$\mathbf{V}_1$: UNIFYING GENERATION AND SELF-VERIFICATION FOR PARALLEL REASONERS

Harman Singh[1*] **Xiuyu Li**[1,*] **Kusha Sareen**[2] **Monishwaran Maheswaran**[1]
Sijun Tan[1] **Xiaoxia Wu**[3] **Junxiong Wang**[3] **Alpay Ariyak**[3] **Qingyang Wu**[3]
Samir Khaki[1] **Rishabh Tiwari**[1] **Long Lian**[1] **Yucheng Lu**[3] **Boyi Li**[1,4]
Alane Suhr[1] **Ben Athiwaratkun**[3] **Kurt Keutzer**[1†]

[1]UC Berkeley [2]Mila - Quebec AI Institute [3]Together AI [4]NVIDIA

ABSTRACT

Test-time scaling for complex reasoning tasks shows that leveraging inference-time compute, by methods such as independently sampling and aggregating multiple solutions, results in significantly better task outcomes. However, a critical bottleneck is *verification*: sampling is only effective if correct solutions can be reliably identified among candidates. While existing approaches typically evaluate candidates independently via scalar scoring, we demonstrate that models are substantially stronger at **pairwise self-verification**. Leveraging this insight, we introduce $\mathbf{V}_1$, a framework that unifies generation and verification through efficient pairwise ranking. $\mathbf{V}_1$ comprises two components: **$\mathbf{V}_1$-Infer**, an uncertainty-guided algorithm using a tournament-based ranking that dynamically allocates self-verification compute to candidate pairs whose relative correctness is most uncertain; and **$\mathbf{V}_1$-PairRL**, an RL framework that **jointly trains** a single model as both generator and pairwise self-verifier, ensuring the verifier adapts to the generator's evolving distribution. On code generation (LiveCodeBench, CodeContests) and math reasoning (AIME, HMMT) benchmarks, $\mathbf{V}_1$-Infer improves Pass@1 by up to 10% over pointwise verification and outperforms recent test-time scaling methods while being significantly more efficient. Furthermore, $\mathbf{V}_1$-PairRL achieves 7–9% test-time scaling gains over standard RL and pointwise joint training, and improves base Pass@1 by up to 8.7% over standard RL.

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable problem-solving abilities, largely driven by the paradigm of "System 2" thinking (OpenAI, 2024a;b; DeepSeek-AI, 2025) of generating extended chains of thought to reflect, refine, and verify answers at inference time. *Parallel reasoning*, which complements this sequential "deep thinking" by sampling multiple independent chains of thought to explore diverse solution paths, has emerged as a powerful technique for test-time scaling (Wang et al., 2023; Cobbe et al., 2021; Pan et al., 2025; Lian et al., 2025; Snell et al., 2024). In this setup, parallel sampling of multiple chains-of-thought is followed by an aggregation step to select the final answer. The simplest form of aggregation is to select the most common solution among the set of candidates (majority voting). While this suffices for domains like math where answers are objective and easily verifiable (Wang et al., 2023), this can not be used for more general domains which do not admit objective ground truth answers. Instead, accurate *self-verification* can support an aggregation method that selects the correct solution from the candidate set, even if it isn't the most common solution. Thus, fully leveraging parallel reasoning hinges on *accurate self-verification*: sampling N solutions is useful if the model can reliably identify the correct one.

Our experiments identify a critical bottleneck in existing approaches that use automatic verification to take advantage of inference-time compute: without a globally comparable scale of solution quality, existing models are not calibrated to evaluate candidate solutions independent of one another. In

[*]Equal contribution.
[†]Correspondence to: {harmans, xiuyu, keutzer}@berkeley.edu

addition, existing work suggests that in such settings, models used as verifiers are biased towards positively evaluating their own samples, even if those samples are incorrect (Lu et al., 2025). We find that, instead, self-verifying between candidate solutions via pairwise comparison leads to more robust and accurate outcomes. We explore two core questions. First, *how can we enable LLMs to more accurately self-verify candidate solutions obtained via parallel reasoning, by leveraging pairwise candidate self-verification?* Second, given that self-verification is typically applied after a model has already been trained, *can we instead train models to be better self-verifiers for parallel reasoning?*

While pairwise ranking has been extensively studied in reward modeling for LLM alignment (Christiano et al., 2017; Ziegler et al., 2019), this approach remains underexplored for self-verification in a parallel reasoning setting. Additionally, while reinforcement learning is commonly used to improve the solution-generation capabilities of LLMs on verifiable domains such as code and math (Shao et al., 2024; DeepSeek-AI, 2025), no existing methods effectively utilizes the parallel reasoning chains of LLMs at training time to jointly optimize both the generation and self-verification capabilities, resulting in distribution shifts at inference time. We demonstrate that inducing pairwise self-verification capabilities during RL training is an effective technique to improve test-time scaling performance for parallel reasoners. We leverage our observations to develop $\mathbf{V}_1$, a unified framework that includes a strong inference time scaling algorithm for parallel reasoning, as well as a reinforcement learning framework that induces self-verification capabilities during RL training with verifiable rewards.

Our contributions include the following:

1. We show that in parallelized reasoning, independent self-verification of candidate solutions suffers from calibration collapse due to lack of comparative reference. On the other hand, self-aggregation methods like recursive self-aggregation (RSA; Venkatraman et al., 2025) induce diversity collapse where Pass@N monotonically decreases with aggregation steps. This motivates pairwise verification as a principled alternative for self-verification and test-time scaling.

2. We develop $\mathbf{V}_1$-Infer, an uncertainty-guided pairwise verification algorithm. Rather than scoring solutions in isolation, it pairs candidates by employing a Swiss-system tournament refinement strategy that dynamically allocates verification compute to the most uncertain pairs. This approach provides a significant boost in selection accuracy, effectively improving the performance of the model closer to Pass@N of the original sampled responses . Notably, $\mathbf{V}_1$-Infer outperforms or matches RSA while requiring significantly fewer verification calls.

3. We develop $\mathbf{V}_1$-PairRL, an RL framework that co-trains a single model as both generator and pairwise self-verifier. Unlike prior co-training approaches that rely on pointwise rewards (Sareen et al., 2025; Liu et al., 2025a) or offline data (Venkatraman et al., 2025), $\mathbf{V}_1$-PairRL uses an online, co-evolving objective where generation and pairwise verification improve together. This ensures that as the generator improves, the verifier trains on in-distribution data from the model's current capabilities, thereby leading to stronger self-verification capabilities at inference time.

We evaluate our framework on code generation (LiveCodeBench, CodeContests) and math reasoning (AIME, HMMT) benchmarks. $\mathbf{V}_1$-Infer improves Pass@1 by up to 10% over pointwise verification and matches or exceeds RSA with a fraction of the compute. $\mathbf{V}_1$-PairRL achieves 7–9% test-time scaling gains over pointwise co-training standard RL, and improves base Pass@1 by up to 8.7% over the standard RL. Our results demonstrate that unified training for solution generation and self-verification capability, coupled with our pairwise self-verification technique, enables effective parallel reasoning. A detailed discussion of related work is provided in Appendix §A.

2 Why Improve Self-Verification?

From a test-time scaling perspective without external verifiers, parallel reasoning offers two prevalent mechanisms to generate the final solution: **a) self-selection** (verifying and choosing the best candidate) and **b) self-aggregation** (combining solutions to generate a better one). We analyze the limitations of current approaches in both categories to motivate our pairwise framework [1].

Pointwise self-verification suffers from calibration collapse. Standard pointwise verification assigns scalar scores to solutions in isolation. This approach is fundamentally limited by the lack of a comparative reference set. Statistically, latent utilities in choice models (e.g., Bradley–Terry)

[1] Majority voting is only applicable to scenarios with objective ground truth answers, such as math.

are identifiable only up to monotonic transformations, meaning absolute scores lack a globally comparable scale (Bradley & Terry, 1952). Consequently, pointwise scores exhibit high variance and poor cross-context calibration, often over-scoring plausible but incorrect solutions. Christiano et al. (2017) noted that for learning from human preferences, relative scores are much easier to provide for humans compared to absolute scores. *Pairwise* judgments simplify the task to a well-posed relative comparison. As shown in Figure 1 (left), this shift to relative ranking yields significantly higher top-1 self-ranking accuracy.

Self-aggregation induces diversity collapse. While self-aggregation methods, including Recursive Self-Aggregation (RSA) (Venkatraman et al., 2025; Khairi et al., 2025; Li et al., 2025b; Madaan et al., 2025) can help consolidate parallel reasoning chains and improve Pass@1, they may lead to *diversity collapse*. See Figure 1 (Left). The Pass@N score, representing the probability that *at least one* correct solution exists in a set of generated N solutions, monotonically decreases as aggregation steps increase for RSA. This indicates that RSA frequently discards or degrades correct outlier solutions during refinement. Specifically, since the refined Pass@1 rarely exceeds the *initial* Pass@N of the raw samples, the value of self-aggregation is unclear compared with a strong self-verifier that can select the best answer and reach near Pass@N performance. Instead of relying on implicit self-verification capabilities of aggregation based method, explicit and accurate self-verification can provide orthogonal improvements to aggregator-based approaches since each aggregation step (that may induce diversity collapse) can benefit from self-verification (that maintains pass@N), which can help select promising candidate solutions to aggregate.

