RLVR). GRPO optimizes policies using relative comparisons within groups, and we set the group size to 16 in all experiments.

C. Grounding of the Three Reweighting Factors

The three factors claimed in Section 4—instance difficulty, task similarity, and trajectory alignment—are grounded in the joint design of the data splits and the bi-level objective rather than through hand-coded weighting rules.

Instance difficulty. Difficulty is encoded structurally at both the domain and instance levels. Domain reweighting partitions data by difficulty level (e.g., easy/medium/hard on LiveCodeBench), while the instance net takes per-sample loss as input, which naturally correlates with problem hardness. Because the meta set is enriched with challenging problems—temporally recent tasks on LiveCodeBench and the hardest 20% of tasks on BigCodeBench—the upper-level gradient drives the optimizer to upweight difficult training instances that are most informative for the target distribution.

Task similarity. Task similarity is governed by the composition of the meta set relative to the training set. On Live-CodeBench, the upper-level dataset consists of problems released closer in time to the test period, ensuring distributional proximity.

Trajectory alignment. Trajectory alignment is induced by the asymmetry in model quality between the two levels: lower-level training data is generated by cost-efficient models (Qwen3-Coder and Qwen2.5-Coder), while the upper-level meta set contains trajectories from stronger models (o4-mini and DeepSeek V3.2 Speciale) whose outputs resemble inference-time candidates. The bi-level gradient, therefore, favors training samples whose solution patterns transfer to the evaluation of stronger-model outputs.

In summary, the bi-level objective jointly emphasizes all three factors through end-to-end gradient-based optimization, without requiring any factor-specific heuristics.

D. Prompt Templates

The judge prompt, shown in Listing 1, instructs the model to follow a structured five-step evaluation protocol before selecting the best candidate solution. The five steps are: (1) *Syntax & Completeness Verification*, which checks that code is syntactically valid, complete, and has all necessary imports and function signatures; (2) *Algorithm Correctness Analysis*, which walks through the logic line by line, looking for off-by-one errors, wrong operators, and other common mistakes; (3) *Edge Case Enumeration & Testing*, which mentally tests the code against comprehensive categories of boundary inputs beyond the public examples; (4) *Input/Output Format Verification*, which confirms the code reads and writes data in the exact format required; and (5) *Runtime Safety & Performance*, which checks for division by zero, index-out-of-bounds, infinite loops, and excessive time or space complexity. Together, these steps decompose code evaluation into complementary static-analysis subtasks that cover the axes most predictive of correctness: well-formedness, algorithmic soundness, robustness, format compliance, and efficiency. By requiring the judge to produce a detailed, step-by-step analysis before making a final selection, the protocol instantiates the reasoning-based judging paradigm described in Section 3 and ensures that the judge's decision is grounded in interpretable evidence rather than a holistic impression.

```
=== PROGRAMMING PROBLEM ===

Title: {question_title}
Platform: {platform}
Difficulty: {difficulty}

Problem Statement:
{question_content}

Starter Code:
```python
{starter_code}
```
```

```
=== PUBLIC TEST CASES ===

The following are example test cases that illustrate the problem.
IMPORTANT: These are NOT the complete test suite. Hidden test cases will include edge
    ↪ cases, boundary conditions, and special inputs not represented below.

{test_cases}

=== CANDIDATE SOLUTION ===

--- Solution 1 ---

```python
{solution_code}
```

--- Solution 2 ---

```python
{solution_code}
```

=== SELECTION TASK ===

You are given {num_solutions} candidate solutions. Select the BEST solution that is
    ↪ most likely to pass ALL test cases (public examples and hidden test cases).

The public test cases you see are merely examples. The actual test suite contains
    ↪ hidden test cases specifically designed to catch edge cases, boundary
    ↪ conditions, and algorithmic errors that may not appear in examples.

Be skeptical and rigorous. Select the solution you are most confident will handle all
    ↪ possible valid inputs.

EVALUATION PROTOCOL (apply to each solution):

STEP 1: SYNTAX & COMPLETENESS VERIFICATION
- Is the code syntactically valid?
- Is the code complete?
- Are all variables defined before use? Are all imports present?
- Are function signatures complete with all parameters and return statements?

STEP 2: ALGORITHM CORRECTNESS ANALYSIS
- Is the chosen algorithm correct for this problem type?
- Walk through the logic line-by-line: Are operators correct? Are loop bounds correct?
    ↪ Are conditionals correct?
- Check for common errors: off-by-one errors, integer overflow, wrong comparison
    ↪ operators (< vs <=), incorrect loop termination
- Does the algorithm match the problem requirements exactly?

STEP 3: EDGE CASE ENUMERATION & TESTING
The public test cases show only "happy path" examples. Mentally test against these
    ↪ comprehensive edge case categories which apply to the problem:

- SIZE EDGE CASES: Empty input, Minimum size, Maximum size, Boundary sizes
- VALUE EDGE CASES: Minimum values, Maximum values, Zero, Negative numbers, Duplicates
- STRUCTURAL EDGE CASES: Sorted arrays, Reverse-sorted arrays, All elements identical,
    ↪ Alternating patterns, Single unique element among duplicates
- PROBLEM-SPECIFIC EDGE CASES: For graphs: disconnected components, self-loops, cycles;
    ↪ For strings: single character, all same character, alternating characters; For
    ↪ trees: single node, linear tree (linked list), complete tree; For intervals:
    ↪ overlapping, touching, disjoint, nested

STEP 4: INPUT/OUTPUT FORMAT VERIFICATION
- Input parsing: Does code read exactly the specified format?
```

```
    - Output format: Exact spacing? Exact newlines? Correct precision for floats? Correct
      ↪ ordering of multiple outputs?

    STEP 5: RUNTIME SAFETY & PERFORMANCE
    Check for errors that appear during execution: Division by zero, Index out of bounds,
      ↪ Infinite loops, Time complexity, Space complexity, Stack overflow

    OUTPUT FORMAT:
    Provide step-by-step analysis following the 5-step protocol above for each solution,
      ↪ then end your response with exactly:
    <selection>INDEX</selection>

    where INDEX is 1 to {num_solutions}, indicating the solution most likely to pass ALL
      ↪ test cases.
```

