CoSPlay: Cooperative Self-Play at Test-Time with Self-Generated Code and Unit Test

Anonymous authors
Paper under double-blind review

Abstract

Reinforcement Learning with Verifiable Rewards (RLVR) and Test-Time Scaling (TTS) have advanced LLM code generation through executable verification, yet both hinge on Ground-Truth Unit Tests (GT UTs), which are expensive to annotate at scale. GT-free methods instead rely on self-generated UTs to verify and refine codes, but these UTs are often noisy, and so are the codes they check. Treating either side as ground truth to improve the other thus only propagates its errors. ***The key challenge is therefore: with no external reference, how can each side extract usable signals from the other, turning error propagation into mutual error reduction?*** We present **CoSPlay**, a **GT-free** framework that co-evolves codes and UTs through cooperative self-play at inference time. Although neither side is reliable alone, we show theoretically and empirically that pass counts on the Code-UT execution matrix correlate positively with correctness and can thus, to some extent, tell trustworthy codes and UTs from unreliable ones. Building on this signal, CoSPlay first derives discriminative UT ideas from the failure modes of diverse solution ideas. It then alternately updates the two pools, keeping the codes and UTs with higher support counts and using them to fix or regenerate the rest. Finally, such a signal indicates the model does not lean toward any particular wrong behaviors in expectation, so the correct behavior dominates the top codes, whose ties CoSPlay breaks by unsupervised clustering of output signatures on a few random inputs. On four challenging benchmarks, CoSPlay improves the average Best-of-N (BoN) accuracy of Qwen2.5-7B-Instruct from 22.1% to **33.2%** and its UT accuracy from 14.6% to **78.3%**. Without any GT UTs, this matches or surpasses CURE-7B, an RLVR model on the same base, trained on 4.5k samples with GT-UT rewards for both code and UT generation, and CoSPlay further improves CURE-7B itself by **5.7%** BoN. CoSPlay also generalizes across diverse backbones and achieves higher average accuracy than GT-free TTS baselines under comparable token budgets. By letting codes and UTs improve each other without any GT, CoSPlay offers not only a scalable inference strategy but also a new perspective on recursive self-improvement for verifiable problems without external supervision.

1 Introduction

Reinforcement learning with verifiable rewards (RLVR) (Guo et al., 2025; Yu et al., 2025a; Team et al., 2025) and test-time scaling (TTS) (Brown et al., 2024; Lightman et al., 2024; Liu et al., 2025) have substantially improved the coding ability of LLMs. Both rely on executable verification, where unit tests (UTs) provide training rewards or guide inference-time refinement and selection (Wang et al., 2025b; Yu et al., 2025b), yet an execution result is only as trustworthy as the UT behind it. Strong methods therefore depend on Ground-Truth UTs (GT UTs) curated by strong models or human experts (Liu & Zhang, 2025; Zhang et al., 2025a; Wang et al., 2025b), which are expensive to obtain and hard to scale (Chen et al., 2023; Liu et al., 2023). Even CURE (Wang et al., 2025b), which jointly trains a coder and a tester, relies on GT UTs as the supervision signal rather than achieving genuine co-evolution. Neither route has thus escaped its dependence on GT UTs.

The difficulty of the GT-free setting is that self-generated codes and tests can both be wrong. Neither is a fully reliable reference for the other, and existing methods anchor on only one side. Absolute Zero (Zhao et al., 2025a) trains on code–input–output triples obtained by executing its own programs, so the quality of its supervision is capped by whether the model can already write a correct code for a given problem. Conversely, CodeT and MPSC (Chen et al., 2023; Huang et al., 2024) select codes with self-generated tests, and refinement methods update codes with execution feedback (Li et al., 2025a; Shinn et al., 2023; Yu et al., 2025b; Zhong et al., 2024), but none of them improves UT quality at test time (Appendix C). CURE also degrades once its GT UTs are replaced by self-generated

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

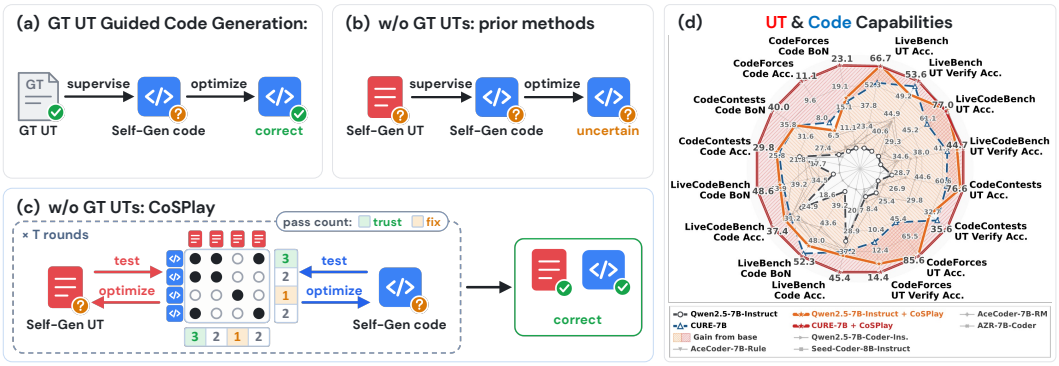


Figure 1: (a–c) **Motivation of CoSPlay.** Verification is only as reliable as its UT: GT UTs anchor code to correctness, prior GT-free methods take unverified self-generated UTs as correct and inherit their errors, and CoSPlay statistically identifies reliable codes and UTs on the execution matrix and lets only these guide both sides over T rounds. (d) On 7B models, CoSPlay lifts Qwen2.5-7B-Instruct to the level of GT-UT-trained CURE-7B and further improves CURE-7B itself.

ones, as their noise is learned as signal (Wang et al., 2025c). On Qwen2.5-7B-Instruct, about 78% of directly sampled UTs are passed by none of the 16 candidate codes (Figure 3). Such UTs neither distinguish candidates nor reveal whether the test or the code pool is at fault. The real question is therefore: ***without GT UTs, how can codes and tests improve each other when neither is an absolutely reliable reference?***

Our key insight is that reliable signals can be extracted from codes and UTs that are not fully reliable themselves. On the Code-UT execution matrix, the two sides discriminate each other: a code passing more UTs, or a UT passed by more codes, is more likely correct (Theorem A.1). Without external supervision, these pass counts can, to some extent, separate reliable from unreliable codes and UTs. CoSPlay therefore filters out the unreliable part of each pool and lets only the reliable part guide the update of the other, rather than taking self-generated codes or UTs as correct (Figure 1).

Building on this insight, CoSPlay proceeds in three stages without GT UTs (Figure 2). *Exploration-Attack-Guided Initialization* derives attack ideas from the failure modes of diverse solution plans, so that the initial UTs are discriminative enough to produce such signals. *Execution-Matrix-Driven Iterative Self-Play* then alternately updates the two pools with pass counts. It refreshes codes and UTs that carry no discriminative information, repairs failing codes with the most-supported discriminative UT, and refreshes the least-supported UTs that may be spuriously coupled with wrong codes, so each improved side yields more reliable counts for the other. After self-play, several codes often tie at the highest pass count. The same condition that makes pass counts informative, namely that the model does not lean to other particular wrong behaviors in expectation, also makes the correct behavior dominates the these codes. *Output-Consensus-Based Cluster Selection* thus breaks the tie by clustering their output signatures on only a few random inputs, without any supervision.

Our contributions are summarized as follows:

1. We introduce **CoSPlay**, a GT-free framework in which codes and self-generated UTs co-evolve at inference time by filtering out unreliable codes and UTs and letting only the reliable ones guide the updates.
2. On four challenging benchmarks, CoSPlay improves **both codes and UTs**. On Qwen2.5-7B-Instruct, UT accuracy rises from 14.6% to 78.3% and BoN from 22.1% to 33.2%, matching or surpassing CURE-7B. CoSPlay further improves CURE-7B by 5.7% BoN, generalizes across base, RL-tuned, and long-CoT backbones, and achieves higher average accuracy than GT-free TTS methods under comparable token budgets.
3. We find and verify, through theory and empirical analyses, that pass counts on the Code-UT execution matrix are a reliable guidance for GT-free Code-UT self-evolution, offering a new perspective on recursive self-improvement for verifiable problems without external supervision.

2 METHODOLOGY

Notation. Let $\mathcal{P}$ denote the problem statement and π the frozen LLM. We write $\pi(\text{role}; \ldots)$ for the output distribution of π under the prompt template named role (Appendix X), with the contents filled into the template listed after the semicolon, so that $y \sim \pi(\text{role}; \ldots)$ denotes one LLM call.