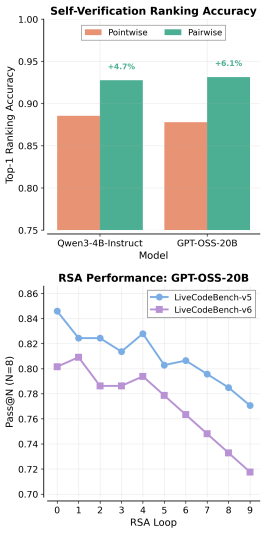

Figure 1: **(Top)** Pairwise outperforms pointwise self-verification (GPT-OSS-20B, LCB-V6). **(Bottom)** RSA diversity collapse on LCB-v5/v6. See §2.

3 LLMs can Self-Verify with $\mathbf{V}_1$-Infer

Pointwise verification is limited in its ability to capture relative nuances and lacks calibration. These limitations, coupled with a desire to maintain solution diversity, motivate us to propose ***V***$_1$*-Infer*, a pairwise verification framework designed for parallel reasoning. While we can use pairwise self-verification to score responses by the model, naively pairing solutions may require quadratic number of $C(N, 2)$ self-verification attempts. Our approach consists of a weighted aggregation mechanism and a two-phase budgeting strategy designed to maximize information gain with each new pair which allows us to efficiently scale test-time self-verification compute while achieving significant improvement in reasoning performance. The full algorithm is provided in App. §D. Fig. 2 for a high-level overview of the algorithm.

Uncertainty-Guided Score Aggregation. Standard pairwise voting often treats all "wins" equally, and thus fails to distinguish between a marginal preference (e.g., a 6 vs. 5 rating) and a decisive victory. Instead, we prompt the models to score their solutions in a pairwise setting instead of simply providing "correct" or "incorrect" as the output as in standard pairwise voting, which provides us with fine-grained information about the goodness of solutions.

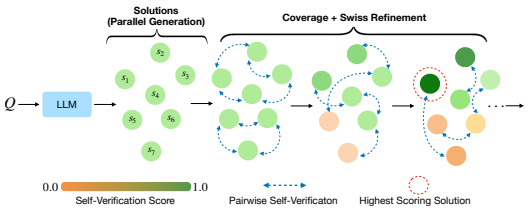

Figure 2: Swiss Refinement Overview. See Section 3.

To capture this nuance of relative quality, we adopt a weighted aggregation scheme in which the magnitude of the rating difference serves as a proxy for the judge's confidence. Given N candidate solutions $\mathcal{S}$ generated by an LLM, and a comparison budget B (number of self-verification LLM calls), a pairwise comparison between (s_i, s_j) using the same LLM outputs calibrated ratings $(r_i, r_j) \in [1, 10]$ for the two solutions. We define a confidence weight $w_{ij} = \max\left(\frac{|r_i - r_j|}{9}, \tau\right)$, where $\tau > 0$ is a small floor that ensures non-zero weights even for near-ties. Let $v_{ij} \in \{0, 0.5, 1\}$ denote the comparison outcome for s_i, corresponding to a win, tie, or loss, respectively. The estimated quality score μ_i is then computed as the uncertainty-weighted win rate $\mu_i = \sum_{j \in \mathcal{N}(i)} w_{ij} v_{ij} \big/ \sum_{j \in \mathcal{N}(i)} w_{ij}$,

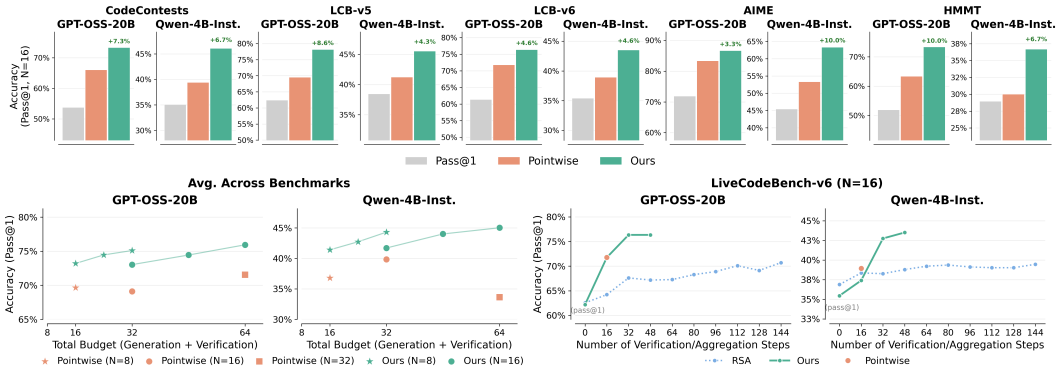

Figure 3: **(Top)** Performance after self-verification using $\mathbf{V}_1$-Infer compared with pointwise self-verification across benchmarks and models at N=16 base generations. Results presented for GPT-OSS-20B and Qwen-4B-Instruct-2507. **(Bottom Left)** Accuracy vs. total budget (generation + verification calls). Stars, circles, and squares denote $N = 8$, $N = 16$, and $N = 32$ base generations. $\mathbf{V}_1$-Infer consistently outperforms pointwise self-verification at equivalent budgets and shows monotonic performance scaling with compute. See Fig. 10 for per-benchmark results. **(Bottom Right)** Comparison with Recursive Self-Aggregation (RSA) (Venkatraman et al., 2025) on LCB-v6. $\mathbf{V}_1$-Infer method achieves higher accuracy with fewer self-verification calls. Results for GPT-OSS-120B and Qwen-4B-Thinking-2507 are in Fig. 7 and show similar trends.

where $\mathcal{N}(i)$ denotes the set of opponents compared against s_i. This formulation ensures that high-confidence judgments dominate the global ranking, while ambiguous comparisons contribute minimally to score variance.

Phase 1: Topology Coverage. A primary failure mode in low-budget pairwise ranking is *path dependence* in which solutions become "orphaned" or misranked due to insufficient pairwise comparisons with other solutions. We mitigate this by enforcing a minimum degree constraint to ensure all solutions are pairwise self-verified at least a minimum number of times. We begin with random disjoint pairings to guarantee global connectivity ($d_i \geq 1$), ensuring every solution enters the tournament. Subsequently, we iteratively target under-sampled nodes ($d_i < d_{\min}$) to meet a minimum degree threshold. Rather than selecting random opponents, we pair these low-degree nodes with candidates having the closest current mean score μ. This "anchors" solutions against comparable peers early in the process, preventing initial noise from propagating into the refinement phase.

Phase 2: Swiss Refinement. With the topology anchored, the remaining budget focuses on resolving rank ambiguity through an uncertainty-aware Swiss system. In each round, solutions are sorted by their current score μ, and we pair neighbors within a local window to minimize the score gap $|\mu_i - \mu_j|$ among unseen pairs. This strategy is grounded in active learning principles: under Bradley-Terry models, comparisons between items of similar skill (near-ties) yield the highest marginal information gain. By concentrating the judge budget on these ambiguous decision boundaries, $\mathbf{V}_1$-Infer efficiently reduces uncertainty where it matters most, achieving high ranking accuracy even with a sparse comparison graph ($K \ll N$).

3.1 Experimental Settings

Models and Benchmarks. We evaluate four diverse models: GPT-OSS-20B, Qwen3-4B-Instruct, GPT-OSS-120B, and Qwen-4B-Thinking. For code generation, we use LiveCodeBench-v5 (279 problems between date range of 24.08 to 25.02), LiveCodeBench-v6 (131 problems between 25.02-25.05 following official Qwen3-2507 models) (Jain et al., 2024), and CodeContests (165 problems) (Li et al., 2022). For math, we use AIME'25 and HMMT'25 (Balunović et al., 2025).

Evaluation Protocol and Baselines. For all methods, we first generate N candidate solutions independently from the model, then apply the verification strategy to select the final answer. We use $N \in \{8, 16\}$ for most experiments, and additionally $N = 32$ for pointwise verification in budget-matched comparisons. For comparison with RSA (Venkatraman et al., 2025) we keep the

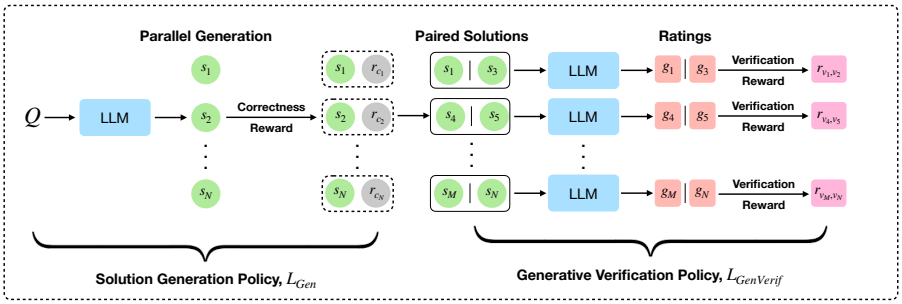

Figure 5: **$\mathbf{V}_1$-PairRL: Unified RL training for co-evolving generation and pairwise verification.** A single LLM is trained with two objectives: J_{Gen} optimizes solution generation using correctness rewards, while $J_{\text{PairVerif}}$ optimizes pairwise verification accuracy. The generator produces G solutions per problem, which are evaluated for correctness and paired for verification training. The verifier evaluates solutions in a paired setting. See Section 4 for more details.

population size as $N = 16$ with $k = 4$ as the aggregation size (number of solutions to aggregate at a time), with $T = 10$ RSA steps (number of recursive steps that the RSA algorithm is run for). See Venkatraman et al. (2025) for a detailed explanation of these parameters. Parameters for inference sampling and swiss-refinement algorithm are provided in Appendix C.

