Cast a Wider Net: Coordinated Pass@K Policy Optimization for Code Reasoning

Anonymous ACL submission

Abstract

Repeated sampling with a verifier is the standard way to allocate test-time compute for code generation, with pass@K as the canonical metric. Yet the standard policy class draws K independent samples from a single answer distribution, so attempts often collapse onto near-duplicate reasoning paths and waste the budget on redundant rollouts. This failure is costly in competitive programming, where many problems admit multiple distinct algorithmic strategies and passing pass@K requires only one correct attempt. We propose Coordinated Pass@K Policy Optimization (CPPO), which turns pass@K generation into joint exploration over strategies: a planner emits a tuple of $K{=}4$ alternative high-level methods, and a shared solver attempts one solution per method. CPPO trains this joint policy with a multiplicative planner reward, $R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$, assigning credit only to valid strategy tuples that lead to verifier-confirmed pass@K success. Across APPS, CodeContests, and LiveCodeBench-v6, CPPO improves pass@4 over direct sampling, planning baselines, planner-only SFT, and pass@K-oriented RL under the same $K{=}4$ solver-attempt budget, with statistically significant gains on six of nine model–benchmark cells; the largest single gain is $+0.16$ on Qwen3.5-9B LiveCodeBench-v6 over the strongest baseline (PKPO, $0.588 \rightarrow 0.748$; paired bootstrap, $p < 0.05$).

1 Introduction

Repeated sampling against a verifier is the standard way to allocate test-time compute in code generation (Brown et al., 2024; Snell et al., 2025), with pass@K as the canonical metric for competitive-programming benchmarks (Chen et al., 2021; Li et al., 2022; Jain et al., 2025). Given a fixed attempt budget, the central question is how to allocate attempts effectively.

Standard pipelines spend this budget on K independent samples from one answer distribution. Once the model is moderately confident, these samples tend to collapse onto the dominant mode: on a programming problem, this can mean several rollouts that implement the same algorithm with only cosmetic changes—for example, four memoized variants of one dynamic-programming solution—even when the problem admits distinct alternatives. Recent pass@K training methods such as pass@K-only RL (Chen et al., 2025) and PKPO (Walder and Karkhanis, 2025), as well as diversity-aware RL methods such as Darling (Li et al., 2025), improve how answer samples are rewarded, but they do not change the policy class: the K attempts remain independent draws from one answer distribution, so the same mode-dropping failure can persist.

We instead train the model to generate the K attempts jointly, not as independent draws. We focus on code reasoning, where competitive-programming problems often admit multiple algorithmic strategies and any one correct attempt is enough to pass. A planner jointly emits a structured *strategy tuple* of $K{=}4$ high-level methods, each conditioned on earlier ones, and a shared solver attempts one solution per method. Improving pass@K therefore becomes a problem of producing complementary alternatives rather than resampling from a single answer distribution. To our knowledge, CPPO is the first pass@K post-training method to treat the K attempts as one learned strategy-tuple action rather than independently scored answer samples. We call this method Coordinated Pass@K Policy Optimization (CPPO; Figure 1); the full training algorithm is in §3.3.[1]

Table 1 compares CPPO against the strongest baseline from each family at the same $K{=}4$ bud-

[1] Here, CPPO denotes *Coordinated* Pass@K Policy Optimization: a joint planner–solver policy trained with the multiplicative planner reward $R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$.

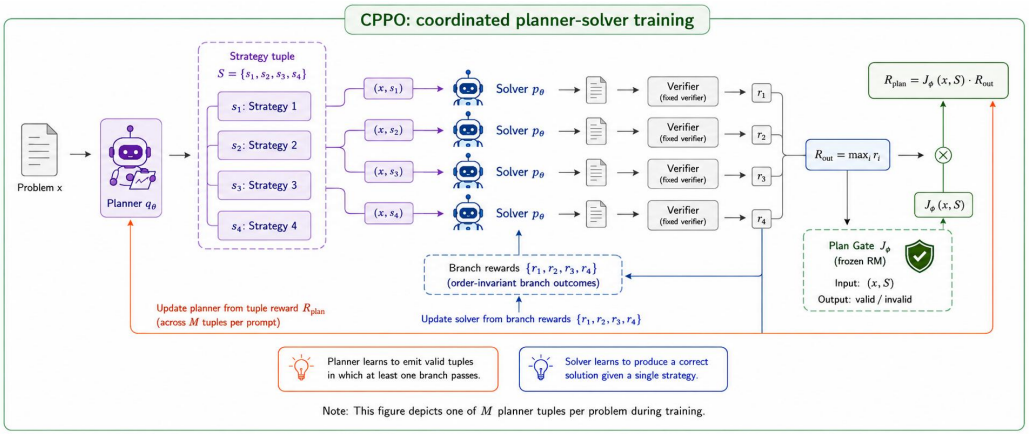


Figure 1: Overview of Coordinated Pass@K Policy Optimization. The planner q_Θ emits a strategy tuple $S = (s_1, \ldots, s_K)$; the shared solver p_Θ produces one solution per strategy; a verifier returns per-branch outcomes $r_i \in \{0, 1\}$, and the outcome reward $R_{\text{out}} = \max_i r_i$ scores pass@K success. A frozen reward model $J_\psi(x, S)$ gates plan validity, giving the planner reward $R_{\text{plan}} = J_\psi(x, S) \cdot R_{\text{out}}$, which is nonzero only when the tuple is accepted by J_ψ *and* at least one of its solver branches passes. Solver tokens are updated from within-tuple outcome advantages; planner tokens are updated from across-tuple advantages of R_{plan}.

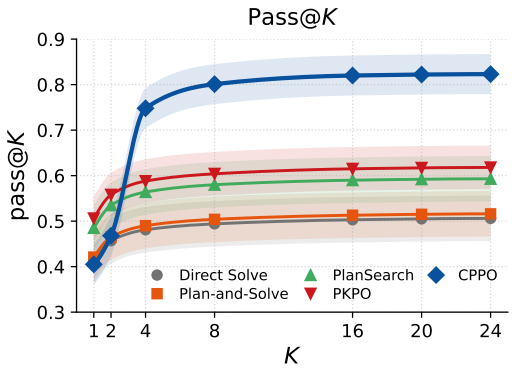


Figure 2: LiveCodeBench-v6 pass@K for Qwen3.5-9B (4B counterpart in Figure 5); all CPPO points use one K_{tuple}=4 planner and pool 4-tuple rollouts for K>4 (Table 18).

get. Planner SFT alone is insufficient: full CPPO improves over Tuple Planner SFT on every (size, benchmark) cell, with pass@4 gains of +0.08 to +0.30, isolating the contribution of joint planner–solver RL beyond planner SFT initialization. Our contributions are:

- **Pass@K as joint exploration over strategies, not answers.** We recast pass@K training from independent samples of a single answer distribution to a joint distribution over strategy tuples, which differs from the policy class used by prior pass@K-only RL (Chen et al., 2025) and PKPO (Walder and Karkhanis, 2025).

- **Multiplicative planner reward.** We introduce a multiplicative planner reward,

$$R_{\text{plan}} = J_\psi \cdot R_{\text{out}},$$

which assigns planner credit only when a strategy tuple is accepted by the learned validity gate J_ψ and at least one solver branch succeeds. We optimize this objective end-to-end with split-region GRPO updates for planner and solver tokens (§3.2).

- **Pass@4 gains at fixed budget.** Across APPS, CodeContests, and LiveCodeBench-v6, CPPO improves pass@4 over the strongest baseline in all nine (size, benchmark) cells of Table 1 and Appendix A under the same K=4 budget and verifier, with six cells reaching $p < 0.05$ under paired bootstrap; the largest single gain is +0.16 on Qwen3.5-9B LiveCodeBench-v6 over PKPO (0.588 → 0.748), and the gains persist at larger attempt budgets (Fig. 2).

2 Background and Related Work

RLVR and optimizer. We work in the Reinforcement Learning with Verifiable Rewards (RLVR) setting popularized by DeepSeek-R1 (Guo et al., 2025): a rule-based verifier $V(x, y) \in \{0, 1\}$ returns whether response y is correct for prompt x, and this binary signal serves as the reward for policy optimization. We use GRPO (Shao et al., 2024) with group-normalized advantages and a clipped, KL-regularized objective (Schulman et al., 2017). CPPO keeps this optimizer fixed and changes only

Table 1: Pass@4 on Qwen3.5-4B and 9B under the same $K{=}4$ solver-attempt budget and verifier.

Method	Base model	APPS (Hendrycks et al., 2021)	CodeContests (Li et al., 2022)	LiveCodeBench (LCBv6) (Jain et al., 2025)
Direct Solve	Qwen3.5-4B	$0.515_{\pm 0.050}$	$0.094_{\pm 0.034}$	$0.214_{\pm 0.041}$
Direct Solve	Qwen3.5-9B	$0.696_{\pm 0.043}$	$0.175_{\pm 0.038}$	$0.481_{\pm 0.050}$
Plan-and-Solve	Qwen3.5-4B	$0.530_{\pm 0.050}$	$0.098_{\pm 0.030}$	$0.255_{\pm 0.034}$
Plan-and-Solve	Qwen3.5-9B	$0.801_{\pm 0.040}$	$0.171_{\pm 0.038}$	$0.490_{\pm 0.050}$
PlanSearch	Qwen3.5-4B	$0.554_{\pm 0.050}$	$0.128_{\pm 0.033}$	$0.236_{\pm 0.033}$
PlanSearch	Qwen3.5-9B	$0.770_{\pm 0.042}$	$0.168_{\pm 0.037}$	$0.564_{\pm 0.050}$
Pass@K Training / RLVR	Qwen3.5-4B	$0.607_{\pm 0.044}$	$0.115_{\pm 0.032}$	$0.316_{\pm 0.046}$
Pass@K Training / RLVR	Qwen3.5-9B	$0.762_{\pm 0.044}$	$0.185_{\pm 0.040}$	$0.503_{\pm 0.052}$
PKPO	Qwen3.5-4B	$0.722_{\pm 0.045}$	$0.156_{\pm 0.036}$	$0.488_{\pm 0.050}$
PKPO	Qwen3.5-9B	$0.787_{\pm 0.040}$	$0.183_{\pm 0.038}$	$0.588_{\pm 0.048}$
UpSkill	Qwen3.5-4B	$0.661_{\pm 0.047}$	$0.121_{\pm 0.033}$	$0.411_{\pm 0.049}$
UpSkill	Qwen3.5-9B	$0.762_{\pm 0.046}$	$0.136_{\pm 0.038}$	$0.542_{\pm 0.048}$
Tuple Planner SFT	Qwen3.5-4B	$0.528_{\pm 0.047}$	$0.142_{\pm 0.038}$	$0.368_{\pm 0.052}$
Tuple Planner SFT	Qwen3.5-9B	$0.713_{\pm 0.054}$	$0.285_{\pm 0.044}$	$0.534_{\pm 0.054}$
CPPO	Qwen3.5-4B	$0.794^{\dagger}{}_{\pm 0.038}$	$0.243^{\dagger}{}_{\pm 0.047}$	$0.535_{\pm 0.048}$
CPPO	Qwen3.5-9B	$0.826_{\pm 0.041}$	$0.436^{\dagger}{}_{\pm 0.046}$	$0.748^{\dagger}{}_{\pm 0.044}$

Tuple Planner SFT emits all K plans in one autoregressive tuple, $q_\Theta(s_i \mid x, s_{<i})$; CPPO further trains this planner–solver policy with validity-gated outcome rewards. The matched iid-inference ablation is in Appendix H.

† CPPO significantly outperforms the strongest non-CPPO baseline in that cell under paired bootstrap over problems ($p < 0.05$, 1000 resamples). Unmarked CPPO gains are positive but not significant.

Baselines. We compare independent sampling, prompt-based planning, inference-time plan search, pass@K-oriented RL, diversity-oriented RL, and planner-only SFT; details are in §4.3 and Appendix R.

the policy factorization, reward, and token regions to which advantages are applied; Appendix I gives the standard GRPO equations.

Pass@K objective. With a fixed attempt budget K per prompt, pass@K is the indicator that at least one of K attempts is correct,

$$R_{\text{pass@}K}(x, y_{1:K}) \;=\; \mathbb{I}\Big\{\max_{i=1}^{K} V(x, y_i) = 1\Big\}. \tag{1}$$

The standard policy class for this objective draws the K attempts independently from a single answer distribution $p_\Theta(\cdot \mid x)$,

$$y_i \sim p_\Theta(\cdot \mid x), \quad i = 1, \ldots, K. \tag{2}$$

Mode coverage under independent sampling. A simple bound characterizes any pass@K method that still draws attempts independently. If $p_\Theta(\cdot \mid x)$ places probability ϵ on a correct-answer region $\mathcal{A}$, then none of K iid samples lands in $\mathcal{A}$ with probability $(1-\epsilon)^K$; for $\epsilon{=}0.05$ even $K{=}16$ leaves a missing-mass probability of ≈ 0.44. Re-weighting the per-sample reward (pass@K-only RL, PKPO) shifts p_Θ but does not change the iid sampling rule, so this bound carries over. CPPO drops the iid assumption: its K branches are sampled jointly through the planner (§3.1), allowing them to target different low-mass regions within a single trajectory.