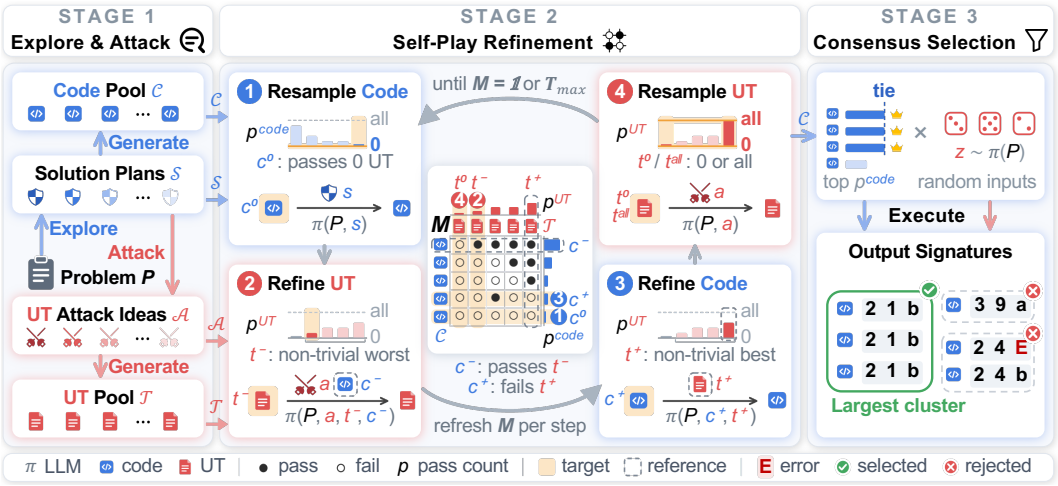


Figure 2: **Method Overview.** Given a coding problem, CoSPlay first explores solution ideas and derives failure-oriented UT ideas from them, so that generated UTs target plausible failure modes. It then builds a Code-UT execution matrix whose bidirectional pass counts drive code cleaning, spurious-coupling breaking, code refinement, and UT refreshing, letting the two pools co-evolve. Finally, codes tied at the highest pass count are executed on random valid inputs and clustered by output signature; the largest consensus cluster yields the final code.
CoSPlay uses seven templates, hint, plan, attack, code, input, output, and fix, each named by what it produces.

2.1 Exploration-Attack-Guided Code and UT Idea Generation

Directly prompting the LLM to generate UTs often yields generic or superficial cases, because the tests are not grounded in the way code candidates may fail (Bekmyradov et al., 2026). Discriminative UTs instead target solution-specific assumptions: for example, duplicate elements can expose solutions that incorrectly assume all values are unique. Motivated by PlanSearch (Wang et al., 2025a), we therefore first explore plausible solution strategies in natural language, and then derive targeted UT ideas from their potential failure modes.

(i) Solution exploration. We first collect a set of high-level hints $\mathcal{H} \sim \pi(\text{hint}; \mathcal{P})$. Each hint is a short observation about $\mathcal{P}$, such as a likely strategy, a useful data structure, or an edge case to watch for, and represents one possible solution direction rather than a complete solution.

$$\mathcal{S} = \{\, s \mid s \sim \pi(\text{plan}; \mathcal{P}, \omega),\ \omega \subseteq \mathcal{H},\ |\omega| \le 2 \,\}, \quad (1)$$

where $\mathcal{S}$ is the solution-plan pool, s is an individual solution plan, and ω is a subset of $\mathcal{H}$ containing at most two hints. Capping the subset size at two balances plan diversity against combinatorial explosion. **(ii) Failure-oriented UT attack idea exploration.** Discriminative UTs should be grounded in how a concrete solution could break, rather than in the problem statement alone. For each plan $s \in \mathcal{S}$, we therefore ask π to *identify potential failure modes, hidden edge cases, and implementation pitfalls* of s, and pool the resulting UT attack ideas over all plans:

$$\mathcal{A} = \{\, a \mid a \sim \pi(\text{attack}; \mathcal{P}, s),\ s \in \mathcal{S} \,\}, \quad (2)$$

where $\mathcal{A}$ is the UT attack-idea pool and a is an individual attack idea derived from a solution plan in $\mathcal{S}$. Figure 3 illustrates the benefit of this initialization. On Qwen2.5-7B-Instruct, approximately 78% of directly sampled UTs are passed by none of the 16 code candidates, providing no distinction among them. Exploration-attack-guided(exp-atk) initialization reduces this proportion to 49%, with more UT passed by some but not all codes. These UT expose differences in code behavior, providing more informative execution feedback for subsequent code-UT self-play.

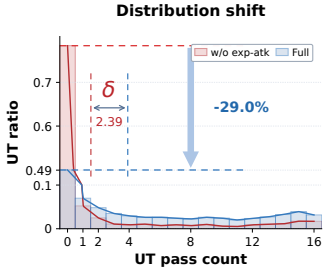


Figure 3: UT pass-count of direct sampling vs. exp-atk; δ denotes the mean gap.

2.2 Code and UT Initialization

The code pool $\mathcal{C} = \{c_i\}_{i=1}^{N_c}$ instantiates shuffled plans from $\mathcal{S}$, one code per plan. Each UT $t = (x, y)$ pairs a valid input with an expected output, generated by two separate calls, $x \sim \pi(\text{input}; \mathcal{P}, \cdot)$ and

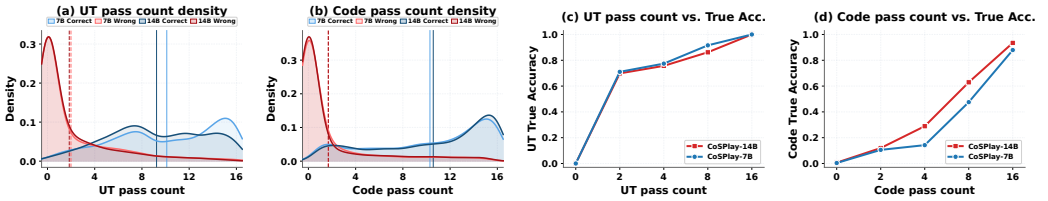


Figure 4: **Round-0 pass-count analysis.** Panels (a-b) show the density distributions of UT/code pass counts for correct/wrong candidates, while panels (c-d) show relationship between GT Acc. and pass count.

then $y \sim \pi(\text{output}; \mathcal{P}, x)$, denoted $t \sim \pi_{\text{UT}}(\cdot)$. The UT pool $\mathcal{T} = \{t_j\}_{j=1}^{N_t}$ mixes two kinds of UTs in equal parts. An attack UT builds its input from an idea drawn from the shuffled $\mathcal{A}$, while a random UT samples a valid input directly from the problem statement without any attack idea. Since a valid input does not guarantee a correct expected output, we sample its expected output four times and accept the UT only if at least three samples agree (*output self-consistency*), which holds for every UT in CoSPlay. Random UTs are needed because highly discriminative inputs often make it difficult for the LLM to produce correct expected outputs (Prasad et al., 2025; Zeng et al., 2025). As a result, initializing all UTs from $\mathcal{A}$ may leave only a less diverse set after self-consistency filtering.

2.3 Execution-Matrix-Driven Iterative Self-Play

Execution matrix. Let $c_i(x)$ denote the output of executing c_i on input x, with $c_i(x) = \perp$ if execution fails. The matrix $\mathbf{M} \in \{0,1\}^{N_c \times N_t}$ with $\mathbf{M}_{ij} = \mathbb{I}[c_i(x_j) = y_j]$ yields two pass counts: $p_j^{\text{UT}} = \sum_i \mathbf{M}_{ij}$, the number of codes passing t_j, and $p_i^{\text{code}} = \sum_j \mathbf{M}_{ij}$, the number of UTs passed by c_i. A UT is *non-trivial* if $0 < p_j^{\text{UT}} < N_c$. Following Figure 2, superscripts mark the pass-count-based selections used below:

$$\begin{aligned} &\mathcal{C}^0 = \{c_i \mid p_i^{\text{code}} = 0\}, \qquad \mathcal{T}^0 = \{t_j \mid p_j^{\text{UT}} = 0\}, \qquad \mathcal{T}^{\text{all}} = \{t_j \mid p_j^{\text{UT}} = N_c\}, \\ &t^{\pm} \in \underset{t_j:\, 0<p_j^{\text{UT}}<N_c}{\arg\max} \pm p_j^{\text{UT}}, \quad \mathcal{C}^- = \{c_i \mid c_i(x^-) = y^-\}, \quad \mathcal{C}^+ = \{c_i \mid c_i(x^+) \neq y^+\}. \end{aligned} \tag{3}$$

Thus $t^{\pm} = (x^{\pm}, y^{\pm})$ are the most- and least-supported non-trivial UTs, $\mathcal{C}^-$ are the codes passing t^-, and $\mathcal{C}^+$ are the codes failing t^+.

Pass counts as evidence. Following prior analyses under idealized i.i.d. sampling (Chen et al., 2023; Huang et al., 2024; Wang et al., 2025b), Theorem A.1 shows that, under the advantage conditions in Eq. 10, the posterior probability of correctness increases with pass count for both codes and UTs. Figure 4 supports this empirically on realistic Round-0 pools before self-play. Each step below therefore uses pass counts to separate reliable from unreliable codes and UTs, and lets only the reliable side guide the update of the other.

Self-play. Each round applies Steps ①–④ with pool sizes fixed. After each step, all Code-UT pairs are re-executed to refresh $\mathbf{M}$, since updating one pool changes the diagnosis of the other. Steps without targets are skipped, and self-play stops once every entry of $\mathbf{M}$ is $\nVdash$ or after $T_{\max}$ rounds.