Pointwise Baseline. For fair comparison, pointwise self-verification uses the same 1–10 grading system as our pairwise method. The model evaluates each solution in isolation by assigning a score between 1 and 10, and the solution with the highest score is selected. This ensures that any performance differences stem from the pairwise comparison structure rather than the scoring mechanism. The exact prompts for both pointwise and pairwise verification (for code and math) are provided in Appendix F.

Verification Budget. $\mathbf{V}_1$-Infer allows flexible control over the verification budget. We report results for budget multipliers of $1\times$, $2\times$, and $3\times$ the number of initially generated solutions (N). For example, with $N = 16$ and budget $2\times$, we perform 32 pairwise comparisons. Figure 3 provides an overview of our results, comparing pointwise verification with pairwise verification (top), budget-matched evaluation (bottom left), and budgeted comparison with RSA (bottom right).

3.2 Results

Our goal is to measure the efficacy of $\mathbf{V}_1$-Infer compared to standard LLM-as-a-judge (pointwise) verification for parallel reasoners, and analyze how it compares with pointwise and aggregation-based test-time scaling methods in a budget-matched setting. Our results are summarized in Figure 3

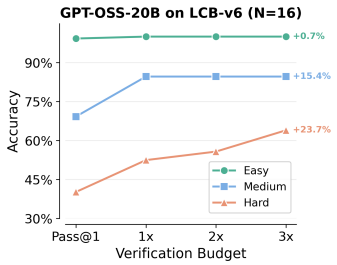

Figure 4: Accuracy vs. verification budget across difficulty levels (GPT-OSS-20B, LCB-v6, N=16). $\mathbf{V}_1$-Infer provides the largest gains on hard problems (+23.7%).

$\mathbf{V}_1$-Infer consistently outperforms pointwise self-verification. On CodeContests, GPT-OSS-20B improves from 66.06% to 73.33% (+7.3%), while Qwen3-4B-Instruct improves from 39.4% to 46.1% (+6.7%). On LiveCodeBench-v5, GPT-OSS-20B gains +8.6% and Qwen3-4B-Instruct gains +4.3%. On HMMT, GPT-OSS-20B gains +10.0% and Qwen3-4B-Instruct gains +6.7%. See Figure 3 (top) for more results. These results demonstrate that pairwise comparisons provide more informative signals than independent pointwise scores. We further observe that as verification budget increases, pairwise verification performance improves or stays consistent across most models and benchmarks (see Appendix Figure 7). While the results above are for same number of base solution generations (N=16) we find similar trends while comparing methods in a compute matched setting as well, Figure 3 (bottom left), for e.g., with compute budget of 64 model calls, with Qwen-4B-Instruct-2507, pointwise verification performs approx. 33% while pairwise verification reaches 45% accuracy on average on code gen. benchmarks.

$\mathbf{V}_1$-Infer enables improved test-time scaling. We compare against Recursive Self-Aggregation (RSA) (Venkatraman et al., 2025), a state-of-the-art test-time scaling method that iteratively refines solutions through an evolutionary self-aggregation process. As shown in Figure 3 (bottom right), $\mathbf{V}_1$-Infer achieves higher accuracy with significantly fewer LLM calls. On LiveCodeBench-v6 with N=16, our method reaches 76% Pass@1 with only 48 verification calls, higher compared to the maximum accuracy attained by RSA. This efficiency stems from our uncertainty-guided Swiss refinement, which concentrates comparisons on informative pairs rather than exhaustively aggregating or verifying all solution pairs. See Figures. 8, 9, 10 for extended results.

$\mathbf{V}_1$-Infer provides the largest gains on hard problems. Figure 4 shows how $\mathbf{V}_1$-Infer improves accuracy across different problem difficulty levels. On easy problems, Pass@1 is already near-optimal (99.3%). However, on hard problems where Pass@1 is only 40.2%, pairwise verification with budget 3x achieves 63.9%, a gain of +23.7%. Medium difficulty problems show an improvement of +15.4%. This pattern demonstrates that $\mathbf{V}_1$-Infer is most valuable precisely where it is needed most: on challenging problems where the model generates a diverse set of candidate solutions and accurate selection among them is critical for bridging the gap between Pass@1 and Pass@N.

$\mathbf{V}_1$-Infer outperforms random pairwise verification. To validate the effectiveness of our uncertainty-guided refinement algorithm, we compare against a baseline that randomly selects pairs for pairwise comparison. On LCB-v6 with GPT-OSS-20B at budget $3\times$, $\mathbf{V}_1$-Infer achieves 76.3% accuracy compared to 72.5% for random pairing, a gain of +3.8%. This demonstrates that the strategic pair selection in $\mathbf{V}_1$-Infer, which prioritizes comparisons between solutions with similar estimated quality scores, yields more informative judgments than naive random sampling.

4 Improving Parallel Self-Verification via Unified RL Training

The previous section demonstrated that pairwise self-verification significantly outperforms pointwise approaches at inference time. In the remainder of the paper we consider a second question: *can we explicitly train models to become stronger self-verifiers?* Current RL paradigms for reasoning focus almost exclusively on optimizing the generation of correct solutions, treating verification as either an afterthought or an external process. While recent work has explored co-training generators with verifiers (Sareen et al., 2025; Liu et al., 2025a), these approaches rely on *pointwise* rewards and fail to leverage the parallel responses that techniques like GRPO naturally generate during training. Additionally, pointwise verification is uncalibrated, which may make optimization hard. Other methods train for aggregation awareness (Venkatraman et al., 2025; Zhao et al., 2025b) but use offline data, limiting the model's ability to adapt to its own evolving generation distribution.

We propose $\mathbf{V}_1$-PairRL, a unified RL framework that trains a single LLM to be both a strong reasoner and an accurate *pairwise* self-verifier. The key insight is that generation and verification should *co-evolve*: as the generator improves, the distribution of responses changes, and the verifier must learn score increasingly high-quality solutions. This online, co-evolving setup ensures verification training data is always in-distribution for the model's current capabilities. However, unified training introduces unique challenges: naive implementations suffer from reward hacking, where the generator and verifier collude to maximize reward without improving actual capabilities. We address these challenges through careful reward design and pairing strategies.

4.1 Preliminaries: RL for LLMs

The goal of RL for LLMs is to maximize expected reward r of an LLM policy π_θ over a distribution of prompts $P(Q)$. This is commonly done with policy gradient methods. Specifically, Group-Relative Policy Optimization (GRPO) DeepSeek-AI (2025) and its variants Yu et al. (2025) have shown significant promise and stable optimization dynamics. In each episode, we sample prompts $q \sim P(Q)$ and a group of G rollouts per prompt $\{o_i\} \sim \pi_{\text{old}}(.|q)$. We maximize $J_{\text{Gen}}(\theta)$:

$$\mathop{\mathbb{E}}_{q,\{o_i\}}\left[\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1,t=1}^{G,|o_i|}\min\Big(\rho_{i,t}A_i,\text{clip}(\rho_{i,t},1-\epsilon,1+\epsilon)A_i\Big)\right] \tag{1}$$

where $A_i = r_i - \text{mean}(\mathbf{r})$ is the advantage computed over the group and $\rho_{i,t} = \pi_\theta(o_{i,t}|q,o_{i,<t})/\pi_{\text{old}}(o_{i,t}|q,o_{i,<t})$ is the importance sampling ratio.

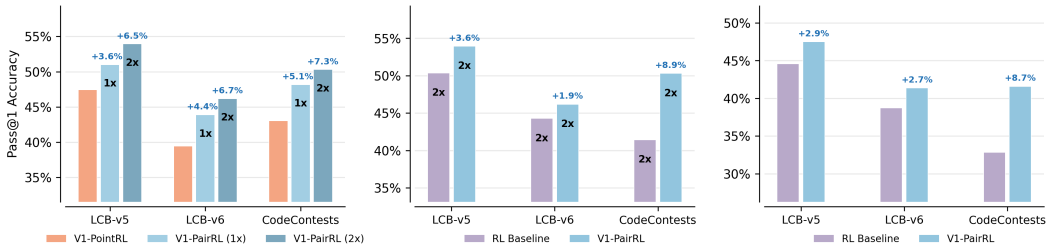

Figure 6: **$\mathbf{V}_1$-PairRL training results (N=16). (left)** Test-time scaling: $\mathbf{V}_1$-PairRL outperforms $\mathbf{V}_1$-PointRL and improves with increased verification budget. **(middle)** With $\mathbf{V}_1$-Infer at 2x budget: $\mathbf{V}_1$-PairRL outperforms RL baseline even when both use pairwise verification. **(right)** Base Pass@1: Co-training with pairwise verification improves generation quality over the RL baseline.