Repeated sampling and test-time compute. Repeated sampling is the standard way to allocate test-time compute for code generation (Brown et al., 2024), and compute-optimal sampling can rival substantially larger models (Snell et al., 2025); CPPO targets this same pass@K regime but changes how the K attempts are generated.

Pass@K training and diversity rewards. Recent post-training methods improve repeated sampling by changing how answer samples receive credit. Pass@K training (Chen et al., 2025) and PKPO (Walder and Karkhanis, 2025) optimize set-level pass@K rewards over sampled answers, while Darling (Li et al., 2025), DIVER (Hu et al., 2026), and UpSkill (Shah et al., 2026) add diversity pressure through reward shaping, exploration bonuses, or latent-skill regularization. These methods can broaden the answer distribution, but they do not model the K attempts as a single coordinated policy action. CPPO instead treats the K-way set as a structured action: one planner rollout emits a coordinated tuple of strategies, and the solver executes one branch per strategy.

Verifiers and LLM-as-judge. Outcome verifiers (Cobbe et al., 2021) and process reward models (Lightman et al., 2024) score model outputs without ground-truth labels. CPPO's plan-validity reward model is a small generative verifier in this tradition, with labels obtained via LLM-as-judge (Zheng et al., 2023).

Planning for reasoning. Planning methods add structure before generation rather than after scoring. Plan-and-solve prompting (Wang et al., 2023a), Tree of Thoughts (Yao et al., 2023), and PlanSearch (Wang et al., 2024) use natural-language plans or search procedures at inference time over a frozen model. They show that explicit

plans can help reasoning, but they do not train a planner under the pass@K objective. CPPO brings planning into post-training: the planner learns to emit a single autoregressive strategy tuple whose later items condition on earlier ones, and receives validity-gated credit only when the tuple leads to verifier-confirmed pass@K success.

3 Coordinated Pass@K Policy Optimization

CPPO changes the policy class: instead of drawing K iid answers, it samples one tuple of K=4 algorithmic strategies. This factorization splits credit assignment: the solver must learn which strategy-conditioned branch succeeds, and the planner must learn which strategy tuple makes at least one branch succeed. CPPO therefore assigns branch-level outcome rewards to solver tokens and a tuple-level reward to planner tokens. Because tuple-level rewards are sparse and can be gamed by malformed plans, we gate planner credit with a narrow plan-validity reward model and train the system through a staged procedure. Table 2 contrasts CPPO with the closest planning and pass@K post-training baselines along the design axes that follow.

Table 2: Comparison of CPPO with the closest planning and pass@K post-training baselines along four axes: output granularity, search strategy, intervention stage, and RLVR training.

Method	Granularity	Search strategy	Stage	RLVR ready
Plan-and-Solve	answer	iid	inference	–
PlanSearch	plan features	enumerate	inference	–
Pass@K-only RL	answer	iid	post-training	✓
PKPO	answer	iid	post-training	✓
CPPO (ours)	**strategy**	**per-item**	**both**	✓

3.1 A Joint Strategy-Tuple Policy

Standard pass@K samples K answers independently from one distribution. CPPO instead samples one structured object: a tuple of K strategy sketches generated autoregressively, so each method conditions on its predecessors and avoids redundant branches. The planner emits the tuple

$$S = (s_1, \ldots, s_K), \tag{3}$$

with factorization

$$q_\Theta(S \mid x) = \prod_{i=1}^{K} q_\Theta(s_i \mid x, s_{<i}). \tag{4}$$

A shared solver produces one answer per method,

$$y_i \sim p_\Theta(\cdot \mid x, s_i), \quad i = 1, \ldots, K, \tag{5}$$

yielding the full trajectory

$$\tau = (S, y_{1:K}) \tag{6}$$

with probability

$$\pi_\Theta(\tau \mid x) = q_\Theta(S \mid x) \prod_{i=1}^{K} p_\Theta(y_i \mid x, s_i). \tag{7}$$

In the simplest implementation, a single backbone with shared parameters Θ induces both q_Θ (planner factor) and p_Θ (solver factor) from two prompt modes: the PLAN prompt produces a concise, task-focused strategy tuple (planner tokens), and the SOLVE prompt produces one answer conditioned on a single strategy (solver tokens). Planner and solver losses then apply to these disjoint token regions (exact prompts in Appendix C).

3.2 Coordinated Rewards and Credit Assignment

CPPO assigns credit at two granularities. Solver tokens receive branch-level verifier feedback: for branch i, $r_i^{\text{out}} = V(x, y_i)$, and the tuple-level outcome reward is the pass@K indicator

$$R_{\text{out}}(x, y_{1:K}) = \mathbb{I}\left\{\max_i V(x, y_i) = 1\right\}. \tag{8}$$

This outcome signal is appropriate for solver branches, but it does not tell the planner whether the visible tuple is well formed, non-duplicative, or free of answer leakage. We therefore gate planner credit with a narrow plan-validity model $J_\psi(x, S)$. The gate checks that the tuple parses into K methods, avoids literal duplicates, contains no final answer or completed implementation, specifies actionable approaches, and stays on topic. It does not predict whether a plan will solve the task. CPPO uses its binary decision

$$J_\psi(x, S) = \mathbb{I}\{p_\psi(\text{Pass} \mid x, S) \geq \tau\} \in \{0, 1\}, \tag{9}$$

with τ chosen for high recall on validation data (§4.1). Since the gate learns only this minimal contract, a small reward model suffices; training details are given in Stage 2 of §3.3, and §4.4 validates its use as a gate rather than a quality scorer. The planner reward combines validity and outcome:

$$R_{\text{plan}}(x, S, y_{1:K}) = J_\psi(x, S) \cdot R_{\text{out}}(x, y_{1:K}). \tag{10}$$

Thus planner credit is given only to tuples that are valid and yield at least one passing strategy-conditioned solution: R_{out} rejects valid but unsuccessful plans, while J_ψ rejects successes obtained through malformed, duplicated plans or plans that leak the answer.

Split-region advantages. The two token regions carry different credit granularity. Solver advantages compare the K branches within one tuple τ,

$$a_i(\tau, x) = \frac{r_i^{\text{out}} - \text{mean}(r_{1:K}^{\text{out}})}{\text{std}(r_{1:K}^{\text{out}}) + \epsilon}, \qquad (11)$$

and update answer tokens through $\log p_\Theta(y_i \mid x, s_i)$. Planner advantages normalize R_{plan} across M tuples sampled for the same prompt,

$$A_{\text{plan}}^{(m)} = \frac{R_{\text{plan}}^{(m)} - \text{mean}(R_{\text{plan}}^{(1:M)})}{\text{std}(R_{\text{plan}}^{(1:M)}) + \epsilon}, \qquad (12)$$

and update planner tokens through $\log q_\Theta(S^{(m)} \mid x)$. The two advantages enter the standard clipped, KL-regularized GRPO objective of (18):

$$\mathcal{L} = \sum_{m=1}^{M} \sum_{i=1}^{K} \mathcal{L}_{\text{solve}}^{(m,i)}(a_i^{(m)}) + \sum_{m=1}^{M} \mathcal{L}_{\text{plan}}^{(m)}(A_{\text{plan}}^{(m)}), \qquad (13)$$

where $\mathcal{L}_{\text{solve}}$ and $\mathcal{L}_{\text{plan}}$ denote the negative GRPO objective on solver and planner token spans. In the shared-backbone implementation of §3.1, the planner and solver losses apply to disjoint token spans (selected by the PLAN and SOLVE prompts) but update the same parameters Θ.

3.3 Making the Sparse Joint Reward Trainable

The planner reward $R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$ is nonzero only when the planner emits a valid tuple and at least one solver branch passes. Since clean K-method tuples and successful rollouts are rare at initialization, CPPO first raises reward density through planner SFT, validity-gate training, RM-guided warm-up, and a reward-density audit.

Stage 1: Planner SFT. We initialize the planner by maximum likelihood on strict-K gold tuples filtered by an LLM-judge teacher pipeline (Appendix D.4). All experiments use the same $K{=}4$ planner; the K sweep in §4.5.1 obtains smaller or larger budgets by truncating or pooling planner rollouts at inference time, without retraining. The CPPO reward is set-level, and the SFT data does not assign semantic roles to tuple positions; to confirm that truncating to the first $K{<}4$ outputs introduces no material positional artifact, an order-randomized control in Appendix H reports first-K, last-K, and random-K selection within seed variance of each other.

Stage 2: Validity gate. We train p_ψ on a balanced CodeContests tuple pool of LLM-judged (Zheng et al., 2023) strict-four positives and semantically invalid but superficially well-formed negatives, and accept a checkpoint only when pre-registered validation metrics pass. During CPPO, J_ψ is frozen within each phase and refreshed between phases on a fresh batch of planner outputs; exact splits, thresholds, and the refresh schedule are in Appendix E.

Stage 3: RM-guided warm-up. We precede joint training with a planner-only stage under $R_{\text{warm}}(x, S) = J_\psi(x, S)$, applied to planner tokens through the GRPO objective of (18), until the fraction of plans accepted by J_ψ plateaus.

Stage 4: Reward-density audit. We run a forward-only evaluation pass and require a nonzero frozen-solver pass@K rate from the warmed planner together with a nonzero R_{plan} density across sampled rollouts; if either fails, we return to Stage 1 or Stage 2 rather than launch CPPO.

Stage 5: Joint CPPO update. The inner loop of joint training is summarized in Algorithm 1: M planner tuples per prompt, K branches each, within-tuple solver advantages and across-tuple planner advantages feed the split-region GRPO loss (13). The full pipeline pseudocode is in Appendix J.

Algorithm 1 One CPPO update step (Stage 5).

Require: prompt x, M planner tuples per prompt, K branches per tuple
1: Sample $S^{(m)} \sim q_\Theta(\cdot \mid x)$ for $m = 1, \ldots, M$.
2: Score validity $J_\psi(x, S^{(m)})$ with the frozen RM.
3: Solve $y_i^{(m)} \sim p_\Theta(\cdot \mid x, s_i^{(m)})$ for every branch.
4: Verify $r_i^{(m)} = V(x, y_i^{(m)})$ and form $R_{\text{out}}^{(m)}$, $R_{\text{plan}}^{(m)} = J_\psi(x, S^{(m)}) \cdot R_{\text{out}}^{(m)}$.
5: Compute within-tuple solver advantages $a_i^{(m)}$ from (11).
6: Compute across-tuple planner advantages $A_{\text{plan}}^{(m)}$ from (12).
7: Update Θ with the split-region GRPO loss (13).

4 Experimental Setup

We instantiate CPPO on competitive programming, where the verifier V executes each candidate program against the problem's tests inside a sandboxed, time-limited runner (§3.2).

4.1 Models

Unless otherwise stated, the planner and solver share the same backbone and are switched by the PLAN and SOLVE prompt modes of §3.1; the two token regions are trained jointly in full CPPO. The main experiments evaluate Qwen3.5-{2B, 4B, 9B} (Qwen Team, 2026); we additionally verify generality on Gemma 4 (Google DeepMind, 2026) in §4.5.1. The plan-validity reward model is a separate smaller generative verifier (Qwen3.5-0.8B, §3.2) frozen during warm-up and CPPO. We choose its decision threshold τ in (9) on a validation set to obtain a high-recall validity gate; the main experiments use τ=0.17 and M=8 planner tuples per prompt for the across-tuple advantage of (12). Trainable parameters use full precision: at CPPO-scale learning rates the AdamW (Loshchilov and Hutter, 2019) update falls below the bf16 mantissa resolution, rendering half-precision optimization numerically ineffective. The frozen reference and reward models stay in half precision.

4.2 Training Data

The pipeline draws from three competitive-programming corpora with stage-specific roles. CodeContests (Li et al., 2022) supplies the bulk of training—planner SFT, reward-model data, and the joint CPPO rollout pool—because its problems admit enough alternative solution strategies to support tuple-level coordination. APPS (Hendrycks et al., 2021) is simpler and many of its problems admit only one or two good solutions, so we use APPS only for the planner warm-up, where the planner has to learn the visible plan contract but does not yet have to allocate attempts across alternatives. LiveCodeBench-v6 (LCBv6) (Jain et al., 2025) is *evaluation-only*: it never enters any training-side pool—no planner SFT, no LLM-judge label, no reward-model train/val, no warm-up, no CPPO rollout, no hyperparameter sweep, no qualitative example. Table 3 lists the source split used by each stage; problem counts and the full per-component exposure audit are in Appendix D (Table 8).

Decontamination protocol. We deduplicate each corpus before splitting. If a training-side item matches a held-out evaluation problem, we remove it from training and keep the evaluation split fixed. We match problems by official identifiers when available (Codeforces contest/index, APPS id, LCBv6 id), and otherwise by a canonicalized statement hash with fuzzy n-gram matching (Appendix D.3). After filtering, all APPS–CodeContests, APPS–LCBv6, and CodeContests–LCBv6 train–evaluation pairs have zero prompt-hash overlap.