Step ①: Code Cleaning. Given the UT prior established above, a code in $\mathcal{C}^0$ fails at least one correct UT with probability approaching one exponentially in N_t (Appendix A), so it is likely incorrect and offers little signal for improving the UT pool (Tang et al., 2024). By contrast, a UT in $\mathcal{T}^0$ may still be a valid hard probe that exposes a shared bug in the current code pool, so deleting it may discard useful tests. We therefore replace each all-failing code with a new code from the plan pool:

$$\mathcal{C} \leftarrow (\mathcal{C} \setminus \mathcal{C}^0) \cup \{c_k' \sim \pi(\text{code}; \mathcal{P}, s_k) \mid s_k \in \mathcal{S}\}_{k=1}^{|\mathcal{C}^0|}. \tag{4}$$

Step ②: Breaking Spurious Code-UT Coupling. We next diagnose the UT pool before using it for refinement. Unlike a zero-pass UT, which rejects all codes universally, the least-supported non-trivial UT t^- is passed by only a few codes and may reflect spurious coupling with a specific wrong logic, which code refinement would then propagate to other codes. The goal is not to discard low-support UTs, but to break this suspicious agreement. We therefore refresh t^- in two separate phases: a new input is re-instantiated from an attack idea $a \in \mathcal{A}$, and its expected output is then derived from the problem and the new input, with t^- and one passing code from $\mathcal{C}^-$ given only as a negative reference (marked as "may contain errors"):

$$\mathcal{T} \leftarrow (\mathcal{T} \setminus \{t^-\}) \cup \{t' \sim \pi_{\text{UT}}(a, t^-, \mathcal{C}^-)\}. \tag{5}$$

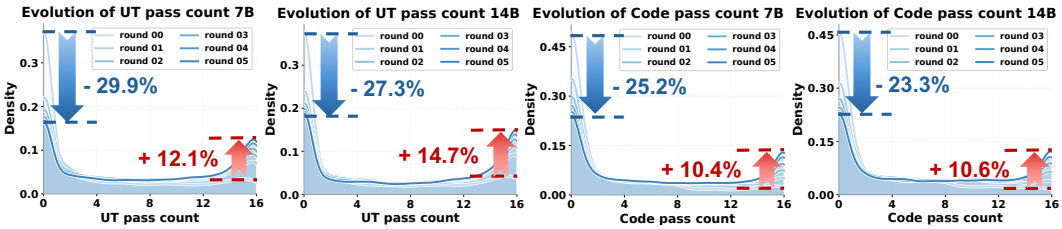


Figure 5: Evolution of pass-count distributions during self-play.

Step ③: Code Fixing. We then refine buggy codes with a trustworthy yet discriminative UT. A UT in $\mathcal{T}^{\text{all}}$ may be trustworthy but provides no refinement signal, so we ***select the non-trivial UT with the highest pass count***, t^+, which is reliable enough and can still distinguish buggy codes. Each code in $\mathcal{C}^+$ is repaired by π given the buggy code, the UT itself, and the actual output of the code:

$$\mathcal{C} \leftarrow \left(\mathcal{C} \setminus \mathcal{C}^+\right) \cup \left\{c_i' \sim \pi\big(\text{fix};\, \mathcal{P}, c_i, t^+, c_i(x^+)\big) \;\middle|\; c_i \in \mathcal{C}^+\right\}. \tag{6}$$

Step ④: Replacing Zero-Discrimination UTs. As refinement improves the code pool, some UTs become trivial: those in $\mathcal{T}^{\text{all}}$ are now too easy, and those in $\mathcal{T}^0$ are more likely noisy or mismatched to the pool (Song et al., 2024; Xie et al., 2025). Since they no longer provide refinement or selection signals (Hu et al., 2025; Ma et al., 2025), we replace them with new UTs built from attack ideas:

$$\mathcal{T} \leftarrow \left(\mathcal{T} \setminus (\mathcal{T}^0 \cup \mathcal{T}^{\text{all}})\right) \cup \left\{t_k' \sim \pi_{\text{UT}}(a_k) \;\middle|\; a_k \in \mathcal{A}\right\}_{k=1}^{|\mathcal{T}^0 \cup \mathcal{T}^{\text{all}}|}. \tag{7}$$

Figure 5 shows that, as self-play proceeds, Both UT and code pass-count distributions progressively shift toward higher-support regions across self-play rounds, suggesting that execution-matrix-driven self-play gradually concentrates support on more reliable UTs and stronger code candidates.

2.4 Output-Consensus-Based Cluster Selection

Because pass counts are integers, the codes with the highest final pass count, $\mathcal{C}_{\text{high}}$, often tie. We break ties without expected outputs: we draw R random valid inputs $z_r \sim \pi(\text{input};\, \mathcal{P})$ and map each $c_i \in \mathcal{C}_{\text{high}}$ to its output signature $\sigma(c_i) = (c_i(z_1), \ldots, c_i(z_R))$.

Motivation. Correct codes share one logic and hence the same signature, whereas wrong codes split across behaviors, and different behaviors coincide on all R inputs with exponentially small probability in R. Hence, because informative pass counts indicate that the model does not lean toward other wrong behaviors over the correct one in expectation (Appendix A.1.1), the largest cluster identifies the correct signatures; Proposition A.3 analyzes this under idealized sampling assumptions. Figure 6 and Appendix R illustrate this: correct codes form a compact cluster, while wrong ones scatter or form smaller groups.

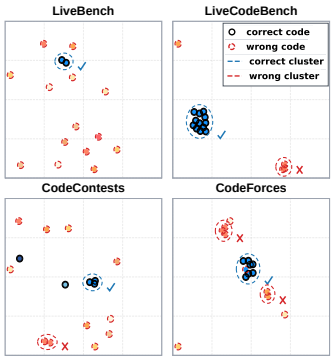


Figure 6: t-SNE visualization of output signature clusters.

Clustering and Selection. If an execution fails ($c_i(z_r) = \perp$), we treat the missing output as missing evidence rather than an output conflict. Two candidates are *compatible* if they produce the same output on every random input on which neither of them fails. Without failures this is exactly signature equality. With failures it is not transitive, so we process candidates in a fixed order: a candidate joins the first existing cluster all of whose members it is compatible with, and starts a new cluster otherwise, so every cluster G_m is pairwise compatible. However, a candidate that fails on most inputs can still appear compatible because little evidence remains to contradict it. We therefore score each candidate by the pairwise agreements it is actually observed to make inside its cluster:

$$w(c_i) = \textstyle\sum_{j \neq i} \sum_{r=1}^{R} \mathbb{I}\big[c_i(z_r) = c_j(z_r) \neq \perp\big], \tag{8}$$

where the sum runs over the other members c_j of the cluster containing c_i. We then select the cluster $G^* = \arg\max_{G_m} \sum_{c_i \in G_m} w(c_i)$ with the largest total score and submit its best-scoring member $c^* = \arg\max_{c_i \in G^*} w(c_i)$. When no execution fails, the cluster score equals $|G_m|(|G_m| - 1)R$ and grows strictly with cluster size, so the procedure reduces to selecting the largest cluster.

Table 1: Performance comparison between **CoSPlay** and **RLVR models [Full]**. **#Data** indicates the number of GT data used for training. **Verifier Acc.** (Verif.): BoN accuracy when the generated UTs select from a fixed pool of Qwen2.5-7B-Ins. code candidates, isolating the discriminative power of corresponding UTs. **UT** : average accuracy of self-generated UTs. **Code** : average accuracy of self-generated codes. **BoN**: Best-of-16 accuracy using self-generated codes and UTs. **Average** reports the dataset-weighted average across benchmarks. Colors (red/blue) indicate values above/below the specific column mean; shades use a rank-stepped scale within each column, so the top ranks are visually separated even when their values are close. Δ: absolute gain of **CoSPlay** over the corresponding base model.