4.2 Co-evolving Solver-Verifier Training

Our training objective combines generation and pairwise verification in a unified RL formulation:

$$J(\theta) = J_{\text{Gen}}(\theta) + \lambda J_{\text{PairVerif}}(\theta) \tag{2}$$

J_{Gen} optimizes the likelihood of correct reasoning paths (using GRPO) and $J_{\text{PairVerif}}$ optimizes the accuracy of pairwise ranking judgments. Crucially, both objectives operate on the *same* rollouts: during each training step, the model generates G solutions per problem, which are used both to compute generation rewards and to form solution pairs for verification training. This ensures the verifier always trains on in-distribution data from the current policy. Figure 5 shows an overview of $\mathbf{V}_1$-PairRL. While our framework is general, we instantiate experiments on RL for code-generation following DeepCoder (Luo et al., 2025).

Rewards. For the solution generator, we use a standard binary correctness reward $r_{\text{gen}} \in \{0, 1\}$ based on passing all ground truth test cases. For the pairwise self-verification objective, the model compares two solutions (s_A, s_B) and outputs a confidence score between 1 and 10, which is normalized to $v_i \in [0, 1]$ for each solution. We reward the verifier based on how well its scores align with ground truth correctness $y_i \in \{0, 1\}$:

$$r_{\text{verif}} = \frac{1}{2} \sum_{i \in \{A,B\}} \mathbb{I}(|v_i - y_i| \leq 0.2) \cdot (1 - |v_i - y_i|) \tag{3}$$

The indicator function $\mathbb{I}(|v_i - y_i| \leq 0.2)$ implements a *sparsity threshold*: the verifier receives reward only when its score is within 0.2 of the ground truth (i.e., scoring a correct solution ≥ 0.8 or an incorrect solution ≤ 0.2). This design choice is critical for preventing reward hacking, as we discuss below.

Mitigating Reward Hacking. Unified training of generators and verifiers is prone to specific collapse modes. We address two critical forms of reward hacking:

1. *The Safe Bet Collapse:* Without the **sparsity threshold**, the verifier learns to output a safe, middle-ground score (e.g., $v_i = 0.5$) for every solution. This minimizes the risk of being "very wrong" but yields a meaningless discriminator. Sparsity threshold forces the model to commit to confident judgments: only scores near 0 or 1 receive positive reward.

2. *The Empty Solution Loop:* If the verifier is trained on pairs of two incorrect solutions, the generator may collapse into producing empty or trivially incorrect outputs. The verifier easily identifies these as incorrect (scoring them near 0), receiving high reward. This creates a situation where the generator degrades to maximize the verifier's ease of judgment. To prevent this, we enforce a strict pairing strategy: we only trigger verification training when we can form pairs containing at least one correct solution (Correct-Incorrect or Correct-Correct pairs).

4.3 Experimental Settings

We instantiate $\mathbf{V}_1$ on RLVR for code generation following the DeepCoder recipe (Luo et al., 2025). Models are trained on the DeepCoder training set, which comprises 24K verified coding problems,

utilizing binary rewards determined by executing generated code against ground-truth test cases (1 if all pass, 0 otherwise). We track solution generation performance on the DeepCoder validation set.

Models and Benchmarks. We train Qwen3-4B-Instruct-2507, an instruction-tuned model, following the DeepCoder experimental protocol. We evaluate trained models on LiveCodeBench V5, V6, and CodeContests. See Appendix §C.1 for evaluation hyperparameters.

Pairwise Verification Training. For $J_{\text{PairVerif}}$, we group multiple verification prompts (problem + solution pairs from the solver) rather than generating multiple rollouts for a single prompt. This strategy allows us to leverage information from all pairwise comparisons without increasing the total rollout budget. The advantage for verification rollouts reduces to a REINFORCE-style estimator with a mean baseline calculated across prompts.

Baselines. We compare our approach against two primary baselines: (1) a standard **RL baseline** trained solely for generation without a verification objective, and (2) **$\mathbf{V}_1$-PointRL**, a model jointly trained with pointwise verification rewards under the same setup. Training details for the pointwise baseline are provided in Appendix B.

Training Setup. We adopt the DAPO (Yu et al., 2025) configuration, removing the KL penalty, using Clip High, and applying token-level loss, and follow Dr. GRPO (Liu et al., 2025b) by removing standard deviation normalization. To ensure a fair comparison, we enforce a fixed compute budget of 8 total rollouts per problem. The baseline allocates all 8 rollouts to the solver, while co-evolving models ($\mathbf{V}_1$-PairRL and PointRL) split this budget into 4 solver and 4 verifier rollouts. We verified that training the baseline for a longer duration with fewer rollouts does not improve performance, confirming that gains stem from the co-training objective. All models are trained for 150 steps, with checkpoints selected based on the validation accuracy (pass@1). Prompts used for code-generation training are in Section E.

4.4 Results

$\mathbf{V}_1$-PairRL Offers Superior Test-Time Scaling Capabilities. To evaluate the test-time scaling benefits of co-trained verification, we compare $\mathbf{V}_1$-PairRL against $\mathbf{V}_1$-PointRL, our pointwise verification baseline trained with the same co-evolving setup. As shown in Figure 6(a), pairwise self-verification consistently outperforms pointwise across all benchmarks at N=16. On LiveCodeBench-v5, $\mathbf{V}_1$-PairRL achieves 53.9% (2x budget) compared to 47.4% for $\mathbf{V}_1$-PointRL (+6.5%). Similar improvements are observed on LiveCodeBench-v6 (+6.8%) and CodeContests (+7.3%). $\mathbf{V}_1$-PairRL exhibits positive scaling with verification budget: increasing budget yields consistent gains across benchmarks, demonstrating that the model learns to effectively leverage additional pairwise comparisons.

$\mathbf{V}_1$-PairRL with $\mathbf{V}_1$-Infer Outperforms RL Baseline with $\mathbf{V}_1$-Infer. In a test-time scaling setup with the same algorithm a key question arises: does $\mathbf{V}_1$-PairRL show improved performance compared to the standard RL baseline? To investigate, we apply $\mathbf{V}_1$-Infer at inference time to both $\mathbf{V}_1$-PairRL and the RL baseline, using identical 2x verification budgets. As shown in Figure 6 (middle), $\mathbf{V}_1$-PairRL consistently outperforms the RL baseline even when both leverage pairwise verification: +3.6% on LiveCodeBench-v5, +1.9% on LiveCodeBench-v6, and +8.9% on CodeContests.

$\mathbf{V}_1$-PairRL improves RL Baseline. Co-training for pairwise self-verification yields substantial improvements in generation quality. As shown in Figure 6(right), $\mathbf{V}_1$-PairRL achieves consistent Pass@1 improvements over the RL baseline across all three code generation benchmarks at N=16: +2.9% on LiveCodeBench-v5, +2.7% on LiveCodeBench-v6, and +8.7% on CodeContests. The gains demonstrate that jointly optimizing for generation and pairwise verification creates a beneficial learning signal that improves the model's underlying reasoning capabilities (which can also include improving the self-verification capability within long chains of thought), not just its verification accuracy. Our results echo prior findings by Sareen et al. (2025), who showed that co-training with pointwise verification improves the base model's Pass@1 performance. We show that pairwise verification takes this one step further: $\mathbf{V}_1$-PairRL also outperforms $\mathbf{V}_1$-PointRL in generation quality, particularly on CodeContests (by about +6%) (see Appendix Figure 11).

5 Conclusion

We presented $\mathbf{V}_1$, a unified framework for advancing self-verification in parallel reasoning, grounded in the insight that pairwise comparison constitutes a fundamentally more robust primitive for verifica-

tion than absolute scoring. We introduced $\mathbf{V}_1$-Infer, which employs tournament-based refinement to dynamically allocate compute toward ambiguous pairs, significantly outperforming pointwise verification and prior aggregation techniques with lower computational overhead. We further improved these results by complementing $\mathbf{V}_1$ with a post-training approach, $\mathbf{V}_1$-PairRL, which demonstrated that jointly training a single model for both generation and pairwise self-verification unlocks superior test-time scaling compared to standard RL and pointwise baselines. By unifying generation with pairwise verification, $\mathbf{V}_1$ provides a robust framework for both effective RL training and scalable parallel reasoning.

REFERENCES

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions, February 2025. URL https://matharena.ai/.

R Bradley and M Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. In *International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=aJeLhLcsh0.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. LLM-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14165–14178, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.792. URL https://aclanthology.org/2023.acl-long.792/.

Ammar Khairi, Daniel D'souza, Marzieh Fadaee, and Julia Kreutzer. Making, not taking, the best of n, 2025. URL https://arxiv.org/abs/2510.00931.

Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning, 2025. URL https://arxiv.org/abs/2508.21107.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation, 2025a. URL https://arxiv.org/abs/2502.14382.