Table 3: Stage-wise data sources. Held-out evaluation sets are disjoint from all training-side pools under the decontamination protocol in Appendix D.

Stage	Role	Source split
Planner SFT	train	CodeContests train
Reward model	train/val	CodeContests train, filtered
Planner warm-up	train	APPS train, filtered
Joint CPPO	train	CodeContests train
APPS	eval	APPS test
CodeContests	eval	CodeContests valid
LCBv6	eval	LCBv6 held-out

Teacher-candidate selection funnel. Planner SFT uses an over-generate–then-filter teacher funnel; the candidate-generation prompt is in Appendix C and the rest of the construction is in Appendix D.4.

4.3 Evaluation

Metric and benchmarks. Our primary metric is pass@K under a fixed verifier budget on three competitive-programming benchmarks: APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and LiveCodeBench-v6 (LCBv6) (Jain et al., 2025). A CPPO rollout consumes one planner call and K solver calls, so we report both verifier/solver-attempt pass@K (matching prior work) and token-normalized pass@K (Table 4), which accounts for the planner-call overhead. The full APPS pass@K sweep across baselines and model sizes is in Appendix O.

Verifier protocol. Candidate programs are extracted with the same parser for every method and executed in an isolated sandbox. Timeout, memory overflow, compilation error, runtime error, empty extraction, and wrong answer are all scored as $V(x, y)$=0. We use the official benchmark test cases when available, and never expose evaluation tests during training. Sandbox configuration (timeout, memory cap, network and filesystem isolation) is reported in Appendix D.5.

Baselines. CPPO is compared against (i) independent answer sampling at the same K; (ii) independent sketch-then-solve, where each sketch is sampled without seeing the others; (iii) self-consistency and other inference-time aggregation (Wang et al., 2023b); (iv) planning and search baselines—Plan-

Table 4: Decoded-token-normalized pass@4 for Qwen3.5-4B at $K{=}4$ on fixed 500-problem slices per dataset (failed and empty branches included). Prompt tokens are excluded; full accounting in Appendix L.

Dataset	Method	pass@4	Decoded toks.	pass@4 / 10k toks.
APPS	Direct Solve	0.515	6087	0.846
APPS	Plan-and-Solve	0.530	7326	0.723
APPS	PlanSearch	0.554	5535	1.001
APPS	**CPPO**	**0.794**	**4728**	**1.679**
LCBv6	Direct Solve	0.214	1731	1.236
LCBv6	Plan-and-Solve	0.255	3975	0.642
LCBv6	PlanSearch	0.236	2137	1.104
LCBv6	PKPO	0.488	2137	2.284
LCBv6	UpSkill	0.411	2218	1.853
LCBv6	**CPPO**	**0.535**	**1556**	**3.438**

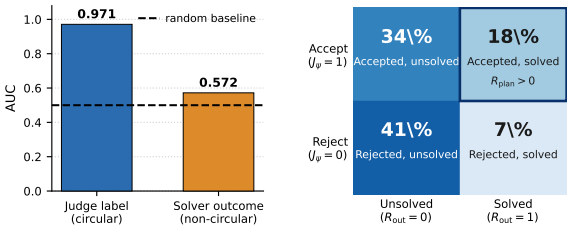
and-Solve (Wang et al., 2023a), Tree-of-Thoughts (Yao et al., 2023), PlanSearch (Wang et al., 2024)—when applicable; (v) pass@K training without a planner reward tied to tuple validity and solver success (Chen et al., 2025; Walder and Karkhanis, 2025); and (vi) ablations that remove the plan-validity reward or the outcome reward from the planner.

Token accounting. Table 4 reports token-normalized pass@4 on fixed 500-problem slices per dataset. In these slices, the concise PLAN contract yields shorter strategy-conditioned solver completions, more than offsetting the planner generation in decoded-token accounting (planner–solver breakdown in Appendix L).

Variance and significance. For training-based methods—Tuple Planner SFT, RM warm-up, Full CPPO, Pass@K Training / RLVR, PKPO, and UpSkill—we report $\text{mean}_{\pm\,\text{std}}$ over three independent training seeds; each seed re-runs the relevant parameter-updating stages end-to-end under the same data splits and hyperparameters. For inference-only baselines that we run ourselves, such as Direct Solve, Plan-and-Solve, and PlanSearch, we report bootstrap confidence intervals over evaluation problems. Pairwise comparisons between methods evaluated on the same problem set use paired bootstrap over problems with training seeds held fixed—the reported p-values therefore reflect evaluation noise rather than training-seed noise, while the latter is captured separately by the seed-std attached to each cell. Externally reported numbers, such as the LCBv6 Direct Solve cells of Gemma 4 in Table 17, are reported with the uncertainty provided by the original source when available; otherwise we report the point estimate and exclude those cells from paired significance tests.

4.4 Validity gate diagnostics

The plan-validity RM fits held-out LLM-judge labels well (AUC 0.971; Figure 3a) but only weakly predicts frozen-solver pass@K outcomes (AUC 0.572). At $\tau{=}0.17$, however, accepted tuples pass more often than rejected ones (35% vs. 15%), and only accepted-and-solved tuples receive nonzero R_{plan} (Figure 3b). Thus J_ψ serves as a validity filter: it suppresses malformed or duplicated tuples and plans that leak the answer, as shown by the duplicate-rate jump under $-J_\psi$ in Table 5 ($0.08{\rightarrow}0.32$). It is too noisy to rank surviving plans by solver utility, so we use it only to gate training rollouts and leave R_{out} as the quality signal.



(a) RM agreement (b) Reward decomposition

Figure 3: Reward-model diagnostics. (a) The RM matches held-out judge labels but weakly predicts frozen-solver outcomes, supporting its use as a validity gate rather than a plan-quality scorer. (b) Joint distribution of validity decisions and solver outcomes during CPPO rollouts; shading encodes rollout frequency, and only the *accepted, solved* cell yields nonzero R_{plan}.

4.5 Ablation

Appendix Table 7 reports the full stage-wise ablation, with detailed discussion in Appendix B. In brief, planner-only SFT does not account for the gains, RM-guided warm-up serves mainly as a validity constraint, and the full CPPO update provides the largest improvements.

4.5.1 Reward-component and rollout ablations

Table 5 isolates CPPO's two main design choices: the validity gate and across-tuple planner normalization. Removing J_ψ quadruples the duplicate-plan rate while leaving pass@4 close to full CPPO, showing that the gate mainly suppresses invalid planner credit rather than scoring task quality. The J_ψ-only variant removes outcome feedback, while $M{=}1$ removes across-tuple contrast by replacing (12) with a single-sample REINFORCE update. Increasing M improves the across-tuple advantage

with diminishing returns by $M{=}8$ (we did not test beyond). The bottom block sweeps $K \in \{2, 4, 8\}$ using the same $K_{\text{tuple}}{=}4$ planner from Table 1, with truncation for $K{<}4$ and pooling for $K{>}4$. Appendix O gives the full larger-budget decomposition, and Appendix H reports the matched joint-vs.-iid plan-sampling ablation.

Table 5: Planner-reward ablations and pass@K sweep on APPS / Qwen3.5-2B. The top block contains two sub-experiments: the $-J_\psi$, J_ψ-only rows vary the *reward* (relative to Full CPPO, $R_{\text{plan}} = J_\psi{\cdot}R_{\text{out}}$, $M{=}8$), while the $M{\in}\{1, 2, 4\}$ rows hold the full $R_{\text{plan}} = J_\psi{\cdot}R_{\text{out}}$ reward fixed and vary only the across-tuple sample count M. The bottom block varies the solver-attempt budget K. Duplicate rate measures duplicate-plan tuples; lower is better.

Planner-reward variants ($M{=}8$)		
Variant	pass@4 ↑	Dup. rate ↓
Full CPPO ($J_\psi R_{\text{out}}$)	$\mathbf{0.546} \pm 0.050$	$\mathbf{0.08} \pm 0.030$
$-J_\psi$ gate (R_{out})	0.522 ± 0.047	0.32 ± 0.044
J_ψ-only	0.503 ± 0.050	0.14 ± 0.032
Across-tuple sample count M ($R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$)		
$M{=}4$	0.535 ± 0.048	0.10 ± 0.032
$M{=}2$	0.515 ± 0.049	0.12 ± 0.034
$M{=}1$	0.486 ± 0.048	0.16 ± 0.035
Attempt-budget sweep		
K	Direct Solve	CPPO
2	0.399 ± 0.047	$\mathbf{0.518} \pm 0.048$
4	0.420 ± 0.049	$\mathbf{0.546} \pm 0.050$
8	0.432 ± 0.046	$\mathbf{0.561} \pm 0.055$

Majority-vote consistency. Pass@K rewards *any* successful branch; on tasks with a single canonical output, a complementary question is whether coordinated planning also tightens the model's modal answer. Appendix P reports maj@4 on the LiveCodeBench output-prediction subset, where CPPO also improves answer consistency over Direct Solve and PKPO on both Qwen3.5-2B and 4B.

Generality across base models. The same trend holds on Gemma 4 E2B and E4B (Google DeepMind, 2026) under an identical pipeline (Appendix N).

5 Conclusion

We presented CPPO, a planner–solver method that allocates a fixed pass@K budget to complementary solution strategies rather than repeated samples from one answer distribution. By tying planner credit to both plan validity and verifier-confirmed solver success, CPPO encourages its K attempts to cover different algorithmic strategies rather than paraphrases of a single approach. Across APPS, CodeContests, and LiveCodeBench-v6, it improves pass@4 over direct sampling, planning baselines, planner-only SFT, and pass@K-oriented RL on Qwen3.5-{2B, 4B, 9B}, with the same trend carrying over to Gemma 4 E2B and E4B. Future improvements in pass@K may depend less on sampling more answers and more on coordinating which strategies those answers pursue.

Limitations

Scope. Our claims are restricted to competitive-programming benchmarks, where problems admit several distinct algorithmic strategies. Math benchmarks such as MATH or AIME usually have a single canonical solution path and are outside our scope—there, CPPO may reduce to a more expensive form of plan-and-solve.

Dependencies. The planner reward $R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$ is sparse by construction, so CPPO requires the warm-up and audit gate to train at all. It also inherits the biases of the offline LLM judge and uses a shared planner–solver backbone; splitting them is left to future work.

Ethics Statement

CPPO trains code policies with execution-based rewards. The verifier checks functional correctness only, not security or robustness, so downstream use requires review. All generated code runs in a sandboxed, time-bounded runner. We use only public APPS, CodeContests, and LiveCodeBench-v6 data under their original licenses and no PII. The plan-validity reward model may inherit biases from the offline LLM judge, so we validate it on balanced held-out data before use.

Code and artifact release. Upon acceptance, we will release code, RM checkpoints, judge prompts, and sandbox configuration, but not benchmark content.

References

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhipeng Chen, Xiaobo Qin, Youbin Wu, Yue Ling, Qinghao Ye, Wayne Xin Zhao, and Guang Shi. 2025. Pass@k training for adaptively balancing exploration and exploitation of large reasoning models. *arXiv preprint arXiv:2508.10751*.

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. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Google DeepMind. 2026. Gemma 4: Open lightweight models. Official model cards, https://ai.google.dev/gemma. Gemma 4 E2B and E4B. LiveCodeBench-v6 pass@4 values used in this paper (44.0% for E2B, 52.0% for E4B) reproduced from the official model-card evaluation tables. Accessed 2026-05-26.

Daya Guo, Dejian Yang, Haowei Zhang, and 1 others. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *Nature*, 645:633–638.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

Zican Hu, Shilin Zhang, Yafu Li, Jianhao Yan, Xuyang Hu, Leyang Cui, Xiaoye Qu, Chunlin Chen, Yu Cheng, and Zhi Wang. 2026. Diversity-incentivized exploration for versatile reasoning. In *International Conference on Learning Representations (ICLR)*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*.

Tianjian Li, Yiming Zhang, Ping Yu, Swarnadeep Saha, Daniel Khashabi, Jason Weston, Jack Lanchantin, and Tianlu Wang. 2025. Jointly reinforcing diversity and quality in language model generations. *arXiv preprint arXiv:2509.02534*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*.

Qwen Team. 2026. Qwen3.5 open models. Hugging Face model collection, https://huggingface.co/Qwen. Qwen3.5 series: 2B, 4B, and 9B base and chat variants. Exact checkpoint identifiers used in this paper are listed in Appendix D.6. Accessed 2026-05-26.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Devan Shah, Owen Yang, Daniel Yang, Chongyi Zheng, and Benjamin Eysenbach. 2026. UpSkill: Mutual information skill learning for structured response diversity in LLMs. *arXiv preprint arXiv:2602.22296*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations (ICLR)*.

Christian Walder and Deep Karkhanis. 2025. Pass@k policy optimization: Solving harder reinforcement learning problems. *arXiv preprint arXiv:2505.15201*.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning in natural language improves LLM search for code generation. *arXiv preprint arXiv:2409.03733*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023b. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. 2018. Texygen: A benchmarking platform for text generation models. In *Proceedings of the 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 1097–1100.