Model	#Data	LiveBench				LiveCodeBench				CodeContests				CodeForces				Average			
		Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN
7B Base Models																					
Qwen2.5-7B-Instruct	-	36.2	9.0	33.1	36.2	31.2	14.4	27.0	31.2	24.1	21.3	21.5	24.1	7.1	12.9	5.0	7.1	22.1	14.6	19.0	22.1
Qwen2.5-7B-Coder	-	40.6	12.1	3.8	19.0	33.4	16.0	4.1	17.7	25.2	21.3	2.6	12.8	6.8	6.7	1.8	3.5	23.4	13.3	3.0	12.0
Qwen2.5-7B-Coder-Ins.	-	42.7	12.5	36.3	47.7	33.7	13.4	28.8	34.4	24.0	12.7	20.9	23.2	6.9	5.3	5.4	7.9	23.5	10.4	20.0	24.5
Seed-Coder-8B-Instruct	-	43.5	26.9	37.6	45.1	36.1	30.0	32.5	40.8	25.0	31.8	18.3	26.9	8.4	17.0	6.4	10.5	25.2	25.5	21.4	28.2
RL Methods																					
AceCoder-7B-Rule	22k	43.5	17.2	36.1	45.8	34.1	17.5	30.5	35.9	24.4	18.8	20.1	23.2	6.4	5.9	5.4	7.7	23.6	13.7	20.5	24.8
AceCoder-7B-RM	329k	43.0	17.7	37.5	48.4	33.7	19.9	30.3	37.1	25.0	20.0	20.6	25.0	6.6	6.6	5.2	7.8	23.6	15.1	20.6	25.8
AZR-7B-Coder	**0**	44.5	23.1	12.5	34.9	37.1	31.7	12.4	29.8	29.1	39.0	13.7	24.4	7.6	16.9	5.3	8.4	26.1	27.0	10.2	21.9
CURE-7B	4.5k	52.3	54.6	37.4	51.6	42.1	58.4	31.2	42.7	33.9	63.7	25.5	34.5	11.5	46.6	7.5	16.1	31.0	54.9	22.5	32.9
Our Method																					
Qwen2.5-7B-Ins. + Ours	**0**	50.3	66.7	38.8	49.7	43.7	77.0	33.2	43.4	32.6	74.4	26.0	34.6	13.6	85.0	6.8	16.8	31.9	78.3	23.3	33.2
Δ vs. Qwen-Ins.	–	**+14.1**	**+57.7**	**+5.7**	**+13.5**	**+12.5**	**+62.6**	**+6.2**	**+12.2**	**+8.5**	**+53.1**	**+4.5**	**+10.5**	**+6.5**	**+72.1**	**+1.8**	**+9.7**	**+9.8**	**+63.7**	**+4.3**	**+11.1**
CURE-7B + Ours	4.5k	53.6	66.2	45.4	52.3	44.7	75.6	37.4	48.6	35.6	76.6	29.8	40.0	14.4	85.6	11.1	23.1	33.4	78.3	27.7	38.6
Δ vs. CURE	–	**+1.3**	**+11.6**	**+8.0**	**+0.7**	**+2.6**	**+17.2**	**+6.2**	**+5.9**	**+1.7**	**+12.9**	**+4.3**	**+5.5**	**+2.9**	**+39.0**	**+3.6**	**+7.0**	**+2.4**	**+23.4**	**+5.2**	**+5.7**
14B Base Models																					
Qwen2.5-14B-Instruct	-	47.7	26.4	39.2	51.0	41.4	38.0	34.5	46.2	30.4	43.2	25.1	34.6	9.9	22.6	7.3	13.3	29.1	32.4	23.9	33.2
Qwen2.5-14B-Coder	-	42.2	13.9	10.8	30.2	36.1	20.7	9.7	26.7	28.3	25.6	6.8	21.3	10.2	16.7	2.1	9.3	26.3	19.5	6.7	20.1
Qwen2.5-14B-Coder-Ins.	-	48.2	37.8	47.2	53.9	39.5	40.6	41.0	49.4	27.5	34.8	25.1	31.4	7.9	18.1	5.6	9.6	27.2	31.5	26.5	32.8
DeepSeek-Coder-V2-16B	-	45.8	31.0	37.4	38.3	39.3	42.1	25.7	36.7	33.6	51.6	22.1	35.1	13.8	37.2	6.5	16.8	30.1	41.0	19.5	29.6
RL Methods																					
AZR-14B-Coder	**0**	39.8	22.2	38.9	43.5	34.1	29.7	35.3	39.8	25.8	35.3	26.4	30.4	8.5	18.9	9.5	16.2	24.3	26.2	25.1	30.3
CURE-14B	4.5k	53.6	77.3	47.7	59.9	45.1	86.1	41.2	50.5	34.9	87.0	32.5	44.1	14.9	77.5	12.0	26.1	33.6	82.4	30.1	41.8
Our Method																					
Qwen2.5-14B-Ins. + Ours	**0**	56.5	70.1	47.9	57.6	46.2	74.4	43.5	54.8	34.0	71.7	31.5	40.0	14.3	86.1	11.8	24.6	34.0	77.6	30.8	42.0
Δ vs. Qwen-Ins.	–	**+8.8**	**+43.7**	**+8.7**	**+6.6**	**+4.8**	**+36.4**	**+9.0**	**+8.6**	**+3.6**	**+28.5**	**+6.4**	**+5.4**	**+4.4**	**+63.5**	**+4.5**	**+11.3**	**+4.9**	**+45.2**	**+6.9**	**+8.8**
CURE-14B + Ours	4.5k	54.9	71.5	52.4	60.7	45.5	76.2	46.9	55.3	34.9	75.1	36.8	45.3	15.3	89.4	19.1	32.6	34.1	80.1	36.0	46.2
Δ vs. CURE	–	**+1.3**	**-5.8**	**+4.7**	**+0.8**	**+0.4**	**-9.9**	**+5.7**	**+4.8**	**+0.0**	**-11.9**	**+4.3**	**+1.2**	**+0.4**	**+11.9**	**+7.1**	**+6.5**	**+0.5**	**-2.3**	**+5.9**	**+4.4**

3 EXPERIMENTS

3.1 SETTINGS

Datasets. For evaluation, we adopt the coding benchmarks suite curated by CURE (Wang et al., 2025b), which includes four widely used datasets: LiveBench (White et al., 2025), LiveCodeBench (Jain et al., 2025b), CodeContests (Li et al., 2022), and CodeForces (Penedo et al., 2025). Detailed information can be viewed in Appendix K. Table 1, Figure 1, and Appendix O report results on four datasets, with the average across benchmarks weighted by dataset size. To reduce computational cost, all other results are obtained on a 3 seed (41, 42, 43) × 200-problem sample of 50 problems per dataset as detailed in Appendix K. Captions label results as **[Full]** or **[3×200]**, with details of the **evaluation protocol** provided in Table 1.

Baselines. We use Qwen2.5-7B-Instruct and Qwen2.5-14B-Instruct (Bai et al., 2023) as the standard base models for CoSPlay and other TTS baselines: S* (Li et al., 2025a), SFS (Light et al., 2025), MPSC (Huang et al., 2024), CodeTree (Li et al., 2025b), ThinkCoder (Zhang et al., 2025b) and PowerSampling (Karan & Du, 2025). We further benchmark against strong instruct and RLVR-trained models at 14B and 7B scales. At the 14B scale, baselines include Qwen2.5-14B-Coder, Qwen2.5-14B-Coder-Instruct (Hui et al., 2024), Absolute-Zero-14B (Zhao et al., 2025a), and CURE-14B (Wang et al., 2025b). At the 7B scale, we include Qwen2.5-7B-Coder, Qwen2.5-7B-Coder-Instruct (Hui et al., 2024), Seed-Coder-8B-Instruct (Seed et al., 2025), AceCoder-RM-7B, AceCoder-Rule-7B (Zeng et al., 2025), CURE-7B (Wang et al., 2025b), and Absolute-Zero-Coder-7B (Zhao et al., 2025a). More detailed baseline settings are provided in Appendix D.

Implementation Details. Open-weight models are served with vLLM (Kwon et al., 2023) on two NVIDIA H100 (80GB) GPUs with temperature 0.8, top-p 0.95, top-k 40, and prompt-guided/enabled CoT. Gemini-2.0-Flash and DeepSeek-V3.2-685B are accessed through their respective APIs. CoS-

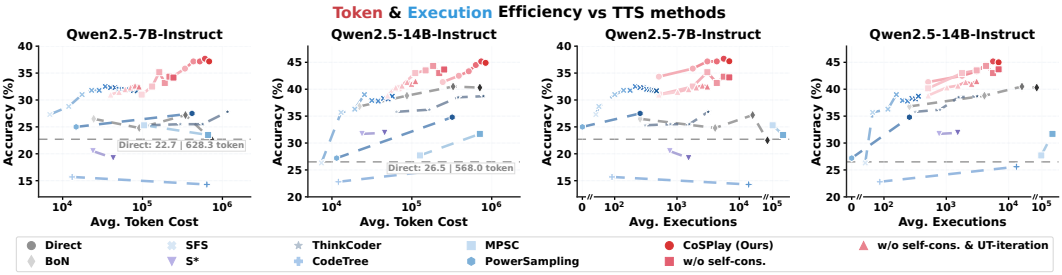


Figure 7: Token cost (left) and execution count (right) versus Acc of TTS methods and CoSPlay on Qwen2.5-Instruct models **[3×200]**. **Acc** is the accuracy of the single program each method submits For each baseline method, darker markers indicate its scaled variant with a larger inference budget. For CoSPlay, markers in each setting show the trajectory from self-play round 0 to round 5. Detailed scaling configurations and data are provided in Appendix D and Appendix F.