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, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Zichong Li, Xinyu Feng, Yuheng Cai, Zixuan Zhang, Tianyi Liu, Chen Liang, Weizhu Chen, Haoyu Wang, and Tuo Zhao. Llms can generate a better answer by aggregating their own responses. *arXiv preprint arXiv:2503.04104*, 2025b.

Long Lian, Sida Wang, Felix Juefei-Xu, Tsu-Jui Fu, Xiuyu Li, Adam Yala, Trevor Darrell, Alane Suhr, Yuandong Tian, and Xi Victoria Lin. Threadweaver: Adaptive threading for efficient parallel reasoning in language models, 2025. URL https://arxiv.org/abs/2512.07843.

Xiaoyuan Liu, Tian Liang, Zhiwei He, Jiahao Xu, Wenxuan Wang, Pinjia He, Zhaopeng Tu, Haitao Mi, and Dong Yu. Trust, but verify: A self-verification approach to reinforcement learning with verifiable rewards, 2025a. URL https://arxiv.org/abs/2505.13445.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025b. URL https://arxiv.org/abs/2503.20783.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling, 2025c. URL https://arxiv.org/abs/2504.02495.

Jack Lu, Ryan Teehan, Jinran Jin, and Mengye Ren. When does verification pay off? a closer look at llms as solution verifiers, 2025. URL https://arxiv.org/abs/2512.02304.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level. Notion Blog, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

Lovish Madaan, Aniket Didolkar, Suchin Gururangan, John Quan, Ruan Silva, Ruslan Salakhutdinov, Manzil Zaheer, Sanjeev Arora, and Anirudh Goyal. Rethinking thinking tokens: Llms as improvement operators, 2025. URL https://arxiv.org/abs/2510.01123.

Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models. *arXiv preprint arXiv:2410.12832*, 2024.

Sadegh Mahdavi, Branislav Kisacanin, Shubham Toshniwal, Wei Du, Ivan Moshkov, George Armstrong, Renjie Liao, Christos Thrampoulidis, and Igor Gitman. Scaling generative verifiers for natural language mathematical proof verification and selection. *arXiv preprint arXiv:2511.13027*, 2025.

OpenAI. Learning to reason with LLMs, 2024a. URL https://openai.com/index/learning-to-reason-with-llms/.

OpenAI. Learning to reason with llms, 2024b. URL https://openai.com/index/learning-to-reason-with-llms/.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. *arXiv preprint arXiv: 2504.15466*, 2025.

Chi Ruan, Dongfu Jiang, Yubo Wang, and Wenhu Chen. Critique-coder: Enhancing coder models by critique reinforcement learning, 2025. URL https://arxiv.org/abs/2509.22824.

Swarnadeep Saha, Xian Li, Marjan Ghazvininejad, Jason Weston, and Tianlu Wang. Learning to plan & reason for evaluation with thinking-llm-as-a-judge, 2025. URL https://arxiv.org/abs/2501.18099.

Kusha Sareen, Morgane M Moss, Alessandro Sordoni, Rishabh Agarwal, and Arian Hosseini. Putting the value back in rl: Better test-time scaling by unifying llm reasoners with verifiers, 2025. URL https://arxiv.org/abs/2505.04842.

Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. Scaling test-time compute without verification or rl is suboptimal, 2025. URL https://arxiv.org/abs/2502.12118.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Wenlei Shi and Xing Jin. Heimdall: test-time scaling on the generative verification, 2025. URL https://arxiv.org/abs/2504.10337.

Nishad Singhi, Hritik Bansal, Arian Hosseini, Aditya Grover, Kai-Wei Chang, Marcus Rohrbach, and Anna Rohrbach. When to solve, when to verify: Compute-optimal problem solving and generative verification for llm reasoning, 2025. URL https://arxiv.org/abs/2504.01005.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4O0v4s3IzY.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F. Christiano. Learning to summarize from human feedback. *CoRR*, abs/2009.01325, 2020. URL https://arxiv.org/abs/2009.01325.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rllm: A framework for post-training language agents, 2025. Notion Blog.

Shubham Toshniwal, Ivan Sorokin, Aleksander Ficek, Ivan Moshkov, and Igor Gitman. Genselect: A generative approach to best-of-n. *arXiv preprint arXiv:2507.17797*, 2025.

Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan Obando-Ceron, Yoshua Bengio, Brian R. Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, Nikolay Malkin, and Moksh Jain. Recursive self-aggregation unlocks deep thinking in large language models, 2025. URL https://arxiv.org/abs/2509.26626.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning, 2025. URL https://arxiv.org/abs/2506.03136.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2550–2575, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.167. URL https://aclanthology.org/2023.findings-emnlp.167/.

Chenxi Whitehouse, Tianlu Wang, Ping Yu, Xian Li, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. J1: Incentivizing thinking in llm-as-a-judge via reinforcement learning, 2025. URL https://arxiv.org/abs/2505.10320.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2025. URL https://arxiv.org/abs/2408.00724.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Kaiwen Zha, Zhengqi Gao, Maohao Shen, Zhang-Wei Hong, Duane S. Boning, and Dina Katabi. Rl tango: Reinforcing generator and verifier together for language reasoning, 2025. URL https://arxiv.org/abs/2505.15034.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction, 2025. URL https://arxiv.org/abs/2408.15240.

Eric Zhao, Pranjal Awasthi, and Sreenivas Gollapudi. Sample, scrutinize and scale: Effective inference-time search by scaling verification, 2025a. URL https://arxiv.org/abs/2502.01839.

Wenting Zhao, Pranjal Aggarwal, Swarnadeep Saha, Asli Celikyilmaz, Jason Weston, and Ilia Kulikov. The majority is not always right: Rl training for solution aggregation. *arXiv preprint arXiv:2509.06870*, 2025b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 2024.

Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

A Extended Related Work

Parallel reasoning and test-time scaling. Test-time scaling approaches fall into two paradigms: sequential methods such as long chain-of-thought that iteratively refine a single solution (Madaan et al., 2023), and parallel methods that generate multiple reasoning paths simultaneously (Cobbe et al., 2021; Setlur et al., 2025; Pan et al., 2025; Zhao et al., 2025a; Singhi et al., 2025; Lian et al., 2025). Parallel scaling helps explore diverse solution paths simultaneously; however, it requires effective mechanisms to combine various solutions. For math, majority voting is effective due to objective answers (Wang et al., 2023; Snell et al., 2024; Wu et al., 2025), but this approach is domain-specific and unavailable for domains such as code generation and scientific discovery, where answers are not directly comparable to each other via exact matching. For code generation, prior work (Li et al., 2025a) has explored test-time scaling but requires executable test cases in the evaluation data or generated by larger models like GPT-4 for clustering and selection, while μCode (Jain et al., 2025) leverages external, multi-turn execution feedback, both of which are specific to code generation. We focus on *self-verification* for parallel reasoners, where a model judges its own generations without external verified or feedback.

Self-verification and self-aggregation Early work by Weng et al. (2023) demonstrated that LLMs can verify their own chain-of-thought reasoning, improving accuracy on arithmetic and commonsense tasks. Related to this, there have been multiple studies analysing the self-refinement capabilities and limitations of LLMs (Madaan et al., 2023; Stechly et al., 2025; Huang et al., 2024), however, they focus on a sequential reasoning setting, while our focus is on explicit self-verification in parallel reasoning. Recently Lu et al. (2025) show that LLMs exhibit a bias toward accepting incorrect solutions during pointwise self-verification. On the other hand, self-aggregation methods like RSA (Venkatraman et al., 2025) and parallel-distill-refine (Madaan et al., 2025) combine solutions generated by the same model, but suffer from diversity collapse (Section 3), resulting in correct solutions being discarded during inference. Such aggregation-based approaches are orthogonal to self-verification and can be combined with better self-verification for more better performance and efficient test-time scaling. Our work addresses limitations of the above prior work by studying pairwise self-verification in a parallel reasoning setup, showing it significantly outperforms pointwise self-verification while preserving solution diversity.

Generative verifiers and reward models. Reward models have been central to LLM alignment and RLHF (Christiano et al., 2017; Ziegler et al., 2019; Stiennon et al., 2020), and have emerged as a natural mechanism for scoring and selecting LLM generations (Cobbe et al., 2021; Zheng et al., 2023). Recent work demonstrates that generative reward models, which produce chain-of-thought reasoning before scoring, substantially outperform discriminative approaches (Mahan et al., 2024; Zhang et al., 2025; Shi & Jin, 2025; Saha et al., 2025), with RL-trained verifiers showing improved judgment performance (Liu et al., 2025c; Whitehouse et al., 2025). Recognizing that pairwise comparison is often easier than absolute scoring, several works have explored pairwise reward models: PairRM (Jiang et al., 2023) trains a discriminative pairwise ranker for ensembling LLM outputs, while GenSelect (Toshniwal et al., 2025) and concurrent work by Mahdavi et al. (2025) scale generative verification for mathematical reasoning. Related to this, Zhao et al. (2025b) propose AggLM, which trains an aggregator via RL to synthesize correct answers for improving majority voting. However, these approaches use *separate* verifiers or aggregator models. Our work differs in two key aspects: (1) we study *self*-verification, where the same parallel reasoner model judges its own outputs. This eliminates the need for additional training with curated judge data, saves memory, and reduces the computational overhead associated with external verification. (2) We co-train the model to be both a solution generator and a pairwise self-verifier in a unified framework.