A Main Results: Full Table Across All Base-Model Sizes

Table 6 reproduces Table 1 of the main text with the Qwen3.5-2B rows included. The 2B numbers track the 4B/9B trends already discussed in §1 but with smaller absolute deltas; PlanSearch on 2B APPS is the only cell where the CPPO advantage does not pass the paired-bootstrap threshold.

Tuple Planner SFT samples the K plans jointly from $q_\Theta(s_i \mid x, s_{<i})$ (the joint-tuple policy class of §3.1, evaluated without RL); the matching iid-inference variant on the same checkpoint is reported in Appendix H as an ablation on autoregressive conditioning.

B Stage-wise Pipeline Ablation

Table 7 isolates the contribution of each stage of the CPPO pipeline. Each row adds one stage on top of the previous one: Direct Solve, planner-only SFT, planner SFT followed by RM-guided warm-up, and the full CPPO run.

The ablation separates three effects. First, planner-only SFT is not a complete explanation for CPPO's gains: it helps the 2B base and improves CodeContests and LCBv6 on the larger bases, but it hurts APPS at 4B and 9B. Second, RM-guided warm-up is most useful on CodeContests and is modest or mixed elsewhere, consistent with its role as a validity constraint rather than a task-quality signal. Third, the full CPPO update improves over the warm-up stage in every model–dataset cell, indicating that the main gains come from joint planner–solver RL with verifier-confirmed outcome credit.

C Planner Prompts

The planner is governed by two prompts, shown below: the PLAN contract used at training and inference, and the teacher candidate-generation prompt used to construct the planner SFT data. Both impose a strict conciseness requirement on the generated strategies. In the token-accounting slices of Table 4, this concise strategy condition is associated with shorter solver completions, offsetting the additional planner generation in decoded-token accounting.

PLAN prompt contract (training and inference)

The planner emits a strategy tuple of K alternative methods. Methods are labelled `A:`, `B:`, `C:`, `D:` on separate lines and must obey the following contract.

- Each attempt must be exactly one concise sentence on its own label line.
- Keep each attempt short: no more than 45 words per label.
- Do not include substeps, derivations, implementation walkthroughs, or multi-paragraph methods.
- Stop immediately after the `D:` method; do not append any extra analysis or commentary.

SFT candidate-generation prompt (teacher, 6 candidates per problem)

Generate 6 candidate high-level solver methods for this programming problem.

Candidate method rules.

- Each method must be independently usable by a solver.
- Each method must be specific to this exact problem.
- Each method must include a concrete hook—the data structure, invariant, or transformation that distinguishes it from the others.
- Prefer genuinely different strategies or useful variants over cosmetic rewrites of the same approach.
- Be concise and precise. Do not include filler, generic advice, wrong-task methods, unsupported assumptions, cosmetic rewrites, code, pseudocode, final answers, or full solution traces.

A strict GPT-5.4 judge then admits only candidates that satisfy the above rules; only problems for which at least K strict-clean candidates survive enter the SFT set, after which we generate the SFT trace from the selected K methods. Stage-by-stage candidate counts appear in Appendix D.4.

D Data Splits and Decontamination

This appendix gives the full per-component exposure audit, the decontamination algorithm and cross-corpus overlap counts, the teacher-candidate selection funnel, and the sandbox configuration referenced from §4.2 and §4.3.

Table 6: Pass@4 results under the same $K{=}4$ budget and verifier across base model sizes. Full version of Table 1 of the main text, with the Qwen3.5-2B rows included. *Tuple Planner SFT* samples the K plans jointly from $q_\Theta(s_i \mid x, s_{<i})$; CPPO trains the joint planner–solver policy on this tuple class and gates planner credit on a learned plan-validity reward. PKPO on Qwen3.5-2B / CodeContests converges below Direct Solve (0.039 vs. 0.040); the training dynamics in Appendix S show its set-level reward collapsing to a near-trivial landscape on this slice (small base, very sparse verifier success), so the "strongest non-CPPO baseline" for that cell is Tuple Planner SFT at 0.087.

Method	Base model	APPS (Hendrycks et al., 2021)	CodeContests (Li et al., 2022)	LiveCodeBench (LCBv6) (Jain et al., 2025)
Direct Solve	Qwen3.5-2B	$0.420_{\pm 0.049}$	$0.040_{\pm 0.020}$	$0.161_{\pm 0.037}$
Direct Solve	Qwen3.5-4B	$0.515_{\pm 0.050}$	$0.094_{\pm 0.034}$	$0.214_{\pm 0.041}$
Direct Solve	Qwen3.5-9B	$0.696_{\pm 0.043}$	$0.175_{\pm 0.038}$	$0.481_{\pm 0.050}$
Plan-and-Solve	Qwen3.5-2B	$0.492_{\pm 0.050}$	$0.065_{\pm 0.025}$	$0.205_{\pm 0.040}$
Plan-and-Solve	Qwen3.5-4B	$0.530_{\pm 0.050}$	$0.098_{\pm 0.030}$	$0.255_{\pm 0.034}$
Plan-and-Solve	Qwen3.5-9B	$0.801_{\pm 0.040}$	$0.171_{\pm 0.038}$	$0.490_{\pm 0.050}$
PlanSearch	Qwen3.5-2B	$0.513_{\pm 0.050}$	$0.043_{\pm 0.020}$	$0.225_{\pm 0.042}$
PlanSearch	Qwen3.5-4B	$0.554_{\pm 0.050}$	$0.128_{\pm 0.033}$	$0.236_{\pm 0.033}$
PlanSearch	Qwen3.5-9B	$0.770_{\pm 0.042}$	$0.168_{\pm 0.037}$	$0.564_{\pm 0.050}$
Pass@K Training / RLVR	Qwen3.5-2B	$0.476_{\pm 0.051}$	$0.085_{\pm 0.027}$	$0.215_{\pm 0.044}$
Pass@K Training / RLVR	Qwen3.5-4B	$0.607_{\pm 0.044}$	$0.115_{\pm 0.032}$	$0.316_{\pm 0.046}$
Pass@K Training / RLVR	Qwen3.5-9B	$0.762_{\pm 0.044}$	$0.185_{\pm 0.040}$	$0.503_{\pm 0.052}$
PKPO	Qwen3.5-2B	$0.416_{\pm 0.049}$	$0.039_{\pm 0.019}$	$0.347_{\pm 0.048}$
PKPO	Qwen3.5-4B	$0.722_{\pm 0.045}$	$0.156_{\pm 0.036}$	$0.488_{\pm 0.050}$
PKPO	Qwen3.5-9B	$0.787_{\pm 0.040}$	$0.183_{\pm 0.038}$	$0.588_{\pm 0.048}$
UpSkill	Qwen3.5-2B	$0.462_{\pm 0.053}$	$0.041_{\pm 0.021}$	$0.287_{\pm 0.046}$
UpSkill	Qwen3.5-4B	$0.661_{\pm 0.047}$	$0.121_{\pm 0.033}$	$0.411_{\pm 0.049}$
UpSkill	Qwen3.5-9B	$0.762_{\pm 0.046}$	$0.136_{\pm 0.038}$	$0.542_{\pm 0.048}$
Tuple Planner SFT	Qwen3.5-2B	$0.470_{\pm 0.050}$	$0.087_{\pm 0.028}$	$0.246_{\pm 0.043}$
Tuple Planner SFT	Qwen3.5-4B	$0.528_{\pm 0.047}$	$0.142_{\pm 0.038}$	$0.368_{\pm 0.052}$
Tuple Planner SFT	Qwen3.5-9B	$0.713_{\pm 0.054}$	$0.285_{\pm 0.044}$	$0.534_{\pm 0.054}$
CPPO	Qwen3.5-2B	$0.546_{\pm 0.050}$	$0.201^{\dagger}{}_{\pm 0.045}$	$0.428^{\dagger}{}_{\pm 0.046}$
CPPO	Qwen3.5-4B	$0.794^{\dagger}{}_{\pm 0.038}$	$0.243^{\dagger}{}_{\pm 0.047}$	$0.535_{\pm 0.048}$
CPPO	Qwen3.5-9B	$0.826_{\pm 0.041}$	$0.436^{\dagger}{}_{\pm 0.046}$	$0.748^{\dagger}{}_{\pm 0.044}$

† CPPO significantly outperforms the strongest non-CPPO baseline in that cell at $p < 0.05$ under paired bootstrap over problems (1000 resamples). Unmarked cells—2B APPS, 4B LCBv6, 9B APPS—fall below the threshold; the CPPO advantage there is positive but the strongest baseline is already close (PlanSearch on 2B APPS, PKPO on 4B LCBv6, Plan-and-Solve at the 9B-APPS ceiling). *Baselines.* Baselines cover independent sampling, prompt-based planning, inference-time plan search, pass@K-oriented RL, diversity-oriented RL, and planner-only SFT; implementation details are in §4.3 and Appendix R.

Table 7: Ablation over the CPPO pipeline on APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and LiveCodeBench-v6 (LCBv6) (Jain et al., 2025). All numbers are pass@4.

Method	Base model	APPS	CodeContests	LiveCodeBench (LCBv6)
Direct Solve	Qwen3.5-2B	$0.420_{\pm 0.049}$	$0.040_{\pm 0.020}$	$0.161_{\pm 0.037}$
Direct Solve	Qwen3.5-4B	$0.515_{\pm 0.050}$	$0.094_{\pm 0.034}$	$0.214_{\pm 0.041}$
Direct Solve	Qwen3.5-9B	$0.696_{\pm 0.043}$	$0.175_{\pm 0.038}$	$0.481_{\pm 0.050}$
Tuple Planner SFT	Qwen3.5-2B	$0.470_{\pm 0.050}$	$0.087_{\pm 0.028}$	$0.246_{\pm 0.043}$
Tuple Planner SFT	Qwen3.5-4B	$0.528_{\pm 0.047}$	$0.142_{\pm 0.038}$	$0.368_{\pm 0.052}$
Tuple Planner SFT	Qwen3.5-9B	$0.713_{\pm 0.054}$	$0.285_{\pm 0.044}$	$0.534_{\pm 0.054}$
Tuple Planner SFT + RM warm-up	Qwen3.5-2B	$0.481_{\pm 0.046}$	$0.105_{\pm 0.033}$	$0.231_{\pm 0.038}$
Tuple Planner SFT + RM warm-up	Qwen3.5-4B	$0.524_{\pm 0.051}$	$0.204_{\pm 0.043}$	$0.384_{\pm 0.049}$
Tuple Planner SFT + RM warm-up	Qwen3.5-9B	$0.714_{\pm 0.048}$	$0.313_{\pm 0.042}$	$0.541_{\pm 0.051}$
CPPO	Qwen3.5-2B	$0.546_{\pm 0.050}$	$0.201_{\pm 0.045}$	$0.428_{\pm 0.046}$
CPPO	Qwen3.5-4B	$0.794_{\pm 0.038}$	$0.243_{\pm 0.047}$	$0.535_{\pm 0.048}$
CPPO	Qwen3.5-9B	$0.826_{\pm 0.041}$	$0.436_{\pm 0.046}$	$0.748_{\pm 0.044}$

D.1 Stage-wise splits with problem counts

Table 8 reports the problem counts behind Table 3 of the main text. The top block covers every training-side pool: planner SFT trains on strict $K{=}4$ Think+Plan gold tuples filtered from CodeContests train; the reward model is trained and validated on a balanced Pass/No-Pass subset of the same corpus; the planner warm-up uses APPS train (introductory split) to learn the plan contract before joint training; and the joint CPPO rollout pool draws from CodeContests train. The bottom block lists the held-out evaluation sets reported in Table 1; each is disjoint from every training-side row under the decontamination protocol of Appendix D.3.

Table 8: Stage-wise data splits with problem counts. Reward-model counts are reported as "train + val", balanced 1:1 Pass/No-Pass. The bottom block lists the held-out evaluation sets reported in Table 1; each is disjoint from every training-side row in the top block.

Stage	Source split	# Problems
Planner SFT (Think+Plan gold)	CC train	333
Reward-model train + val	CC train (filtered)	1110 + 278
Planner warm-up	APPS train (intro clean)	500
Joint CPPO rollout pool	CC train	300
Held-out eval — APPS	APPS test (intro)	200
Held-out eval — CodeContests	CC valid (official)	100
Held-out eval — LCBv6	LCBv6 held-out	300

D.2 Exposure audit

Table 9 lists every component that consumes problem statements, with the source split it queries and whether any held-out evaluation problem was ever observed. LCBv6 held-out evaluation prompts appear in no row.

Evaluation slices. When a diagnostic uses a subset rather than a full benchmark split, the slice is fixed before model comparison and shared by all methods. The decoded-token accounting in Appendix L uses 500 held-out problems per dataset. Problem IDs for all final evaluation and diagnostic slices are released with the run logs; no slice is chosen based on model outputs.