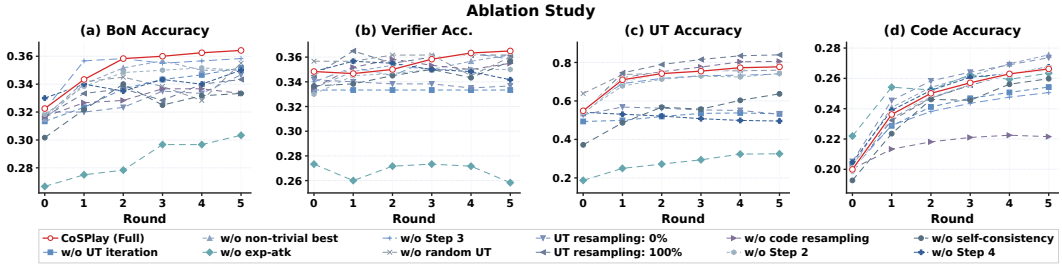


Figure 8: **Per-round ablation of CoSPlay on Qwen2.5-7B-Instruct [3×200].** Each panel tracks one metric across self-play rounds (R0–R5) when a single component is removed: (a) BoN, (b) Verifier Acc., (c) UT accuracy, and (d) Code accuracy, defined as in Table 1. **Full** is the complete method.

Play uses $N_c = N_t = 16$, $T_{\max} = 5$, and $R = 16$. Expected outputs are accepted when at least three of four samples agree. Execution time limits follow the CURE benchmark suite (Wang et al., 2025b). Further details are given in Appendix E.

3.2 Results

CoSPlay substantially improves both UT and code quality. Table 1 shows that CoSPlay consistently improves base models across most metrics without GT data or additional training. On Qwen2.5-7B-Instruct, Verifier Acc., UT accuracy, code accuracy, and BoN increase from 22.1%, 14.6%, 19.0%, and 22.1% to 31.9%, 78.3%, 23.3%, and 33.2%, respectively. CoSPlay surpasses CURE-7B in BoN (33.2% vs. 32.9%) and further improves CURE-7B and CURE-14B by 5.7% and 4.4 % , demonstrating complementarity with RLVR.

Efficiency Analysis of CoSPlay. Figure 7 plots Acc against token cost and execution count on both backbones, and three conclusions follow. First, CoSPlay scales fastest with tokens and reaches the highest ceiling: every baseline flattens as its budget grows, whereas CoSPlay keeps climbing round by round, reaching 37.2% on 7B with 682K tokens while ThinkCoder spends 1.16M to reach 27.8%. These results suggest that, CoSPlay converts additional inference compute into accuracy gains more efficiently. Extra budget thus buys accuracy for CoSPlay but not for existing TTS methods, whose bottleneck is verification quality rather than sample count. Second, the gain is not paid for in execution: at matched execution counts CoSPlay leads by a wide margin and surpasses execution-heavy MPSC with an order of magnitude fewer executions. Third, the lightweight variant without self-consistency and UT iteration already beats the strongest TTS baseline at about 90K tokens and fewer executions, offering a cheaper operating point that stays ahead on both axes.

Ablation Study. Figure 8 shows that the full model achieves the highest final-round BoN. We focus on BoN, which captures the combined effectiveness of code generation and verification, and Verifier Acc., which assesses UT discriminative ability on a fixed code pool. Average code accuracy and average UT accuracy serve as auxiliary diagnostic metrics. Removing exp-atk leaves code accuracy nearly intact yet collapses UT accuracy and verify ability, making it the worst variant in BoN: the bottleneck is verification rather than generation, which is exactly what Stage 1 targets. Disabling self-consistency substantially reduces Round-0 UT accuracy, and freezing the UT pool (*w/o UT iteration*) flattens Verifier Acc. after Round 1, highlighting their roles in improving verification.

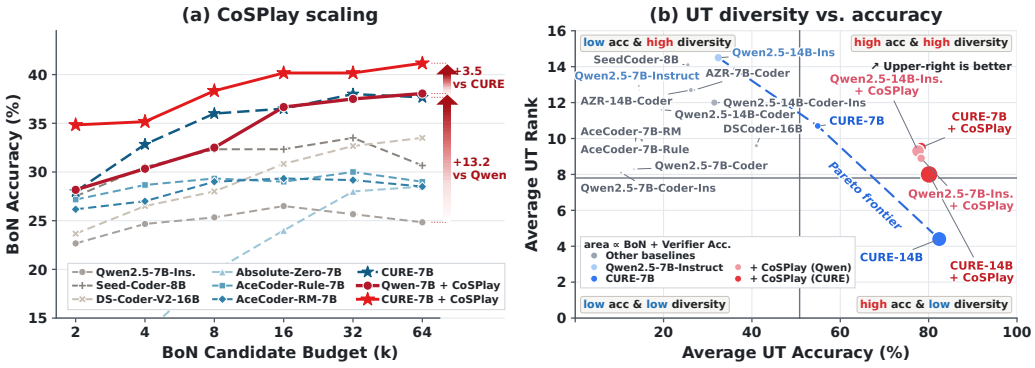


Figure 9: (a) shows the scalability of CoSPlay with candidate-pool size **[3×200]**. (b) shows the trade-off between UT diversity and UT Acc. for CoSPlay and other methods **[Full]**.

On the code side, retaining all-failing candidates (*w/o code resampling*) stalls code accuracy at approximately 22%. Steps 1–3 configuration starts at 33.0% BoN before self-play and reaches 35.0% at Round 5 without Step 4, compared with 36.7% for the full model. Steps 2 and 3 and non-trivial-best selection also improve final-round BoN. The incremental analysis in Appendix G further examines component gains and costs. Detailed ablation results are provided in Appendix L.

4 Discussion

Generalization across model families. CoSPlay is a general inference-time complement whose benefit scales with how much verification quality it can recover from the backbone. Table 2 covers base, RL-tuned, and long-CoT backbones (per-dataset results in Appendix H): all ten improve in UT accuracy, code accuracy, and BoN, with average BoN rising from 45.8% to 52.2%, and Verifier Acc. rises on seven of them by up to +10.7%, so the more accurate UTs are also more discriminative. The gains are largest where the backbone's own UTs are weakest: four models with initial UT accuracy below 40% gain over 10% BoN, whereas Qwen3-4B and CURE-4B, whose native UTs are already accurate, gain about 2%.

Table 2: Generalization of CoSPlay **[3×200]**.

Model / Method	Verif. ↑	UT ↑	Code ↑	BoN ↑
Base Models				
Seed-Coder-8B-Instruct	29.7	22.1	24.1	32.3
+ CoSPlay (Ours)	**36.5**$^{+6.8}$	**77.4**$^{+55.3}$	**33.2**$^{+9.1}$	**42.3**$^{+10.0}$
DeepSeek-Coder-V2-16B	33.3	36.6	22.5	30.8
+ CoSPlay (Ours)	**36.2**$^{+2.9}$	**82.7**$^{+46.1}$	**34.2**$^{+11.7}$	**41.3**$^{+10.5}$
Gemini-2.0-Flash	**36.0**	40.8	48.6	53.8
+ CoSPlay (Ours)	35.8$^{-0.2}$	**74.0**$^{+33.2}$	**52.2**$^{+3.6}$	**55.3**$^{+1.5}$
DeepSeek-V3.2-685B	**40.0**	74.0	56.6	65.7
+ CoSPlay (Ours)	39.2$^{-0.8}$	**79.5**$^{+5.5}$	**63.8**$^{+7.2}$	**68.2**$^{+2.5}$
RL-tuned Models				
Absolute-Zero-7B	29.0	23.6	9.2	24.0
+ CoSPlay (Ours)	**31.7**$^{+2.7}$	**84.8**$^{+61.2}$	**22.4**$^{+13.2}$	**30.2**$^{+6.2}$
AceCoder-Rule-7B	27.7	9.5	26.2	29.0
+ CoSPlay (Ours)	**37.2**$^{+9.5}$	**73.6**$^{+64.1}$	**30.1**$^{+3.9}$	**39.5**$^{+10.5}$
AceCoder-RM-7B	27.0	12.0	25.8	29.3
+ CoSPlay (Ours)	**37.7**$^{+10.7}$	**72.7**$^{+60.7}$	**30.8**$^{+5.0}$	**40.5**$^{+11.2}$
Long-CoT Models				
Qwen3-4B	37.7	64.9	59.5	64.7
+ CoSPlay (Ours)	**39.2**$^{+1.5}$	**77.0**$^{+12.1}$	**62.0**$^{+2.5}$	**67.0**$^{+2.5}$
CURE-4B	**38.7**	80.9	59.5	65.0
+ CoSPlay (Ours)	38.5$^{-0.2}$	**87.6**$^{+6.7}$	**62.5**$^{+3.0}$	**66.7**$^{+1.7}$
OCR-Nemotron-1.1-7B	36.8	28.6	52.1	63.2
+ CoSPlay (Ours)	**39.5**$^{+2.7}$	**75.0**$^{+46.4}$	**65.2**$^{+13.1}$	**71.3**$^{+8.1}$

Scalability of CoSPlay with Increasing Candidate Pool Size. As shown in Figure 9(a), both Qwen2.5-7B-Instruct and CURE-7B obtain higher BoN as k increases with CoSPlay, indicating that our method can effectively exploit larger inference-time candidate pools. Notably, Qwen2.5-7B-Instruct + CoSPlay becomes comparable to CURE-7B when $k \geq 16$, despite using no GT data or training. At $k = 64$, CoSPlay improves Qwen2.5-7B-Instruct by 13.2%, while CURE-7B + CoSPlay continues to benefit from larger budgets, showing that CoSPlay can further enhance already RLVR-tuned models. Moreover, both CoSPlay settings show no clear saturation at $k = 64$, suggesting that performance may continue to improve with larger candidate pools; however, we leave this further scaling study to future work due to computational resource constraints.