Co-training for generation and verification. Training separate verifiers incurs significant overhead in compute, memory, and data curation. Recent work has explored co-training a single model for both generation and verification. Sareen et al. (2025) train models to self-verify online during RL, ensuring the verifier sees in-distribution generations and uses this capability to perform better majority voting at test-time for math reasoning. Liu et al. (2025a) extends this to unified CoT verifiers, while Zha et al. (2025) co-trains a separate process reward model that provides intermediate rewards. Offline approaches train verifiers or aggregators on data sampled from the base model (Venkatraman et al., 2025; Ruan et al., 2025). For code generation specifically, several works co-train generators with

unit-test producers (Wang et al., 2025; Lee et al., 2025). However, existing co-training methods rely on *pointwise* verification rewards. Our $\mathbf{V}_1$-PairRL framework instead co-trains for *pairwise* self-verification in an online, co-evolving setup, which we show yields superior test-time scaling performance.

B $\mathbf{V}_1$-POINTRL TRAINING DETAILS

To provide a fair comparison for our pairwise verification approach, we train $\mathbf{V}_1$-PointRL using the same co-evolving setup as $\mathbf{V}_1$-PairRL. The key difference lies in the verification objective: instead of comparing pairs of solutions, $\mathbf{V}_1$-PointRL assigns an independent score to each solution.

Verification Reward. For pointwise verification, the model evaluates a single solution s and outputs a confidence score between 1-10 which is normalized to $v \in [0, 1]$. The reward is computed as:

$$r_{\text{verif}} = \mathbb{I}(|v - y| \leq 0.2) \cdot (1 - |v - y|) \tag{4}$$

where $y \in \{0, 1\}$ is the ground truth correctness. The sparsity threshold prevents the "safe bet" collapse mode where the model learns to always output $v = 0.5$ (in practice, not applying the sparsity threshold leads to collapse in solution generation performance and is necessary for stable training).

Training Configuration. All hyperparameters (learning rate, batch size, rollout allocation, etc.) are identical to $\mathbf{V}_1$-PairRL as shown in Table 1. The only difference is the verification prompt and reward computation.

Inference. At inference time, $\mathbf{V}_1$-PointRL generates N solutions and independently scores each one. The solution with the highest pointwise score is selected as the final answer.

C HYPERPARAMETERS

C.1 INFERENCE SAMPLING PARAMETERS

Inference Framework. We use SGLang (Zheng et al., 2024) for all inference experiments, which provides efficient batched inference for large language models.

Sampling Parameters. We set temperature $T = 0.6$ for all code generation experiments and $T = 1.0$ for all math reasoning experiments to encourage better exploration of the solution space. We use top-p sampling with $p = 0.95$ for all experiments. For experiments with $\mathbf{V}_1$-Infer in Section 3 where we test the inference algorithm on widely used open source LLMs like GPT-OSS-20B, GPT-OSS-120B, Qwen-4B-Instruct-2507, Qwen-4B-Thinking-2507, we set the max generation length to 32768. We find that this does not lead to truncation for most evaluation examples, as also found in Venkatraman et al. (2025). We keep the same generation length for $\mathbf{V}_1$-Infer, pointwise verification as well as RSA.

For trained model evaluations in Section 4 we find that training all models lead to longer chains of thoughts, as is commonly the case in RLVR (DeepSeek-AI, 2025). This leads to frequent truncation, especially for the baseline RL model. To resolve this, we adopt the truncate-and-continue-generation method from prior work (Yang et al., 2025; Sareen et al., 2025). When generation reaches the maximum token budget (`finish_reason=length`), we resume decoding by appending the partially generated assistant output to the message history and issuing a continuation request. If the truncated output does not contain a closing `</thinking>` tag, we append the following transition sentence before continuing: *"Considering the limited time by the user, I have to directly give the required response based on the reasoning till now directly.`</thinking>`"*. This explicitly closes the thinking block and forces the continuation to produce the final answer. If the truncated output already contains `</thinking>`, we directly continue generation without appending any additional text. We continue the generation for 2K tokens, making the total maximum generation length 34 K for all experiments in Section 4.

Number of Samples. We generate $N \in \{8, 16\}$ candidate solutions for most experiments. For budget-matched comparisons between pointwise and pairwise verification, we additionally evaluate pointwise verification with $N = 32$ samples to match the total LLM calls of pairwise verification with $N = 16$ and budget $3\times$. This budget-matched comparison is performed only for GPT-OSS-20B and Qwen3-4B-Instruct models.

Verification Budget. For $\mathbf{V}_1$-Infer experiments in Section 4, we evaluate verification budgets of $1\times$, $2\times$, and $3\times$ the number of base solutions (N). Full results across all budgets are provided in the appendix figures.

Experimental Runs. All inference experiments for base models (Section 4) are run once. All trained model results (Section 5) are run 3 times with different random seeds, and we report the mean. Due to the computational cost of running multiple seeds, we limit the budget exploration for trained models to $1\times$ and $2\times$, though we believe performance can further scale with additional budget for the pairwise method ($\mathbf{V}_1$-Infer).

Swiss Refinement Algorithm Parameters. For $\mathbf{V}_1$-Infer, we use the following settings: Minimum degree $d_{\min} = 2$ (each solution compared at least twice in coverage phase), Swiss window size $h = 8$, Confidence floor set to a small value, $\tau = 0.1$ for weighted aggregation. We do not tune parameters to any specific benchmark. We use the same parameters across benchmarks or models across all experiments.

C.2 Training Hyperparameters

Table 1 summarizes the key training hyperparameters. We train models using rLLM (Tan et al., 2025) and verl (Sheng et al., 2024) backend.

Table 1: Training hyperparameters for $\mathbf{V}_1$.

Hyperparameter	**Qwen3-4B-Inst**
Learning rate	1×10^{-6}
Batch size	64
Rollouts per prompt (G)	4 (8 for baseline RL)
Judge rollouts	4 (0 for baseline RL)
Max prompt length	10240
Max response length	24576
Temperature	0.6
Top-p	0.95
Clip ratio (low)	0.2
Clip ratio (high)	0.28
λ (loss weight)	1.0
KL coefficient	0
Entropy coefficient	0
Std normalization	No
Loss aggregation	token-mean

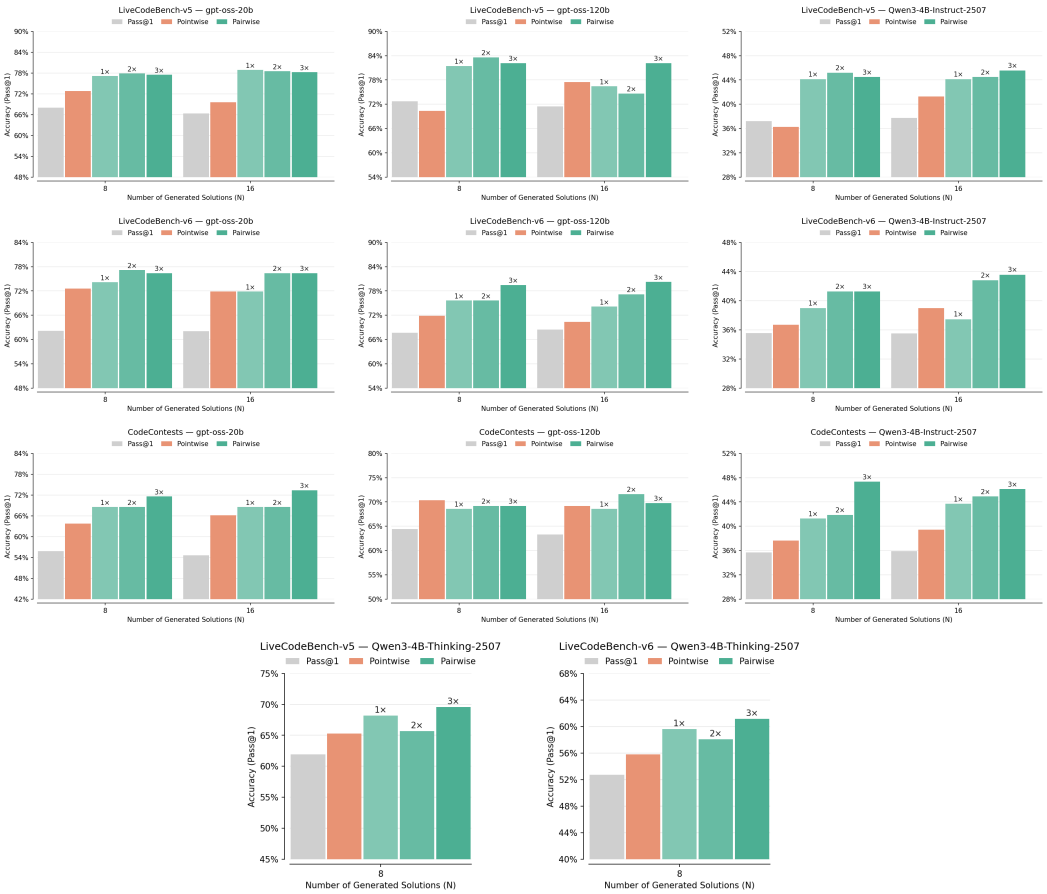

Figure 7: Self-verification bar plots across benchmarks and models (LiveCodeBench-v5, LiveCodeBench-v6, and CodeContests). Each plot compares Pass@1, pointwise verification, and pairwise verification (budgets $1\times/2\times/3\times$) at $N \in \{8, 16\}$. For Qwen3-4B-Thinking (bottom row), we run experiments only on LiveCodeBench at $N = 8$ to manage compute.