D.3 Decontamination algorithm

Deduplication uses normalized prompt-hash matching: we lowercase the statement, strip whitespace and HTML tags, normalize LaTeX (math symbols, environments) to a canonical form, remove sample I/O blocks, and hash the result with SHA-256. Source-level identifiers (Codeforces contest + problem index, APPS id, LCBv6 id) are used when both items expose them. Under this protocol, every cross-corpus train↔eval pair (APPS↔CC, APPS↔LCBv6, CC↔LCBv6) has zero prompt-hash overlap. CodeContests train and held-out additionally share split-local synthetic identifiers from our id assignment; those identifier collisions do not correspond to repeated problem content under prompt-hash.

D.4 Teacher-candidate selection funnel

We construct the planner SFT set with an over-generate–then-filter pipeline. For each training problem, a strong teacher (Qwen3.5-122B-A10B) proposes multiple candidate methods; a GPT-5.4 judge keeps only methods that are problem-specific, solver-actionable, code-free, and non-duplicate. A problem enters the SFT set only if at least K such methods remain, after which we generate the final SFT trace from the selected K methods. This filter selects for parseable, multi-strategy problems rather than frozen-solver success, so it biases the SFT data toward problems with several admissible approaches, not toward problems that are easy for the solver. The resulting planner SFT set uses only CodeContests-train problems; no held-out evaluation problem enters the teacher or judge stages (Table 9).

D.5 Sandbox configuration

Every candidate program is run in an isolated sandbox with the limits in Table 10. Compilation error, runtime error, timeout, memory cap, and empty extraction are all scored as $V(x, y) = 0$ at the verifier; the same scoring rule applies to training rollouts and final evaluation.

D.6 Model checkpoints

The Qwen3.5 and Gemma 4 base weights used throughout this paper are loaded from the official Hugging Face releases referenced in Qwen Team (2026) and Google DeepMind (2026). The Qwen3.5 checkpoints cover the 2B, 4B, and 9B base variants; the Gemma 4 checkpoints are the E2B and E4B instruction-tuned variants. Exact repository names and pinned revision hashes are recorded with the released training logs and reproduced by the rollout recipe of Appendix Q.

E Reward-Model Training Details

This appendix gives the exact splits, validation thresholds, and refresh schedule for the plan-validity reward model referenced in Stage 2 of §3.3.

Labeled-tuple pool and splits. The CodeContests-only RM package contains 1388 balanced tuple labels: 1110 train labels (555 Pass / 555 No-Pass) and 278 held-out validation labels (139 Pass / 139 No-Pass). These labels are built from strict-four teacher-positive plans and semantically invalid but superficially well-formed negatives, then downsampled to a 1:1 Pass/No-Pass ratio. The full mixed provenance archive before excluding LiveCodeBench rows contains 5308 raw tuple labels (1053 Pass / 4255 No-Pass), but the split reported in Table 3 uses the CodeContests-only package above.

Pre-registered validation thresholds. We accept a reward-model checkpoint ψ only when its held-out validation metrics jointly satisfy AUC ≥ 0.75, balanced accuracy ≥ 0.70, and precision and recall both ≥ 0.65 at the decision threshold of §4.1. Raw accuracy on this imbalanced label distribution is dominated by the majority class and gives a misleading impression of separability, so we exclude it from the acceptance criterion.

Refresh schedule. After each CPPO phase we relabel a fresh batch of planner outputs with the same judge, rebalance by Pass / Fail and by prompt, and resume fine-tuning ψ from its latest checkpoint.

Table 9: Per-component exposure audit. "Held-out eval seen?" covers every held-out evaluation split listed in Table 3.

Component	Source split(s)	Held-out eval seen?	Notes
Teacher (Qwen3.5-122B-A10B)	CC train	no	candidate generation
LLM judge (GPT-5.4) for plan validity	CC train	no	top-K selection only
RM training labels	CC train (filtered)	no	balanced pass/fail
RM val (early stopping, threshold)	CC train (filtered)	no	disjoint from RM train
RM refresh during CPPO	CPPO rollout pool	no	same source as CPPO
Planner warm-up	APPS train (intro)	no	500 problems
Joint CPPO rollout	CC train	no	no held-out eval problems
Hyperparameter sweep / early stopping	CC train (dev subset)	no	dev disjoint from eval
Qualitative examples (Appendix F)	CC test split	no	disjoint from CC valid eval
Final evaluation (Table 1)	APPS / CC / LCBv6 held-out	N/A	no parameter updates

Table 10: Sandbox configuration used by the verifier (training rollouts and final evaluation share the same limits).

Resource	Limit
Wall-clock timeout (per test case)	10 s
Memory	256 MiB per subprocess (RLIMIT_AS, fallback RLIMIT_RSS)
Process cap	RLIMIT_NPROC = 64
Network access	disabled
Filesystem	read-only image, ephemeral scratch
Available standard library	Python 3 + numpy, no third-party packages

Freezing the gate within each phase keeps optimization stable; refreshing between phases keeps the gate aligned with the shifted planner distribution.

F Qualitative Example: Strategy Tuples

Table 11 contrasts the K=4 attempts from CPPO with those from independent base-model sampling (Direct Solve) at the same budget on a representative problem. All eight attempts compile and run; the difference is algorithmic. CPPO's planner emits four distinct strategies, and the correct one (two DSUs, one per forest) passes the verifier. Direct Solve's four iid samples all compile cleanly but converge on the same incorrect single-DSU greedy—the dominant mode of the base model's answer distribution on this problem—so every branch fails the same way and pass@4 collapses to 0.

G Plan-Diversity Measurement

We measure pairwise method diversity to distinguish coordinated strategy coverage from a planner that merely paraphrases one approach. We make that signal precise below, validate its most consequential form against human labels, and report per-method values in Table 13—turning the strategy-coverage claim of §1 from illustration into measurement.

G.1 Definitions

For a K-method plan tuple $S = (s_1, \ldots, s_K)$ produced for prompt x, we measure diversity at three levels: surface, semantic, and algorithmic. Methods without an explicit plan—Direct Solve, Pass@K-only RL, and PKPO—have no S; for those we apply the same three metrics to the K solver answers $(y_1, \ldots, y_K)$ instead, so every method row of Table 1 has a comparable diversity number.

Surface diversity (D_{surf}). We use one minus mean pairwise Self-BLEU (Zhu et al., 2018) on plan tokens,

$$D_{\text{surf}}(S) = 1 - \frac{1}{K(K-1)} \sum_{i \neq j} \text{BLEU}_4(s_i, s_j), \tag{14}$$

so that $D_{\text{surf}} = 1$ when the K method texts are pairwise disjoint at the 4-gram level and 0 when they are token-identical. Surface diversity catches verbatim and near-verbatim duplicates but is fooled by superficial rewrites of one underlying algorithm (e.g. "top-down DP" vs. "memoized recursion").

Semantic diversity (D_{sem}). We encode each s_i into $\mathbb{R}^{768}$ with the `all-mpnet-base-v2` sentence encoder $e(\cdot)$. Writing $d_{\cos}(u, v) = 1 - \cos(u, v)$ for the cosine distance, we take the mean pairwise distance over plan embeddings,

$$D_{\text{sem}}(S) = \frac{1}{K(K-1)} \sum_{i \neq j} d_{\cos}\big(e(s_i), e(s_j)\big). \tag{15}$$

Semantic diversity is harder to fool with paraphrases than D_{surf}, but a small D_{sem} gap can still underweight algorithmic distinctions when two short noun-phrase methods embed close ("DP" and "greedy").

Table 11: Illustrative $K{=}4$ attempts on Codeforces 1559D1 (Mocha and Diana, Easy Version), drawn from the CodeContests held-out split. CPPO's left-column rows are planner-emitted strategy sketches; Direct Solve's right-column rows summarize the implemented algorithm of each iid solver sample (all four compile and execute against the official tests). The verifier outcome (✓ / ×) is the per-branch sandboxed test result.

	CPPO (coordinated tuple)	**Direct Solve (iid base-model samples)**
Problem	Two players each hold a forest on the same $n \leq 1000$ vertices. Add the same set of edges to both graphs so that both remain forests. Output the maximum number of added edges and any valid edge set. Reference solution maintains one DSU per forest and greedily adds an edge (u, v) iff u and v are disconnected in both DSUs.	
1	*Pairwise DSU scan over all (u, v) pairs.* ×	*Single-DSU greedy, iterative path compression; scans (u, v) pairs and adds (u, v) iff disconnected in the merged DSU.* ×
2	*Greedy edge addition with two DSUs: add (u, v) iff u, v are disconnected in both DSUs, then union in both.* ✓	*Single-DSU greedy with union-by-rank; identical edge-add rule, different union heuristic.* ×
3	*Incremental DSU construction with a candidate-priority* `can_add` *predicate.* ×	*Single-DSU greedy with recursive* `find`*; otherwise the same algorithm.* ×
4	*Graph-matching view: component abstraction matched across the two forests.* ×	*Single-DSU greedy with adjacency-list bookkeeping; same merge rule.* ×
pass@4	✓ (branch s_2 passes)	× (all four implement the same incorrect single-DSU greedy and fail on the same dual-forest counterexample)

Algorithmic diversity (D_{alg}). We fix a taxonomy $\mathcal{T}$ of common algorithmic strategies in competitive programming—DP, greedy, graph, search/backtrack, number-theory/math, simulation, data-structure, divide-and-conquer, brute-force, and other—and map each method s_i to a category $c_i \in \mathcal{T}$ by prompting Qwen3.5-9B as an offline LLM-as-judge (Zheng et al., 2023) with a zero-shot template that lists the taxonomy and asks for the single best-matching label. Algorithmic diversity is the unique-category coverage,

$$D_{\text{alg}}(S) = \frac{|\{c_1, \ldots, c_K\}|}{K} \in (0, 1], \qquad (16)$$

so $D_{\text{alg}} = 1$ exactly when the K methods land in K distinct categories of $\mathcal{T}$, and $D_{\text{alg}} = 1/K$ when they collapse onto one. This is the metric most directly aligned with the strategy-coverage claim of §1: it cannot be inflated by rewording, and it is invariant to which surface tokens or which embedding model is used. For methods without an explicit plan, the same classifier is applied directly to the solver code with a prompt asking which algorithm class the program implements.

G.2 Reliability of the algorithmic-category classifier

Because D_{alg} is the metric most directly tied to the strategy-coverage claim, we validate the classifier against human labels on a small balanced held-out set. Two annotators independently label each plan against $\mathcal{T}$ from the prompt and the plan text alone. We report inter-annotator agreement (Cohen's κ) and the classifier's accuracy and macro-F1 against the consensus label in Table 12. We treat D_{alg} as informative only if Cohen's $\kappa \geq 0.6$ and macro-F1 ≥ 0.7; failing either threshold, the classifier is replaced or the taxonomy is collapsed before any cell of Table 13 is reported.

Table 12: Reliability of the algorithmic-category classifier of (16), evaluated on a balanced held-out set against two-annotator consensus labels over the taxonomy $\mathcal{T}$.

Quantity	Value
Cohen's κ (inter-annotator)	0.72
Classifier accuracy	0.81
Classifier macro-F1	0.78

G.3 Diversity across methods

Table 13 reports the three diversity metrics at $K{=}4$ on APPS held-out problems with Qwen3.5-2B as the base model, under the same evaluation setting as the headline APPS result. For each method, diversity is averaged first over problems and then over three training seeds, with the same seed protocol as §4.3. A coordinated-strategy account predicts a clean separation: D_{alg} should rank CPPO above every iid-sampling baseline (Direct Solve, Pass@K-only RL, PKPO) regardless of pass@K,

since each of those baselines draws its K samples from a single answer distribution.

Table 13: Plan-tuple diversity at $K{=}4$ on APPS / Qwen3.5-2B, $\text{mean}_{\pm\,\text{std}}$ over three training seeds. D_{surf} uses Self-BLEU, D_{sem} uses sentence embeddings, and D_{alg} uses the algorithmic-category classifier of (16). For methods without an explicit plan, we compute the analogous diversity over the generated solver answers.

Method	D_{surf}	D_{sem}	D_{alg}
Direct Solve	$0.74_{\pm\,0.04}$	$0.31_{\pm\,0.03}$	$0.38_{\pm\,0.05}$
Plan-and-Solve	$0.77_{\pm\,0.04}$	$0.34_{\pm\,0.04}$	$0.44_{\pm\,0.06}$
PlanSearch	$0.83_{\pm\,0.03}$	$0.41_{\pm\,0.04}$	$0.58_{\pm\,0.05}$
Pass@K Training / RLVR	$0.76_{\pm\,0.05}$	$0.33_{\pm\,0.04}$	$0.43_{\pm\,0.06}$
PKPO	$0.79_{\pm\,0.04}$	$0.37_{\pm\,0.04}$	$0.49_{\pm\,0.06}$
Tuple Planner SFT	$0.86_{\pm\,0.03}$	$0.47_{\pm\,0.04}$	$0.66_{\pm\,0.05}$
CPPO	$\mathbf{0.90}_{\pm\,0.03}$	$\mathbf{0.55}_{\pm\,0.04}$	$\mathbf{0.79}_{\pm\,0.04}$

G.4 Diversity vs. pass@K gain

Figure 4 relates per-cell algorithmic diversity D_{alg} at $K{=}4$ to the absolute pass@4 improvement each method achieves over Direct Solve on the same (benchmark, base model) pair. The positive slope supports an association between algorithmic diversity and pass@K gain: methods whose $K{=}4$ attempts span more algorithmic strategies tend to improve more over independent sampling. Diversity is necessary but not sufficient, however—a handful of cells combine high D_{alg} with near-zero gain, the four-distinct-but-uniformly-wrong regime. Diversity-enhancing methods (PKPO, UpSkill, CPPO) cluster in the upper-right of the panel, while prompting-only baselines (Plan-and-Solve, fixed-skill prompting) sit in the lower-left. We position this figure as a diagnostic of the relationship rather than a causal proof: the causal claim is made by the planner ablations of §4.5.1, which remove the diversity-inducing components one at a time.