UT Accuracy-Diversity Trade-off of CoSPlay. CoSPlay improves UT diversity and accuracy together rather than trading one for the other. We measure diversity by the *UT rank* $\mathcal{T}_{\text{rank}} = |\{x_j \mid (x_j, y_j) \in \mathcal{T}\}|$, the number of unique test inputs in $\mathcal{T}$; it ignores semantic diversity but is cheap to compute. In Figure 9(b), baselines lie on a frontier that favors either diversity or accuracy, and RL-tuned CURE-14B in particular reaches high accuracy at low rank, consistent with RL narrowing

Table 3: **Decomposing BoN into coverage and selection [3×200].** All scores are percentages under Best-of-16. Coverage (Cov.) is the percentage of problems whose final pool contains at least one correct solution. Selection (Sel.) is the percentage of these problems for which the final choice is correct. BoN is the overall accuracy (%), equal to Cov. × Sel. /100.

Metric	Setting	Qwen2.5-7B-Ins	CURE-7B	Qwen2.5-14B-Ins	CURE-14B	Qwen3-4B	CURE-4B	OCR-1.1-7B
Coverage	Base	42.8	47.8	50.0	53.8	74.5	72.7	74.5
	w/o self-play	42.0	46.8	50.3	54.0	74.3	74.0	73.3
	w/ self-play	**45.8**$^{+3.0}$	**51.7**$^{+3.9}$	**54.2**$^{+4.2}$	**59.7**$^{+5.8}$	**74.2**$^{-0.3}$	**73.0**$^{+0.3}$	**77.8**$^{+3.3}$
Selection	Base	61.9	79.1	73.7	88.9	86.8	89.4	84.8
	w/o self-play	81.7	80.1	84.1	83.0	87.2	88.5	90.9
	w/ self-play	**81.1**$^{+19.2}$	**80.6**$^{+1.5}$	**83.0**$^{+9.3}$	**83.5**$^{-5.3}$	**90.3**$^{+3.5}$	**91.3**$^{+1.9}$	**91.6**$^{+6.9}$
BoN	Base	26.5	37.8	36.8	47.8	64.7	65.0	63.2
	w/o self-play	34.3	37.5	42.3	44.8	64.8	65.5	66.7
	w/ self-play	**37.2**$^{+10.7}$	**41.7**$^{+3.9}$	**45.0**$^{+8.2}$	**49.8**$^{+2.0}$	**67.0**$^{+2.3}$	**66.7**$^{+1.7}$	**71.3**$^{+8.2}$

the output distribution (Zhao et al., 2025b). CoSPlay moves every backbone toward the upper right, since idea-level exploration adds diverse inputs while pass-count filtering removes unreliable ones; larger markers show that the better trade-off also yields higher BoN and Verifier Acc.

Attributing BoN Gains to Coverage and Selection. CoSPlay improves BoN by simultaneously raising the quality of both codes and UTs, with their relative contributions varying across backbones. Table 3 decomposes BoN into Coverage and Selection based on expert verification. CoSPlay improves BoN on all seven backbones, typically by raising both factors, consistent with the two roles of the execution matrix: pass counts sharpen UT scoring and thus Selection, and they also expose failing codes for repair and thus raise Coverage. When verification is the bottleneck (Qwen2.5-7B-Instruct, Qwen3-4B, CURE-4B), the gain comes almost entirely from Selection. When code generation is the bottleneck (CURE-14B), it comes from Coverage, with Selection dipping slightly because newly covered problems are harder to select from. When both sides have headroom (OCR-Nemotron-1.1-7B), the gains compound. Notably, although CoSPlay's UTs are entirely model-generated, the high Selection indicates their high quality: when a correct solution is present in the final code pool, the UTs select it in most cases, so few failures stem from the UTs failing to identify correct solutions.

Why Cluster-Based Selection Works. A model that yields discriminative pass counts does not lean to other particular wrong behaviors in expectation, so the correct behavior should be the most common one among the top codes and form the largest signature cluster. Figure 10 tests this with statistics averaged equally across four datasets. For both 7B and 14B models, correct codes fall mostly into large clusters while incorrect codes scatter into small ones (left), and code accuracy rises with cluster size (right). Cluster size is thus an effective GT-free proxy for correctness.

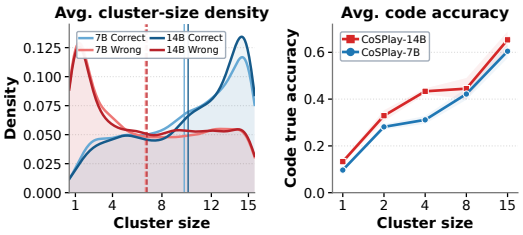


Figure 10: Cluster-size statistics averaged equally across four datasets. Left: density of cluster sizes for correct and incorrect code. Right: code accuracy as a function of cluster size.

5 Conclusion

We introduce **CoSPlay**, a **GT-free** test-time framework for code generation, and show theoretically and empirically that execution-matrix pass counts are a reliable internal signal for letting codes and UTs improve each other without GT. Starting from failure-oriented UT ideas, CoSPlay does not take self-generated codes or UTs as correct, but keeps those with higher support counts and uses them to fix or regenerate the rest. Since such a signal indicates that the model does not favor any particular wrong answer in expectation, unsupervised signature clustering suffices as a lightweight tie-breaker. CoSPlay raises UT accuracy from 14.6% to 78.3% and BoN from 22.1% to 33.2% on Qwen2.5-7B-Instruct, matching or surpassing RLVR training of the same model. It also generalizes across backbones and outperforms GT-free TTS baselines at comparable budgets, offering a scalable inference strategy and a new perspective on recursive self-evolution for verifiable problems without external supervision. *Limitation and Future Directions.* As a proof of concept, CoSPlay is validated only on single-turn code generation, leaving long-horizon agentic tasks to future work.

AI USE STATEMENT

All core research ideas, the proposed method, the experimental design, and the interpretation of results in this work were conceived and developed entirely by the human authors, and the manuscript was written by the authors. We have not used generative AI tools for research ideation, for formulating mathematical claims or assisting with proofs, for designing the methodology or experiments, for generating synthetic data, or for interpreting results. For the implementation, the experimental code was primarily written by the authors, with generative AI tools assisting in code development and debugging; all AI-assisted code was reviewed, verified, and tested for correctness by the authors. Additionally, we used generative AI tools to polish the authors' written text for grammar and readability; no scientific content, claim, or result was produced by AI. We note that LLM-based generation of code and unit tests is itself the subject of study in this paper; all such LLM usage constitutes the proposed method and experiments and is fully described in the main text. We have reviewed all AI-assisted work and take responsibility for the final content of this work.

REFERENCES

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, and et. al. Qwen technical report. *arXiv*, 2023.

Vekil Bekmyradov, Noah C. Pütz, and Thomas Bartz-Beielstein. Llms taking shortcuts in test generation: A study with sap hana and leveldb. *arXiv*, 2026.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *ICLR*, 2023.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Xinyu Zhang, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. Revisit self-debugging with self-generated tests for code generation. In *ACL Long*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, and et. al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *EMNLP*, 2023.

Haichuan Hu, Xiaochen Xie, and Quanjun Zhang. Repair-r1: Better test before repair. *arXiv*, 2025.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. Enhancing large language models in coding through multi-perspective self-consistency. In *ACL 2024 (Long Papers)*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *arXiv*, 2024.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. In *ICML*, 2025a.

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

Aayush Karan and Yilun Du. Reasoning with sampling: Your base model is smarter than you think. *arXiv*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning. *arXiv*, 2025.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025a.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. In *NAACL*, 2025b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. SFS: Smarter code space search improves LLM inference scaling. In *ICLR*, 2025.

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

Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*, 2023.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling. *arXiv*, 2025.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. Dynamic scaling of unit tests for code reward modeling. In *ACL Long*, 2025.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. Codeforces. *Hugging Face repository*, 2025.

Archiki Prasad, Elias Stengel-Eskin, Justin Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. In *COLM*, 2025.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. Seed-coder: Let the code model curate data for itself. *arXiv*, 2025.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, 2017.

Jialin Song, Jonathan Raiman, and Bryan Catanzaro. Effective large language model debugging with best-first tree search. *arXiv*, 2024.

Hao Tang, Keya Hu, Jin Peng Zhou, Si Cheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. Code repair with LLMs gives an exploration-exploitation tradeoff. In *NeurIPS*, 2024.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, and et. al. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv*, 2025.

Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M. Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves LLM search for code generation. In *ICLR*, 2025a.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. CURE: Co-evolving coders and unit testers via reinforcement learning. In *NeurIPS*, 2025b.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. OpenReview discussion of "CURE: Co-evolving coders and unit testers via reinforcement learning". NeurIPS 2025. `https://openreview.net/forum?id=wPdBe9zxNr`, 2025c.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-limited llm benchmark. In *ICLR*, 2025.