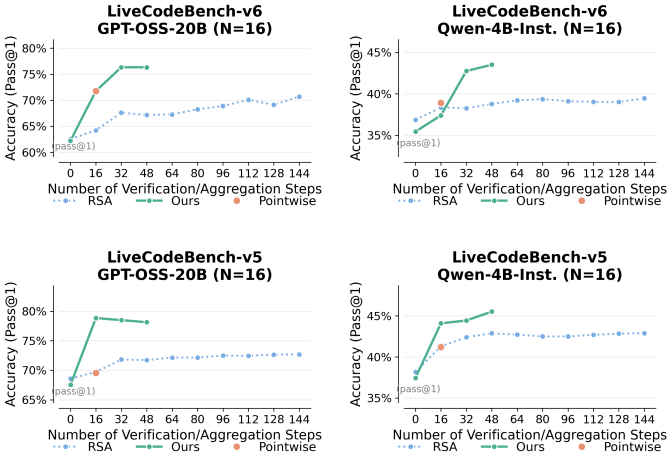

Figure 8: **Comparing Pairwise Verification with RSA.** Top row: LiveCodeBench-v6. Bottom row: LiveCodeBench-v5. Columns (left→right): gpt-oss-20b, Qwen3-4B-Instruct-2507. **Here number of generations by the base model is 16**, followed by verification (pointwise, pairwise) or aggregation (RSA).

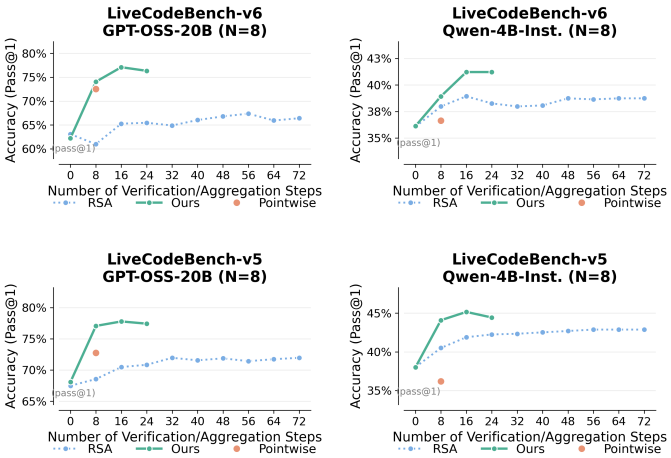

Figure 9: **Comparing Pairwise Verification with RSA.** Top row: LiveCodeBench-v6. Bottom row: LiveCodeBench-v5. Columns (left→right): gpt-oss-20b, Qwen3-4B-Instruct-2507. **Here number of generations by the base model is 8**, followed by verification (pointwise, pairwise) or aggregation (RSA).

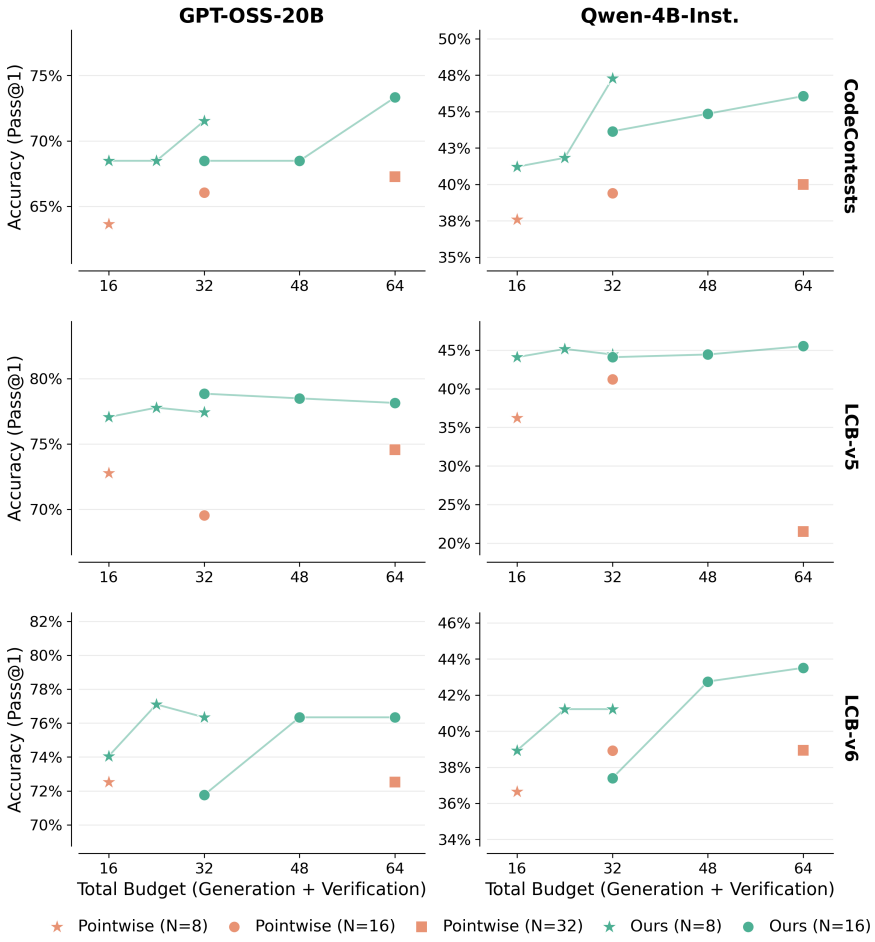

Figure 10: **Budget vs accuracy per benchmark.** Accuracy vs. total budget (generation + verification calls) for CodeContests (top row), LiveCodeBench-v5 (middle row), and LiveCodeBench-v6 (bottom row). Left column: GPT-OSS-20B. Right column: Qwen-4B-Inst. Stars denote $N = 8$ base generations, circles denote $N = 16$, squares denote $N = 32$ (pointwise only).

D DETAILED ALGORITHM

This section provides the full $\mathbf{V}_1$-Infer ranking algorithm and the specification of all its subroutine procedures. Algorithm 1 presents the main two-phase ranking procedure, followed by the helper procedures required for reproducibility.

Algorithm 1 Uncertainty-Guided Pairwise Ranking ($\mathbf{V}_1$-Infer)

Require: Problem x, candidates $\mathcal{S} = \{s_i\}_{i=1}^N$, budget B, min-degree $d_{\min}$, window h, floor τ
Ensure: Ranking π
Init: $\forall i$: $\mu_i \leftarrow 0.5$, $d_i \leftarrow 0$, $\mathcal{H} \leftarrow \emptyset$
$\mathcal{T} \leftarrow (\{\mu_i\}, \{d_i\}, \mathcal{H})$; used $\leftarrow 0$
Phase 1: Topology Coverage
while used $< B$ **and** $\exists i : d_i < d_{\min}$ **do**
  $\mathcal{P} \leftarrow$ COVERAGEPAIRS$(\mathcal{T}, d_{\min})$
  $\mathcal{O} \leftarrow$ PAIRSELFVERIFY$(x, \mathcal{S}, \mathcal{P})$
  UPDATESTATS$(\mathcal{T}, \mathcal{O}, \tau)$
  used $\leftarrow$ used $+|\mathcal{P}|$
end while
Phase 2: Swiss Refinement
while used $< B$ **and** $N > 2$ **do**
  $\pi \leftarrow$ RANK$(\{\mu_i\}_{i=1}^N)$
  $\mathcal{P} \leftarrow$ SWISSPAIRS$(\pi, \mathcal{T}, h)$
  $\mathcal{O} \leftarrow$ PAIRSELFVERIFY$(x, \mathcal{S}, \mathcal{P})$
  UPDATESTATS$(\mathcal{T}, \mathcal{O}, \tau)$
  used $\leftarrow$ used $+|\mathcal{P}|$
end while
return RANK$(\{\mu_i\}_{i=1}^N)$

State. We maintain a state $\mathcal{T} = (\{\mu_i\}_{i=1}^N, \{d_i\}_{i=1}^N, \mathcal{H})$, where μ_i is the current estimated quality score of solution s_i, d_i is the number of distinct opponents it has been compared against, and $\mathcal{H} \subset [N] \times [N]$ records previously compared pairs.

CoveragePairs. Ensures minimum degree while anchoring comparisons to similar-quality peers.