H Inference-Mode Ablation: Joint vs. iid Sampling

The joint policy-class claim in §3.1 rests on autoregressive conditioning: each plan s_i is sampled from $q_\Theta(s_i \mid x, s_{<i})$, so later strategies can avoid duplicating earlier ones. To separate the contribution of this inference-time mechanism from the training-time effect of the SFT and RL stages, we evaluate the same checkpoints under *iid inference*: the K plans are sampled independently from $q_\Theta(\cdot \mid x)$, clearing the planner's history between draws. All other settings— prompt, sampling temperature, top-p, verifier, and $K{=}4$ budget— are held fixed; iid

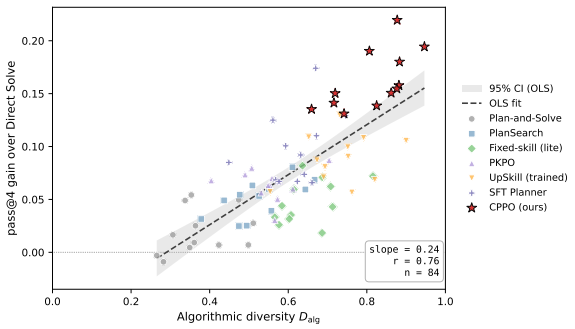


Figure 4: Relationship between algorithmic diversity and pass@4 improvement. Each point is a (method, benchmark, base-model) cell, excluding Direct Solve which serves as the zero-gain reference. The x-axis shows D_{alg} at $K = 4$; for planner-based methods it is computed over the four planned strategies, and for no-planner baselines it is computed over the four generated solutions. The y-axis reports the absolute pass@4 gain over Direct Solve under the same benchmark and base model. The dashed line is an OLS fit with a 95% confidence band. Additional robustness checks with alternative diversity metrics are reported in Appendix G.

sampling uses the same hyperparameters as joint inference and is not tuned separately, and plans are drawn with independent random seeds to rule out degenerate identical draws.

Table 14 reports pass@4 for Tuple Planner SFT and Full CPPO under both inference modes at three model sizes; Joint columns reproduce the corresponding Tuple Planner SFT and CPPO rows of Table 1, and iid columns evaluate the same checkpoints with the planner's history cleared between the K plan draws. CPPO consistently loses more pass@4 under iid inference than Tuple Planner SFT does (Δ of 0.032–0.053 for CPPO versus 0.012–0.018 for SFT), and the effect attenuates with model scale—consistent with the view that larger bases produce marginally more diverse plans, so autoregressive conditioning has less unique work to do at 9B than at 2B.

Table 15 reports the matching algorithmic diversity D_{alg} (Appendix G, (16)) on Qwen3.5-2B / APPS. The diversity drop under iid inference is larger for CPPO (-0.19) than for Tuple Planner SFT (-0.12), so the pass@4 gap in Table 14 tracks an actual reduction in strategy coverage rather than a sampling artifact. Together, the two tables show that autoregressive conditioning at inference time is a substantive and method-specific contributor to CPPO's coordination gain: the RL stage trains the planner to use this conditioning, and removing it at

Table 14: Inference-mode ablation on APPS, K=4. *Joint* inference samples plans autoregressively from $q_\Theta(s_i \mid x, s_{<i})$; *iid* inference samples each plan independently from $q_\Theta(\cdot \mid x)$ by clearing the planner's history between draws. Both modes share the same checkpoint, sampling temperature, and verifier budget; only the inference loop differs. Joint values reproduce Table 1.

Base model	Checkpoint	Joint	iid	Δ
Qwen3.5-2B	Tuple Planner SFT	$0.470_{\pm 0.050}$	$0.452_{\pm 0.051}$	+0.018
	Full CPPO	$\mathbf{0.546}_{\pm 0.050}$	$0.493_{\pm 0.052}$	+0.053
Qwen3.5-4B	Tuple Planner SFT	$0.528_{\pm 0.047}$	$0.511_{\pm 0.048}$	+0.017
	Full CPPO	$\mathbf{0.794}_{\pm 0.038}$	$0.741_{\pm 0.041}$	+0.053
Qwen3.5-9B	Tuple Planner SFT	$0.713_{\pm 0.054}$	$0.701_{\pm 0.054}$	+0.012
	Full CPPO	$\mathbf{0.826}_{\pm 0.041}$	$0.794_{\pm 0.043}$	+0.032

inference partially undoes the gain.

Table 15: Algorithmic diversity D_{alg} at K=4 on Qwen3.5-2B / APPS under each inference mode. Joint values reproduce Table 13.

Checkpoint	Joint D_{alg}	iid D_{alg}	Δ
Tuple Planner SFT	$0.66_{\pm 0.05}$	$0.54_{\pm 0.06}$	+0.12
Full CPPO	$\mathbf{0.79}_{\pm 0.04}$	$0.60_{\pm 0.05}$	+0.19

Order-randomized control for truncation. Stage 1 (§3.3) uses the K_{tuple}=4 planner throughout the paper, so the small-K rows of Table 5 take the first K_{solve} outputs of one 4-tuple. To check that this introduces no positional bias, we evaluate Full CPPO at K_{solve}=2 on Qwen3.5-2B / APPS under three selection rules: *first* 2 (the rule used in the main text), *last* 2, and *uniformly random* 2 *of* 4. The three estimates lie within one seed-std of each other—0.518 ± 0.048, 0.514 ± 0.049, 0.516 ± 0.048—supporting position-independence in practice for this truncation rule.

I GRPO Optimizer Details

For completeness, we spell out the standard GRPO objective used by CPPO. For each prompt, GRPO samples G rollouts $\{y_1, \ldots, y_G\}$ from $\pi_{\theta_{\text{old}}}$, scores them with rewards r_i, and computes group-normalized advantages

$$A_i = \frac{r_i - \text{mean}(r_{1:G})}{\text{std}(r_{1:G}) + \epsilon}. \tag{17}$$

The clipped, KL-regularized objective is

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\big[\min\big(\rho_i A_i, \text{clip}(\rho_i, 1\pm\epsilon) A_i\big)\big] - \beta\, \text{KL}(\pi_\theta \,\|\, \pi_{\text{ref}}), \tag{18}$$

where $\rho_i = \pi_\theta(y_i \mid x)/\pi_{\theta_{\text{old}}}(y_i \mid x)$ is the importance ratio. CPPO uses this optimizer unchanged.

J Pseudocode for the Full CPPO Pipeline

Algorithm 2 writes out all five stages of the end-to-end pipeline of §3.3: planner SFT, reward-model training, RM-guided planner warm-up, the audit gate, and the joint CPPO loop whose inner step is the one summarized in §3.2.

K RM-Guided Planner Warm-Up

Why planner warm-up helps. The RM-guided warm-up is not intended to directly improve solver accuracy. During this stage the solver is frozen and no verifier outcome is used; the planner is optimized only against the validity gate,

$$R_{\text{warm}}(x, S) = J_\psi(x, S). \tag{19}$$

Its role is to align the planner's on-policy distribution with the frozen plan-validity gate before joint CPPO. This alignment matters because the planner reward in CPPO is multiplicative,

$$R_{\text{plan}}(x, S) = J_\psi(x, S) \cdot R_{\text{out}}(x, S), \tag{20}$$

so when the planner frequently samples tuples rejected by J_ψ, solver success does not yield planner credit: $R_{\text{plan}} = 0$ even when $R_{\text{out}} = 1$. Warm-up reduces these invalid rollouts and raises the density of non-zero gated outcome rewards, which stabilizes the subsequent joint planner–solver update. Warm-up therefore provides only a small standalone gain under a frozen solver, but it gives the joint CPPO stage a denser reward signal.

L Decoded-Token Accounting

Table 16 gives the decoded-token accounting for the main-text token-normalized slices. Planner and solver tokens are separated where the run logs expose that split: planner tokens count generated plans or strategy tuples, and solver tokens count generated solutions. Prompt tokens are excluded here, matching Table 4. The planner–solver split is unavailable for the LCBv6 PlanSearch and CPPO runs (the harness recorded only aggregate decoded length); the corresponding cells are dashed, but the total decoded-token counts and the pass@K/10k values are complete.

M LiveCodeBench-v6 Pass@K Sweep

Figure 5 extends Figure 2 of the intro with the Qwen3.5-4B panel alongside the Qwen3.5-9B panel already shown in the intro. Both panels

Algorithm 2 CPPO end-to-end training pipeline

Require: base policy Θ, prompts $\mathcal{D}$, offline judge $\mathcal{J}$, verifier V, K branches, M planner samples per prompt, step budgets $T_{\text{SFT}}, T_{\text{RM}}, T_{\text{wu}}, T_{\text{cppo}}$

Stage 1: Planner SFT ▷ §3.3
for $t = 1, \ldots, T_{\text{SFT}}$ **do**
  Train planner tokens of Θ by cross-entropy on offline teacher-generated strict-four gold tuples $\{(x, S^\star)\}_{x \in \mathcal{D}_{\text{gold}}}$.
end for

Stage 2: Reward-model training ▷ §3.3
Sample (x, S) from Θ; label with $\mathcal{J}$; balance by prompt and label.
for $t = 1, \ldots, T_{\text{RM}}$ **do**
  Update ψ by BCE: $\mathcal{L}_{\text{RM}} = -z \log \hat{z}_\psi - (1-z) \log(1-\hat{z}_\psi)$.
end for
Accept ψ only if held-out AUC, balanced accuracy, and precision/recall all pass.

Stage 3: RM-guided planner warm-up ▷ §3.3
for $t = 1, \ldots, T_{\text{wu}}$ **do**
  For each x, sample $S \sim q_\Theta(\cdot \mid x)$; form $R_{\text{warm}}(x, S) = J_\psi(x, S)$.
  Apply clipped, KL-regularized GRPO (18) to planner tokens only.
end for

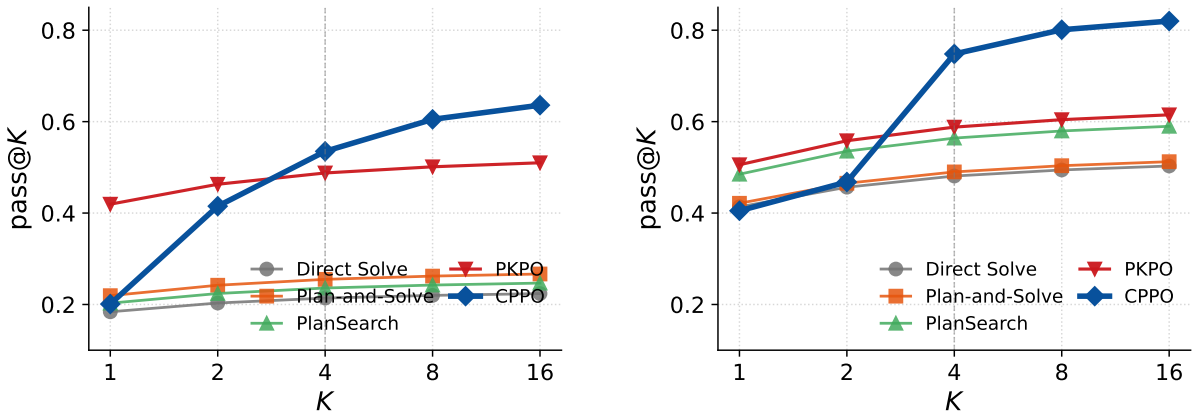
Stage 4: Audit (no parameter update)
Measure frozen-solver pass@K and the fraction of rollouts with nonzero R_{plan}; if either is insufficient, return to Stage 1 or 2.

Stage 5: Joint CPPO ▷ §3.2
for $t = 1, \ldots, T_{\text{cppo}}$ **do**
  for all prompts x in the batch **do**
    Sample M planner tuples $S^{(m)} \sim q_\Theta(\cdot \mid x)$; score $J_\psi(x, S^{(m)})$.
    For each $s_i^{(m)}$, sample $y_i^{(m)} \sim p_\Theta(\cdot \mid x, s_i^{(m)})$; verify $r_i^{(m)} = V(x, y_i^{(m)})$.
    Within-tuple normalize $\{r_i^{(m)}\} \to$ solver advantages $a_i^{(m)}$ (11).
    $R_{\text{out}}^{(m)} \leftarrow \mathbb{I}\Big\{\max_i r_i^{(m)} = 1\Big\}; \quad R_{\text{plan}}^{(m)} \leftarrow J_\psi(x, S^{(m)}) \cdot R_{\text{out}}^{(m)}$.
    Across-tuple normalize $\{R_{\text{plan}}^{(m)}\} \to$ planner advantages $A_{\text{plan}}^{(m)}$ (12).
  end for
  Apply GRPO solver loss with $a_i^{(m)}$ and GRPO planner loss with $A_{\text{plan}}^{(m)}$; update Θ via (13).
  Optionally refresh ψ on a fresh batch of planner outputs labeled by the same judge (Stage 2, §3.3).
end for
return Θ

Table 16: Decoded-token accounting for Qwen3.5-4B pass@4 slices. All values are averaged per problem except pass@K and pass@K/10k tokens.