Listing 1. Judge prompts for reasoning-based judging
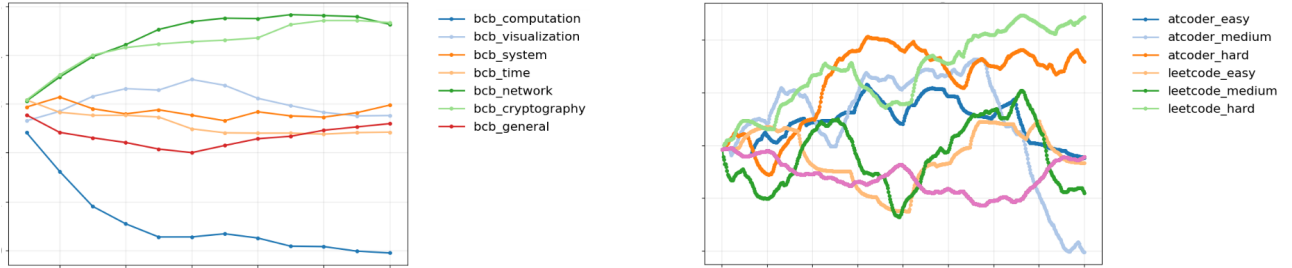
E. Evolution of Data Weights

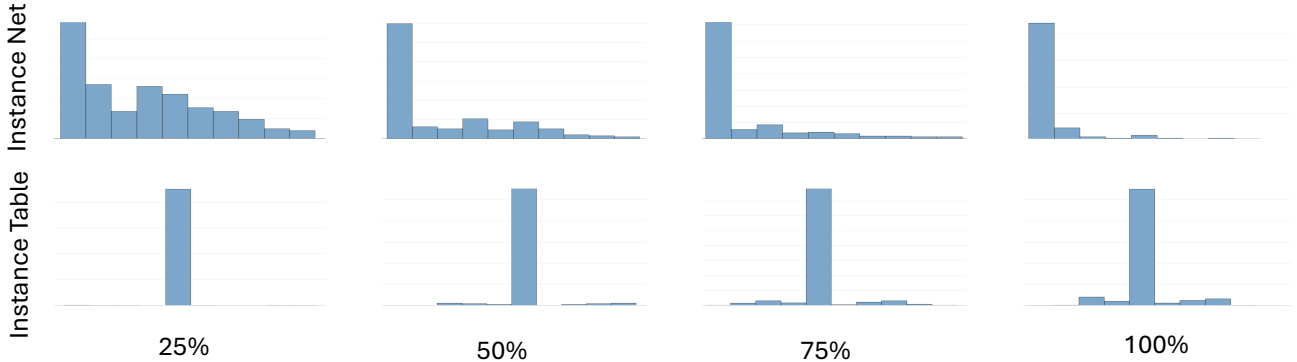*Figure 5.* Evolution of learned domain weights during training for domain reweighting.

Figure 6. Evolution of learned data weights during training (progress from 25% to 100%) for instance net (top part) and instance table (bottom part) reweighting methods.

Here, we provide the evolution of learned data weights during training for domain reweighting and instance net and instance table reweighting methods. The evolution of learned data weights provides further insight into the behavior of our method. As shown in Figure 5, domain weights start uniformly and rapidly diverge during training, emphasizing informative and challenging domains such as hard AtCoder tasks, while downweighting easier or less relevant ones. At the instance level (Figure 6), both the instance-net and instance-table approaches exhibit increasing weight dispersion over time, with training focus gradually shifting from lower-quality samples to higher-quality, more informative examples.