Algorithm 2 COVERAGEPAIRS$(\mathcal{T}, d_{\min})$

Require: State $\mathcal{T} = (\{\mu_i\}, \{d_i\}, \mathcal{H})$, minimum degree $d_{\min}$
Ensure: Disjoint pair set $\mathcal{P}$
$\mathcal{P} \leftarrow \emptyset$, $\mathcal{U} \leftarrow \emptyset$ ▷ used indices
$\mathcal{L} \leftarrow \{i : d_i < d_{\min}\}$; sort $\mathcal{L}$ by increasing d_i
for each $i \in \mathcal{L}$ **do**
  if $i \in \mathcal{U}$ **then**
    continue
  end if
  $\mathcal{C} \leftarrow \{j \neq i : j \notin \mathcal{U},\ (i,j) \notin \mathcal{H}\}$
  if $\mathcal{C} = \emptyset$ **then**
    continue
  end if
  $j \leftarrow \arg\min_{j \in \mathcal{C}} |\mu_i - \mu_j|$
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(i,j)\}$
  $\mathcal{U} \leftarrow \mathcal{U} \cup \{i,j\}$
end for
return $\mathcal{P}$

UpdateStats. This is where the mathematical aggregation is implemented. Note. In practice, the

Algorithm 3 UPDATESTATS$(\mathcal{T}, \mathcal{O}, \tau)$

Require: State $\mathcal{T} = (\{\mu_i\}, \{d_i\}, \mathcal{H})$, outcomes $\mathcal{O}$, floor $\tau > 0$
Ensure: Updated state $\mathcal{T}$

```
for each (i, j, w, r_i, r_j) ∈ O do
    v_ij ← I[w = A] + 0.5 · I[w = tie]
    w_ij ← max( |r_i − r_j| / 9 , τ )
    update μ_i, μ_j using the score aggregation rule (§3)
    if (i, j) ∉ H then
        d_i ← d_i + 1;   d_j ← d_j + 1
        H ← H ∪ {(i, j), (j, i)}
    end if
end for
```

update of μ_i is implemented incrementally using sufficient statistics, but the score aggregation rule in §3 fully specifies the aggregation semantics.

The above procedures jointly ensure (i) early stabilization of the comparison topology via minimum-degree coverage and (ii) efficient use of remaining budget through uncertainty-focused Swiss refinement, enabling accurate ranking with $O(N)$–scale pairwise verification.

SwissPairs. Allocates remaining budget to the most informative near-ties.

Algorithm 4 SWISSPAIRS$(\pi, \mathcal{T}, h)$

Require: Ranking π, state $\mathcal{T} = (\{\mu_i\}, \{d_i\}, \mathcal{H})$, window size h
Ensure: Disjoint pair set $\mathcal{P}$

```
P ← ∅, U ← ∅
for k = 1 to N do
    i ← π_k
    if i ∈ U then
        continue
    end if
    C ← ∅
    for t = k + 1 to min(k + h, N) do
        j ← π_t
        if j ∈ U then
            continue
        end if
        add (|μ_i − μ_j|, I[(i, j) ∈ H], j) to C
    end for
    if C ≠ ∅ then
        choose j with lexicographically minimal key in C
        P ← P ∪ {(i, j)}
        U ← U ∪ {i, j}
    end if
end for
return P
```

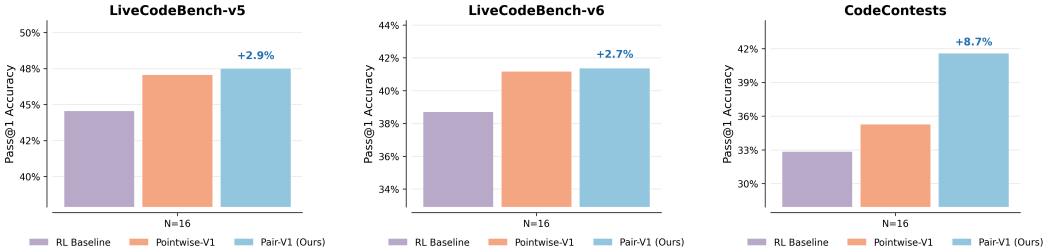

Figure 11: **V_1-PairRL vs RL baseline across benchmarks (N=16).** Left: LiveCodeBench-v5. Middle: LiveCodeBench-v6. Right: CodeContests. Co-training with pairwise verification consistently improves Pass@1 accuracy across all benchmarks.

E GENERATION PROMPTS

E.1 CODE GENERATION

Code Generation Prompt

```
**Problem**
{problem}


Think and reason step by step before coding the final solution
for the problem above.  Put your reasoning and any draft coding
solutions between <thinking> ...  </thinking> tags.  After the
reasoning (i.e.  after the </thinking> tag), use the format
provided in the problem above (code-block with backticks) to format
your final code solution.  Do not include any thinking within the
code block.


### Answer:
```

Note that "Do not include any thinking within the code-block." is added to the prompt, since in the absence of it, training the model (Qwen-4B-Instruct-2507) leads to it generating large amounts of comments within the code block, effectively leading to truncation of the response (due to reaching max length) and hence, 0 rewards and training collapse.

E.2 MATH SOLUTION GENERATION

Math Solution Generation Prompt

```
{problem}.  Let's think step by step and output the final answer
within \boxed{}.
```

F VERIFICATION PROMPTS

This section provides the exact prompts used for pointwise and pairwise self-verification in our experiments. Both verification methods use a 1–10 rating scale for fair comparison.

F.1 POINTWISE VERIFICATION PROMPTS

F.1.1 CODE GENERATION

Pointwise Code Verification Prompt

```
You are an expert code reviewer.  Rate the correctness of a
solution to a programming problem.

**Evaluation Guidelines:**
- Analyze the problem's requirements and constraints.
- Mentally trace the solution with test cases (including edge cases)
to verify correctness.
- Give a higher score if the solution is robust and fault-tolerant.

**Problem**
{problem}

**Solution**
{code}

**Output Format:**
First, provide your step-by-step reasoning.  Then, on a new line,
provide your final rating using the EXACT tags below.  Add no other
text after the tags.
<rating>INTEGER_1_TO_10</rating>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = correct & robust, 5 =
borderline, 1 = incorrect).
Please provide your analysis now.
```

F.1.2 MATH REASONING

Pointwise Math Verification Prompt

```
You are an expert math contest grader.  Rate the correctness of a
submission based solely on the final answer.

**Evaluation Guidelines:**
- Extract the submission's final answer.  Use any provided reasoning
only to help you assess whether the stated final answer is
trustworthy.  Do not award credit for method quality or rigor.
- Carefully analyze the problem statement and the submission to
assess whether the final answer is correct.  Grade only the final
answer.

**Problem**
{problem}

**Solution**
{solution}

**Output Format:**
First, provide your reasoning (what checks you performed).  Then,
on a new line, provide your final rating using the EXACT tag below.
Add no other text after the tag.
<rating>INTEGER_1_TO_10</rating>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = certainly correct, 8 = very
likely correct, 5 = uncertain/borderline, 3 = likely incorrect, 1 =
```

```
certainly incorrect).
Please provide your analysis now.
```

F.2 Pairwise Verification Prompts

F.2.1 Code Generation

Pairwise Code Verification Prompt

```
You are an expert code reviewer.  Compare two solutions to a
programming problem and rate their correctness.

**Evaluation Guidelines:**
- Analyze the problem's requirements and constraints.
- Mentally trace each solution with test cases (including edge
cases) to verify correctness.
- If both solutions appear correct, prefer the more robust and
fault-tolerant one.

**Problem**
{problem}

**Solution A**
{code_A}

**Solution B**
{code_B}

**Output Format:**
First, provide your step-by-step reasoning.  Then, on separate new
lines, provide your final ratings using the EXACT tags below.  Add
no other text after the tags.
<rating_A>INTEGER_1_TO_10</rating_A>
<rating_B>INTEGER_1_TO_10</rating_B>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = correct & robust, 5 =
borderline, 1 = incorrect).
- The higher rating wins.  Equal ratings imply a tie.
Please provide your analysis now.
```

F.2.2 Math Reasoning

Pairwise Math Verification Prompt

```
You are an expert math contest grader.  Compare two submissions and
rate correctness based solely on the final answer.

**Evaluation Guidelines:**
- Extract each submission's final answer.  Use any provided
reasoning only to help you assess whether the stated final answer
is trustworthy.  Do not award credit for method quality or rigor.
- Carefully analyze the problem statement and the submissions to
assess whether each final answer is correct.  Grade only the final
answer.

**Problem**
{problem}

**Solution A**
{sol_A}

**Solution B**
{sol_B}

**Output Format:**
```

```
First, provide your reasoning (what checks you performed).  Then,
on separate new lines, give ratings using the EXACT tags below.
Add no other text after the tags.
<rating_A>INTEGER_1_TO_10</rating_A>
<rating_B>INTEGER_1_TO_10</rating_B>
**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = certainly correct, 8 = very
likely correct, 5 = uncertain/borderline, 3 = likely incorrect, 1 =
certainly incorrect).
- Higher rating wins.  Equal ratings imply a tie.
Please provide your analysis now.
```