Zhihui Xie, Jie chen, Liyu Chen, Weichao Mao, Jingjing Xu, and Lingpeng Kong. Teaching language models to critique via reinforcement learning. In *ICML*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and et. al. Dapo: An open-source llm reinforcement learning system at scale. In *NeurIPS*, 2025a.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Xingru Jiang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Reasoning through execution: Unifying process and outcome rewards for code generation. In *ICML*, 2025b.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACECODER: Acing coder RL via automated test-case synthesis. In *ACL Long*, 2025.

Kechi Zhang, Ge Li, Jia Li, Yihong Dong, Jia Li, and Zhi Jin. Focused-DPO: Enhancing code generation through focused preference optimization on error-prone points. In *ACL findings*, 2025a.

Xiaoqing Zhang, Yuhan Liu, Flood Sung, Xiuying Chen, Shuo Shang, and Rui Yan. Thinking before running! efficient code generation with thorough exploration and optimal refinement. In *ACL findings*, 2025b.

Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. o1-coder: an o1 replication for coding. *arXiv*, 2024.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *NeurIPS*, 2025a.

Rosie Zhao, Alexandru Meterez, Sham M. Kakade, Cengiz Pehlevan, Samy Jelassi, and Eran Malach. Echo chamber: RL post-training amplifies behaviors learned in pretraining. In *COLM*, 2025b.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *ACL findings*, 2024.

APPENDIX

A Theory Analysis **15**

A.1 Analysis of Pass-Count Signals 15

A.2 Analysis of Output-Consensus-Based Cluster Selection 18

B Detailed Algorithm Design **20**

C Related Work **21**

C.1 Fine-tuning Methods for Code Generation 21

C.2 TTS for LLM Code Generation 21

C.3 Self-Play in LLM 21

C.4 Search in Natural Language Space 21

D Detailed baseline setting **22**

E Additional Implementation Details **22**

F Analysis of the cost and performance balance for TTS methods **24**

G Detailed data of incremental ablation **26**

H Detailed data about generalization of our method **27**

I CoSPlay under Partially Available GT UTs **28**

J Effectiveness of CoSPlay for Code Verification versus Reward Model **29**

K Dataset Overview **29**

L Detailed Ablation Results **30**

L.1 Ablation Variant Descriptions 30

L.2 Analysis 31

M Detailed data of UT pass count distribution about Idea-Level Exploration vs. direct sample **31**

N Scalability Analysis of Random Input Size for Clustering **31**

O Variance analysis **33**

P Detailed Evolution of Execution Statistics **34**

P.1 Detailed Evolution of Cluster-Size Statistics for Correct and Incorrect Codes . . . 34

P.2 Detailed Evolution of UT Pass-Count Density 37

P.3 Detailed Evolution of Code Pass-Count Density 40

Q Detailed Breakdown of the Relationship between Execution Statistics and True Accuracy **43**

Q.1 Detailed Breakdown of the Relationship between Cluster Size and Code True Accuracy . . . 43

Q.2 Detailed Breakdown of the Relationship between UT Pass Count and True Accuracy . . . 44

Q.3 Detailed Breakdown of the Relationship between Code Pass Count and True Accuracy . . . 46

R t-SNE Visualization of Clustering Strategy **47**

S Average evolution of the UT and code pass-count distributions across iterative self-play rounds **48**

T Detailed metrics evolution during self-play stage **49**

T.1 Detailed Verifier Accuracy Evolution during Self-Play Rounds . . . 49

T.2 Detailed BoN evolution during self-play rounds. . . . 49

T.3 Detailed Code accuracy evolution during self-play rounds . . . 50

T.4 Detailed UT accuracy evolution during self-play rounds . . . 50

U CoSPlay on the Latest LiveCodeBench v6 **51**

V CoSPlay under Equal Token Budgets on Reasoning Model. **52**

W Case study **53**

W.1 Case Study of successful code fixing . . . 53

W.2 Case study of successful locating Code–UT Coupling . . . 60

X Prompts used in our method **67**

A Theory Analysis

A.1 Analysis of Pass-Count Signals

For a clean analysis, we study pass counts under idealized assumptions, namely one round of i.i.d. sampling without self-play updates, and show why they signal the correctness of both codes and UTs.

A.1.1 When Pass Counts Indicate Correctness

Consider one round of independent sampling without self-play updates, as in the standard BoN baseline. Fix a problem with code pool $\mathcal{C} = \{c_i\}_{i=1}^{N_c}$ and UT pool $\mathcal{T} = \{t_j\}_{j=1}^{N_t}$, $t_j = (x_j, y_j)$. The execution matrix and pass counts are

$$M_{ij} := \mathbf{1}\{c_i(x_j) = y_j\}, \qquad p_j^{\mathrm{UT}} = \sum_{i=1}^{N_c} M_{ij}, \qquad p_i^{\mathrm{code}} = \sum_{j=1}^{N_t} M_{ij},$$

where $c_i(x)$ is the output of c_i on x, with $c_i(x) = \perp$ if execution fails, so a failed execution never passes.

Sampling model. Let H_C^+ and H_T^+ be the events that a randomly sampled code and a randomly sampled test are correct, and H_C^- and H_T^- their complements; these superscripts mark correctness and are unrelated to the selection superscripts of Section 2.3. Write

$$\rho_C := \mathbb{P}(H_C^+), \qquad \rho_T := \mathbb{P}(H_T^+), \qquad 0 < \rho_C, \rho_T < 1.$$

Codes and tests are sampled independently. Given the correctness of a code c_i, its pass events $M_{i1}, \ldots, M_{iN_t}$ on different tests are independent; likewise, given the correctness of a test t_j, the pass events $M_{1j}, \ldots, M_{N_c j}$ of different codes are independent. A correct code passes every correct test and fails every incorrect one, so an outcome is random only when the code is incorrect; let $\varepsilon_1, \varepsilon_2 \in [0, 1]$ be the probabilities that an incorrect code passes a correct test and an incorrect test:

$$\begin{aligned} \mathbb{P}(M = 1 \mid H_C^+, H_T^+) &= 1, \qquad \mathbb{P}(M = 1 \mid H_C^+, H_T^-) = 0, \\ \mathbb{P}(M = 1 \mid H_C^-, H_T^+) &= \varepsilon_1, \qquad \mathbb{P}(M = 1 \mid H_C^-, H_T^-) = \varepsilon_2. \end{aligned}$$

Averaging over the other side gives the probability that a correct or incorrect test is passed by a random code, θ_1, θ_0, and that a correct or incorrect code passes a random test, φ_1, φ_0:

$$\begin{aligned} \theta_1 &:= \mathbb{P}(M = 1 \mid H_T^+) = \rho_C + (1 - \rho_C)\varepsilon_1, & \theta_0 &:= \mathbb{P}(M = 1 \mid H_T^-) = (1 - \rho_C)\varepsilon_2, \\ \varphi_1 &:= \mathbb{P}(M = 1 \mid H_C^+) = \rho_T, & \varphi_0 &:= \mathbb{P}(M = 1 \mid H_C^-) = \varepsilon_1\rho_T + \varepsilon_2(1 - \rho_T). \end{aligned} \tag{9}$$

Advantage conditions. Pass counts are informative when correct objects pass more often than incorrect ones, i.e., when the gaps below are positive:

$$\begin{aligned} \Delta_U &:= \theta_1 - \theta_0 = \rho_C + (1 - \rho_C)(\varepsilon_1 - \varepsilon_2), & \Delta_U > 0 &\iff \rho_C > \rho_C^\star := \frac{\varepsilon_2 - \varepsilon_1}{1 + \varepsilon_2 - \varepsilon_1}, \\ \Delta_C &:= \varphi_1 - \varphi_0 = \rho_T(1 - \varepsilon_1) - \varepsilon_2(1 - \rho_T), & \Delta_C > 0 &\iff \rho_T > \rho_T^\star := \frac{\varepsilon_2}{1 - \varepsilon_1 + \varepsilon_2}. \end{aligned} \tag{10}$$

Both equivalences hold whenever $(\varepsilon_1, \varepsilon_2) \neq (1, 0)$; in that degenerate case incorrect codes behave exactly like correct ones.

UT side. In Δ_U, correct codes contribute ρ_C, since they pass every correct UT and no incorrect one, and incorrect codes contribute $(1 - \rho_C)(\varepsilon_1 - \varepsilon_2)$, so $\Delta_U \geq \rho_C > 0$ whenever $\varepsilon_1 \geq \varepsilon_2$. We expect this of a reasonable base model, which should not favor wrong answers: on the same input, an incorrect code is no more likely to return the particular wrong answer an incorrect UT expects than the correct answer. Figure 4(a, c) supports this at Round 0: correct and incorrect UTs have well-separated pass-count distributions, and UT accuracy rises with pass count.

Code side. In Δ_C, correct tests contribute $\rho_T(1 - \varepsilon_1)$ and incorrect tests $-(1 - \rho_T)\varepsilon_2$: every correct code fails an incorrect test while an incorrect code may pass it, so this term always favors incorrect codes, and $\Delta_C > 0$ requires enough correct, discriminative tests, $\rho_T > \rho_T^\star$. Stage 1 (Section 2.1) is designed for this: attack UTs aim to lower ε_1 and output self-consistency raises ρ_T; Figure 3 shows that it turns many UTs passed by no code into UTs that separate codes. As a result, code pass counts are informative from Round 0, where self-play begins: Figure 4(b, d) shows that correct and incorrect codes have well-separated pass-count distributions, and code accuracy rises with pass count.