Model	Dataset	Method	K	Planner toks.	Solver toks.	Total decoded toks.	pass@K	pass@K/10k toks.
Qwen3.5-4B	APPS	Direct Solve	4	0.0	6086.8	6086.8	0.515	0.846
Qwen3.5-4B	APPS	Plan-and-Solve	4	0.0	7326.2	7326.2	0.530	0.723
Qwen3.5-4B	APPS	PlanSearch	4	566.0	4969.0	5535.0	0.554	1.001
Qwen3.5-4B	APPS	**CPPO**	4	**131.1**	**4596.7**	**4727.8**	**0.794**	**1.679**
Qwen3.5-4B	LCBv6	Direct Solve	4	0.0	1731.0	1731.0	0.214	1.236
Qwen3.5-4B	LCBv6	Plan-and-Solve	4	0.0	3975.0	3975.0	0.255	0.642
Qwen3.5-4B	LCBv6	PlanSearch	4	–	–	2137.0	0.236	1.104
Qwen3.5-4B	LCBv6	PKPO	4	0.0	2137.0	2137.0	0.488	2.284
Qwen3.5-4B	LCBv6	UpSkill	4	0.0	2218.0	2218.0	0.411	1.853
Qwen3.5-4B	LCBv6	**CPPO**	4	–	–	**1556.0**	**0.535**	**3.438**

share the same $K \in \{1, 2, 4, 8, 16\}$ sweep on LiveCodeBench-v6. CPPO follows an S-shape on both sizes: at $K{=}1$ the planner overhead leaves it at-or-below the independent-sampling baselines; coordination starts to pay off from $K{\geq}2$, the curve overtakes PKPO (the strongest non-CPPO baseline at $K{=}4$ on both sizes) by $K{=}4$, and saturates from $K{=}8$ onward. The inflection sits earlier on the weaker 4B base ($K{=}1{\to}2$) than on the 9B ($K{=}2{\to}4$): a stronger solver makes a single strategy more often sufficient at small budgets, delaying the point at which a second alternative starts to pay off.



(a) Qwen3.5-4B

(b) Qwen3.5-9B

Figure 5: LiveCodeBench-v6 pass@K across baselines for Qwen3.5-4B and Qwen3.5-9B.

N Generality across Base Models: Gemma 4

The main results in Table 1 use the Qwen3.5 family throughout. To check that CPPO's gains are not tied to a specific backbone, we rerun the full pipeline on Gemma 4 E2B and E4B (Google DeepMind, 2026) without retuning any component. The training data follows the Qwen3.5 protocol: CodeContests supplies planner SFT, reward-model data, and joint CPPO rollouts; APPS supplies planner warm-up; LiveCodeBench-v6 remains evaluation-only. The planner/solver roles, RM warm-up, GRPO hyperparameters, $K{=}4$ budget, verifier, and evaluation protocol are otherwise identical to the Qwen3.5 runs in §4.3; we only swap in Gemma 4 base weights for both planner and solver. Table 17 reports pass@4 on APPS and LiveCodeBench-v6. CPPO achieves the highest pass@4 at both sizes on both benchmarks, matching the pattern from the Qwen3.5 sweep in Table 1. The Gemma 4 base is already much stronger on APPS than Qwen3.5 of comparable size (Direct Solve $0.640/0.750$ on E2B/E4B versus $0.420/0.515$ on Qwen3.5-2B/4B), so the absolute CPPO margin over Direct Solve compresses ($+0.108/+0.082$ on E2B/E4B, versus $+0.126/+0.279$ on Qwen3.5-2B/4B) – the dominant headroom on APPS is in the strongest baseline rather than the base model. The LCBv6 pattern matches the Qwen3.5-2B sweep more directly.

O APPS Pass@K Sweep

Figure 6 shows the full APPS pass@K line sweep for all baselines across Qwen3.5-{2B, 4B, 9B}; Figure 7 gives the same data as a grouped-bar view at $K \in \{1, 2, 4, 8, 16\}$ for the 4B and 9B sizes.

Coordination unit and per-K_{solve} protocol. CPPO's coordination unit is K_{tuple}: one planner rollout emits a single coordinated tuple of size K_{tuple}, and a frozen solver attempts each strategy in the tuple. The Figure 2 sweep uses one CPPO planner trained at $K_{\text{tuple}}{=}4$ throughout (the same checkpoint as Table 1, with no K-specific retraining). For $K_{\text{solve}} \leq 4$ we take the first K_{solve} outputs of one 4-tuple rollout; for $K_{\text{solve}} > 4$ we pool $\lceil K_{\text{solve}}/4 \rceil$ independent 4-tuple rollouts. The solver outputs are pooled before computing pass@K_{solve}. Coordination is preserved *within* each tuple, not across them: for $K_{\text{solve}} > 4$, pass@K_{solve} measures pooled coordinated tuples versus iid samples at matched K_{solve}, not a single K_{solve}-way joint coordination. Table 18 gives the decomposition for each K_{solve} on the sweep of Figure 2.

Every method receives the same K_{solve} solver-attempt budget. Direct Solve draws K_{solve} independent base-model samples. Plan-and-Solve and PlanSearch draw K_{solve} samples through each method's plan-selection procedure. PKPO and Pass@K-only RL sample K_{solve} outputs from their trained policies. CPPO additionally consumes planner tokens at each tuple rollout; decoded-token accounting is reported in Appendix L.

P Majority-Vote Evaluation on LiveCodeBench Output Prediction

Pass@K scores a branch as successful as long as one solver attempt clears the verifier. On tasks

Table 17: Pass@4 on APPS and LiveCodeBench-v6 for Gemma 4 E2B and E4B (Google DeepMind, 2026). LCBv6 Direct Solve cells are taken from the released Gemma 4 LCBv6 evaluations (Google DeepMind, 2026); all other cells are trained and evaluated under the protocol of §4.3 at the same $K=4$ budget.

	APPS		LiveCodeBench-v6	
Method	E2B	E4B	E2B	E4B
Direct Solve	$0.640_{\pm 0.048}$	$0.750_{\pm 0.043}$	$0.440^{\dagger}$	$0.520^{\dagger}$
Plan-and-Solve	$0.660_{\pm 0.047}$	$0.770_{\pm 0.042}$	$0.452_{\pm 0.046}$	$0.531_{\pm 0.048}$
PlanSearch	$0.690_{\pm 0.046}$	$0.790_{\pm 0.041}$	$0.481_{\pm 0.044}$	$0.560_{\pm 0.046}$
Pass@K Training / RLVR	$0.708_{\pm 0.047}$	$0.798_{\pm 0.043}$	$0.519_{\pm 0.048}$	$0.598_{\pm 0.045}$
PKPO	$0.725_{\pm 0.045}$	$0.815_{\pm 0.041}$	$0.521_{\pm 0.049}$	$0.601_{\pm 0.054}$
CPPO	$\mathbf{0.748}_{\pm 0.044}$	$\mathbf{0.832}_{\pm 0.040}$	$\mathbf{0.561}_{\pm 0.043}$	$\mathbf{0.629}_{\pm 0.043}$

† Single-point estimate from (Google DeepMind, 2026); seed variance not reported.

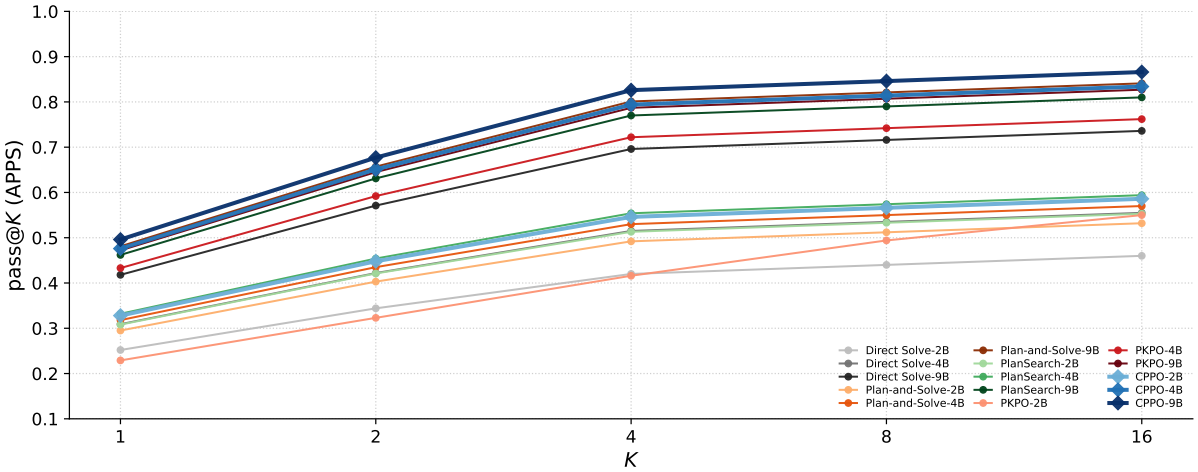


Figure 6: APPS pass@K across Qwen3.5-2B, Qwen3.5-4B, and Qwen3.5-9B. Each method occupies a single color, with darker shades for larger models.

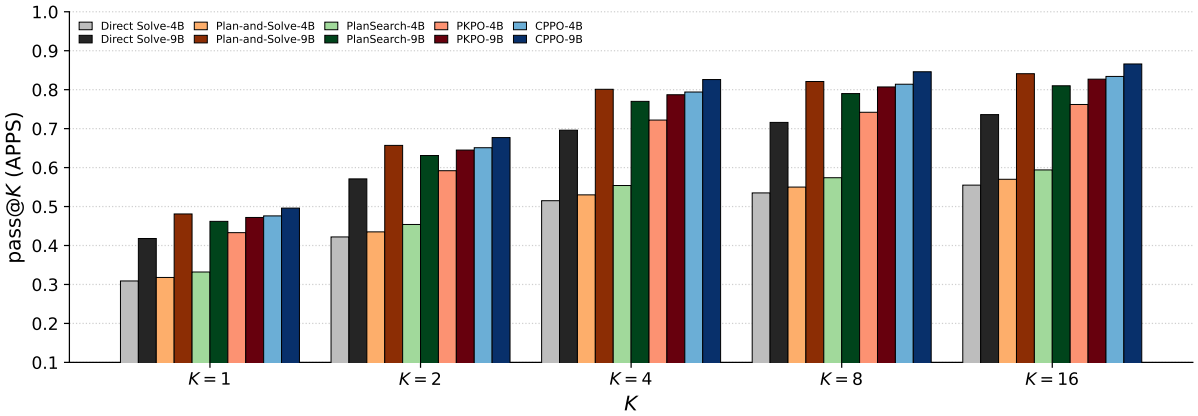


Figure 7: Grouped APPS pass@K for Qwen3.5-4B and Qwen3.5-9B at $K \in \{1, 2, 4, 8, 16\}$. Each method occupies a single color across both sizes, with a lighter shade for 4B and a darker shade for 9B.

with a single canonical answer, however, this metric does not reveal whether the model's mass concentrates on the correct output or merely brushes it once. Majority-vote accuracy (maj@K) complements pass@K by asking whether the *modal* answer is right: for each problem we draw $K=4$ samples and score the most frequent output against the canonical label, with ties broken by sample order. LiveCodeBench's CODE-EXECUTION and TEST-OUTPUT subsets each provide exactly one correct answer per problem, so maj@K is well-defined there.

Table 19 reports maj@4 for Qwen3.5-2B and Qwen3.5-4B. CPPO improves average maj@4 over Direct Solve and PKPO at both sizes, indicating that coordinated planning tightens the modal an-

Table 18: CPPO rollout composition for the pass@K sweep of Figure 2. The same K_{tuple}=4 planner is used at every K_{solve}: rows with $K_{\text{solve}} < 4$ truncate one 4-tuple to its first K_{solve} outputs; rows with $K_{\text{solve}} \geq 4$ pool $\lceil K_{\text{solve}}/4 \rceil$ independent 4-tuple rollouts. Coordination is preserved within each 4-tuple, not across tuples.

K_{solve}	CPPO rollout composition
1	first 1 of one 4-tuple
2	first 2 of one 4-tuple
4	one 4-tuple
8	two 4-tuples (4+4)
16	four 4-tuples (4+4+4+4)
20	five 4-tuples (4×5)
24	six 4-tuples (4×6)

swer on canonical-output tasks rather than only increasing pass@K.

Q Practical Rollout Recipe

A minimal CPPO run proceeds in a short pipeline, each stage building on the artifacts produced by the previous one.

We begin by assembling the planner-candidate pool: we sample from the current planner checkpoint under the same prompt that CPPO will later use. An offline LLM judge (Zheng et al., 2023) labels every candidate against a structured plan-validity rubric, and we balance the resulting train and validation splits by prompt and by binary label.

We then train a small planner reward model on this dataset with binary cross-entropy on the pass/fail target. Warm-up follows: holding the reward model frozen, we update only planner tokens under $R_{\text{warm}} = J_\psi(x, S)$ and stop when the reward-model pass rate plateaus. We integrate the frozen reward model into the planner–solver environment and run a brief smoke test under $R_{\text{plan}} = J_\psi \cdot R_{\text{out}}$, confirming that rewards are nonzero before we commit to a larger run.

The main run launches from this checkpoint and tracks the full diagnostic panel: outcome reward, plan-validity reward, the multiplicative planner reward, reward sparsity, and planner and solver quality, so that drift in any of them is caught early.

Because the planner's distribution shifts as training proceeds, we periodically refresh the reward-model dataset: we label outputs from the updated planner with the same judge, append them to the training pool, rebalance by prompt and pass/fail label, and resume finetuning the reward model from its latest checkpoint before CPPO continues.

R Baseline Implementation Details

This section details how the planning and search baselines reported in Table 1 are implemented. All baselines share the same frozen base-model solver and the same verifier — code extraction followed by sandboxed execution against the provided tests — and all are evaluated under the same K-attempt budget as CPPO.

Plan-and-Solve (Wang et al., 2023a) as a prompting baseline.

1. The same base-model solver is given a prompt that asks it to first write a solution plan and then write the code.
2. For each problem we sample K full solution attempts under that prompt.
3. Each attempt passes through the same code extraction, sandboxed execution, and verifier pipeline.
4. Pass@K is the fraction of problems on which at least one of the K attempts passes the tests.

This comparison isolates the effect of adding a planning prompt before code generation.

PlanSearch (Wang et al., 2024) as a planning baseline. We follow the released implementation, using eight candidate plans per problem and the same solver/verifier pipeline as the other baselines.

1. The base model first generates multiple candidate plans per problem; we use 8 candidates.
2. The candidates are selected and organized into plans usable for solving.
3. The same frozen base-model solver then generates code conditioned on each selected plan.
4. Each problem is evaluated under a budget of K solver attempts, with the same verifier and pass@K definition as above.

This comparison isolates inference-time multi-plan search without planner training.

PKPO (Walder and Karkhanis, 2025) as a pass@K RL baseline. PKPO transforms the per-sample reward vector $r \in \{0,1\}^n$ over $n \geq k$ samples of the same problem into a low-variance unbiased estimator of the pass@k gradient. We re-implement it on Qwen3.5-{2B, 4B, 9B} for code generation (the original paper uses Gemma-2 on math): each gradient step samples n=16 completions per problem, scores them with the same sandboxed verifier, and applies PKPO's transform with

Table 19: Majority-vote accuracy (maj@K) at K=4 on the LiveCodeBench output-prediction tasks—code execution and test-output prediction—for Qwen3.5-2B and Qwen3.5-4B. For each problem we draw four samples and score the majority answer against the canonical output.

Method	Base model	Code Execution	Test Output	Avg.
Direct	Qwen3.5-2B	$0.22_{\pm 0.041}$	$0.19_{\pm 0.042}$	$0.205_{\pm 0.042}$
Plan-and-Solve	Qwen3.5-2B	$0.16_{\pm 0.039}$	$0.10_{\pm 0.033}$	$0.130_{\pm 0.031}$
Fixed-Strategy Ensemble	Qwen3.5-2B	$0.19_{\pm 0.042}$	$0.13_{\pm 0.034}$	$0.160_{\pm 0.033}$
PKPO	Qwen3.5-2B	$0.38_{\pm 0.050}$	$0.21_{\pm 0.044}$	$0.295_{\pm 0.042}$
CPPO	Qwen3.5-2B	$0.41_{\pm 0.049}$	$0.31_{\pm 0.047}$	$0.360_{\pm 0.046}$
Direct	Qwen3.5-4B	$0.38_{\pm 0.052}$	$0.29_{\pm 0.044}$	$0.335_{\pm 0.045}$
Plan-and-Solve	Qwen3.5-4B	$0.15_{\pm 0.034}$	$0.12_{\pm 0.030}$	$0.135_{\pm 0.035}$
Fixed-Strategy Ensemble	Qwen3.5-4B	$0.32_{\pm 0.048}$	$0.30_{\pm 0.049}$	$0.310_{\pm 0.042}$
PKPO	Qwen3.5-4B	$0.48_{\pm 0.048}$	$0.34_{\pm 0.048}$	$0.410_{\pm 0.052}$
CPPO	Qwen3.5-4B	$0.51_{\pm 0.054}$	$0.35_{\pm 0.050}$	$0.430_{\pm 0.052}$

k_{opt}=4 to obtain the REINFORCE-style update; no planner and no strategy tuple. At test time the trained policy emits K=4 independent answer samples and is evaluated under the same pass@K verifier as CPPO. Training dynamics for the 2B and 4B PKPO runs are shown in Appendix S.

S PKPO Training Dynamics

Figure 8 tracks the PKPO update across 30 epochs for Qwen3.5-2B, 4B, and 9B, trained on the same CodeContests train pool as our other RL methods (see Table 3). Each step draws n=16 completions per problem at temperature 0.7, applies the SLOO-minus-one transform of Walder and Karkhanis (2025) with k_{opt}=4, and updates with AdamW at learning rate 5×10^{-7}, clip ratio ϵ=0.2, and KL coefficient 0.01 against a frozen reference. The three sizes share these settings; only the base checkpoint differs.

The gradient and parameter-step norms decay as the policy converges, with 9B contracting fastest and 2B plateauing highest—larger models spread the same outcome signal over more parameters. The advantage nonzero rate is the fraction of problems still producing gradient: the SLOO-minus-one transform zeros it out whenever a problem is entirely solved or entirely failed, and larger models cross out of the all-fail regime sooner ($\approx$0.55 at 2B versus $\approx$0.85 at 9B). The transformed-reward variance rises early as the policy first generates mixed pass/fail groups, plateaus through mid-training, and falls late as the policy converges on the solvable subset; the late drop is sharpest at 9B and most muted at 2B.

The 2B trace is the source of the PKPO 2B / CodeContests cell in Table 6 (Appendix A), where PKPO converges at 0.039, below Direct Solve at 0.040. The elevated gradient plateau, the lowest advantage-nonzero rate of the three sizes, and the muted late variance drop all describe the same failure mode: on the 2B base, CodeContests is sparse enough that most groups remain entirely failed, so the SLOO-minus-one signal collapses to a near-trivial landscape and the policy never reliably escapes the all-fail regime.

T UpSkill Training Dynamics

UpSkill shares PKPO's policy class but adds a token-level mutual-information reward r_{mi} to the binary correctness reward r_{correct}. The MI term sets UpSkill apart: PKPO's signal vanishes whenever a problem is entirely solved or entirely failed, while UpSkill's MI term continues to produce a nonzero per-token gradient as long as the latent skill code distinguishes trajectories. The diagnostic panels therefore differ from Figure 8: we report the gradient L2 norm, the two reward components, and the combined-reward variance.

Each step draws one completion per skill at temperature 0.7 under a fixed prompt prefix encoding the latent skill, with n_{skills}=4 (matched to the K=4 inference budget) and MI weight α_{mi}=5.0. Optimizer settings mirror PKPO above; we replace the rank-32 LoRA adapter of the original UpSkill with full fine-tuning across all three sizes, leaving the rest of the hyperparameters unchanged. Parameter-step norm is omitted because the training logger did not record it.

The gradient L2 norm decays for every size, with the 9B trace smoother and the 2B trace visibly noisier. The correctness reward r_{correct} rises along a sigmoid whose asymptote scales with model size, consistent with stronger models reaching higher pass@4. The MI reward r_{mi} saturates early and stays high across all sizes: the latent-skill conditioning keeps trajectories distinguishable even after correctness plateaus, so the MI term can continue to supply token-level signal after correctness rewards

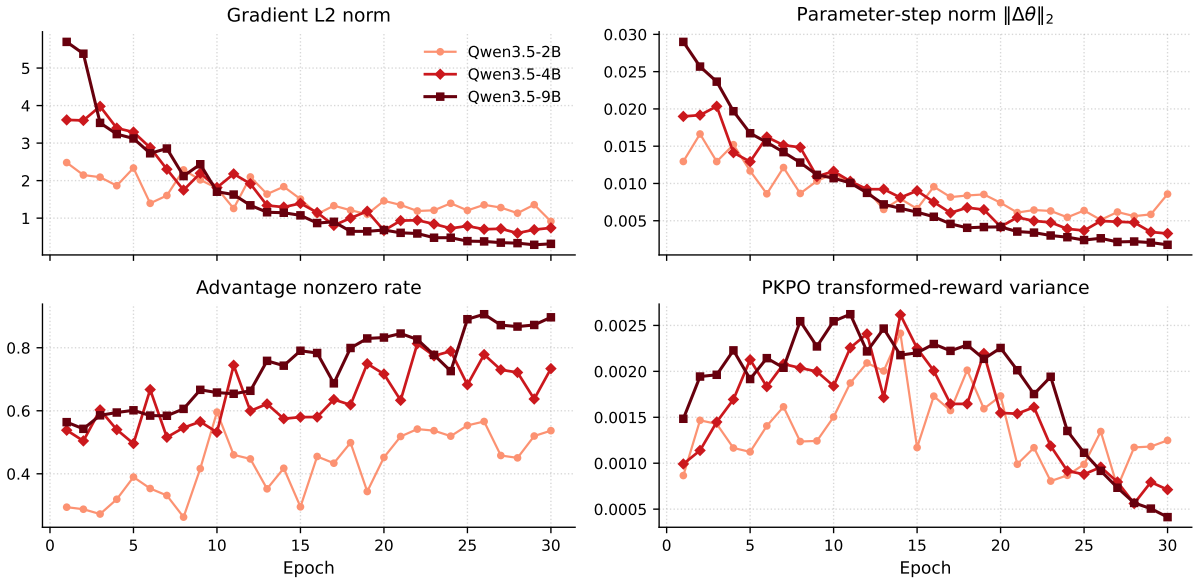


Figure 8: PKPO training dynamics for Qwen3.5-2B, 4B, and 9B over 30 epochs (light to dark red): gradient L2 norm, parameter-step norm $\|\Delta\theta\|_2$, advantage nonzero rate after group normalization, and variance of the PKPO-transformed reward.

become sparse. The combined-reward variance falls over training as the policy concentrates on a smaller set of higher-quality skills.

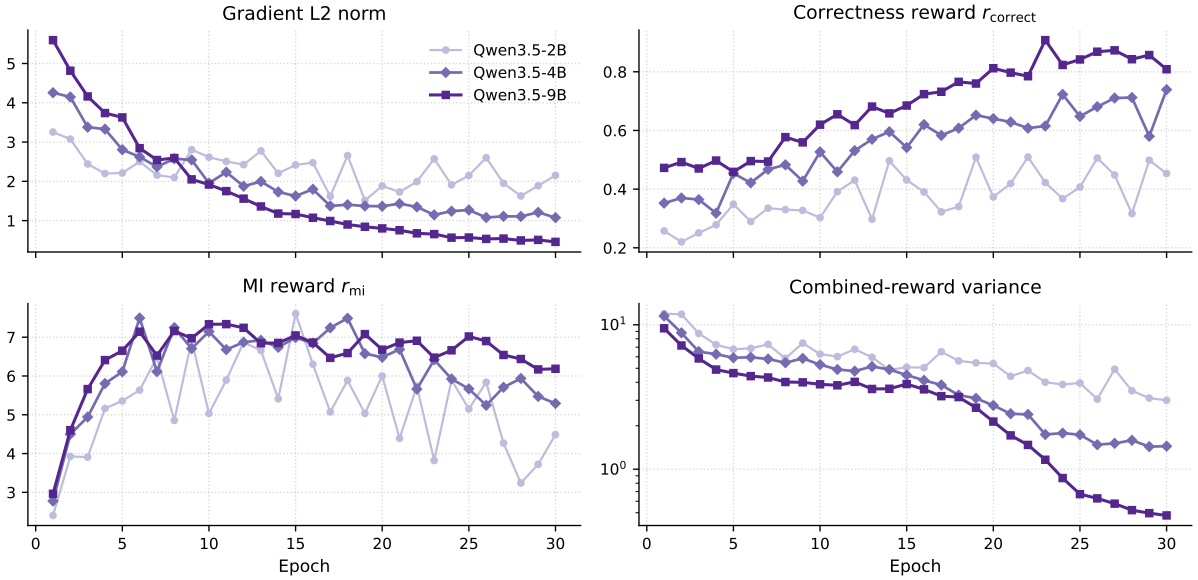


Figure 9: UpSkill training dynamics for Qwen3.5-2B, 4B, and 9B over 30 epochs (light to dark purple): gradient L2 norm, correctness reward r_{correct}, mutual-information reward r_{mi}, and combined-reward variance.