Table 4: Round-wise statistics and pass-count analysis of Qwen2.5-7B-Instruct with CoSPlay. Each dataset pairs an R0-R5 statistics table (left) with four pass-count plots (right). Here, ρ_C and ρ_T denote the proportions of correct codes and UTs, while $\rho_C^\star$ and $\rho_T^\star$ denote the corresponding theoretical thresholds for pass counts to be positively informative of correctness. The term $\varepsilon_1 - \varepsilon_2$ measures the difference between the probability that an incorrect code passes a correct UT and that it passes an incorrect UT.

LiveBench

Round	$\epsilon_1 - \epsilon_2$	ρ_C	$\rho_C^\star$	ρ_T	$\rho_T^\star$
R0	0.1657	0.2839	-0.1987	0.5221	0.1225
R1	0.1435	0.3407	-0.1675	0.6280	0.2728
R2	0.1224	0.3584	-0.1395	0.6468	0.3396
R3	0.1424	0.3696	-0.1661	0.6620	0.3718
R4	0.1477	0.3800	-0.1733	0.6731	0.4040
R5	0.1536	0.3875	-0.1814	0.6673	0.4348

LiveCodeBench

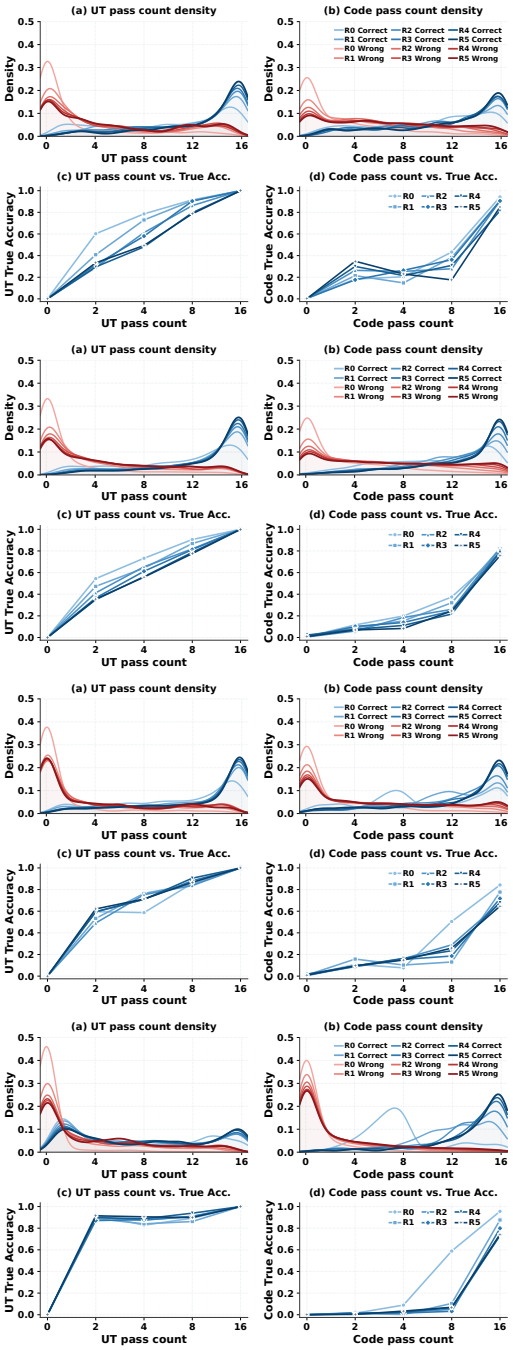
Round	$\epsilon_1 - \epsilon_2$	ρ_C	$\rho_C^\star$	ρ_T	$\rho_T^\star$
R0	0.2467	0.2498	-0.3274	0.6004	0.1502
R1	0.2462	0.2922	-0.3267	0.7066	0.3012
R2	0.2360	0.3076	-0.3090	0.7279	0.3590
R3	0.2425	0.3176	-0.3202	0.7492	0.3935
R4	0.2221	0.3262	-0.2856	0.7592	0.4298
R5	0.2323	0.3324	-0.3025	0.7696	0.4374

CodeContests

Round	$\epsilon_1 - \epsilon_2$	ρ_C	$\rho_C^\star$	ρ_T	$\rho_T^\star$
R0	0.2485	0.1913	-0.3306	0.5329	0.0991
R1	0.2243	0.2232	-0.2891	0.6905	0.2421
R2	0.2008	0.2379	-0.2513	0.7086	0.2974
R3	0.1996	0.2464	-0.2493	0.7344	0.3086
R4	0.2292	0.2548	-0.2974	0.7331	0.2954
R5	0.2073	0.2604	-0.2615	0.7437	0.3174

CodeForces

Round	$\epsilon_1 - \epsilon_2$	ρ_C	$\rho_C^\star$	ρ_T	$\rho_T^\star$
R0	0.1532	0.0398	-0.1810	0.4397	0.0314
R1	0.0538	0.0509	-0.0569	0.7697	0.1561
R2	0.0183	0.0572	-0.0187	0.8356	0.2060
R3	0.0206	0.0620	-0.0210	0.8339	0.2200
R4	0.0430	0.0653	-0.0449	0.8445	0.2159
R5	0.0275	0.0679	-0.0282	0.8497	0.2414



Round-wise statistics. Table 4 reports the quantities in Eq. 10, estimated from the execution matrix of each self-play round, with GT labels used only for this analysis. The gap $\varepsilon_1 - \varepsilon_2$ is positive on every dataset and in every round. An incorrect code is therefore more likely to pass a correct UT than an incorrect one: on a given input, it returns the correct answer more often than the specific wrong answer that an incorrect UT expects. This is the empirical form of the assumption that the

model does not systematically lean toward any wrong behavior, which also underlies cluster selection (Section 2.4). A positive gap makes $\rho_C^\star$ negative, so the UT-side condition holds regardless of code accuracy, even on CodeForces, where very few codes are correct.

The code side is harder, because a correct code fails every incorrect UT while an incorrect code may pass it; code pass counts are therefore informative only when correct UTs are common enough ($\rho_T > \rho_T^\star$). Stage 1 makes this hold already at Round 0 with a clear margin. Self-play then raises the bar, since repaired codes that remain wrong tend to be near misses that pass more UTs of both kinds, so $\rho_T^\star$ increases over rounds. UT refreshing raises ρ_T at the same time and keeps it well above $\rho_T^\star$ in every round, so code pass counts stay informative throughout self-play. In the pass-count plots, correct UTs and codes sit at high pass counts in every round, and true accuracy generally increases with pass count, confirming both conditions.

A.1.2 Support-Count Posterior

Each object to be judged is evaluated by m sources. The i-th source returns a binary support result $X_i \in \{0, 1\}$, and the support count is $S = \sum_{i=1}^{m} X_i$. If these results are independent and identically distributed under each hypothesis, then S follows the corresponding binomial distribution. In the notation of Section A.1.1, the UT side takes $m = N_c$, $S = p_j^{\mathrm{UT}}$, $H^\pm = H_T^\pm$, $(q_1, q_0) = (\theta_1, \theta_0)$ and prior ρ_T, while the code side takes $m = N_t$, $S = p_i^{\mathrm{code}}$, $H^\pm = H_C^\pm$, $(q_1, q_0) = (\varphi_1, \varphi_0)$ and prior ρ_C; the condition $q_1 > q_0$ is then exactly $\Delta_U > 0$ or $\Delta_C > 0$ in Eq. (10).

Theorem A.1 (Posterior direction of support counts)**.** *Suppose the support count S satisfies*

$$S \mid H^+ \sim \mathrm{Binomial}(m, q_1), \qquad S \mid H^- \sim \mathrm{Binomial}(m, q_0),$$

where H^+ is the hypothesis that the object is correct, H^- is the hypothesis that the object is incorrect, q_1 is the probability that a correct object is supported by one evaluator, q_0 is the probability that an incorrect object is supported by one evaluator, and $0 < q_0, q_1 < 1$. If $q_1 > q_0$, then a larger support count implies a larger posterior probability of correctness. If $q_1 = q_0$, the support count does not change the posterior. If $q_1 < q_0$, then a larger support count implies a smaller posterior probability of correctness. The posterior odds ratio changes exponentially with the support count.

Proof. Let the prior be $\rho = \mathbb{P}(H^+) \in (0, 1)$. By Bayes' rule:

$$\mathrm{OR}(s) := \frac{\mathbb{P}(H^+ \mid S = s)}{\mathbb{P}(H^- \mid S = s)} = \frac{\rho}{1-\rho} \frac{\binom{m}{s} q_1^s (1-q_1)^{m-s}}{\binom{m}{s} q_0^s (1-q_0)^{m-s}} = \frac{\rho}{1-\rho} \left(\frac{1-q_1}{1-q_0} \right)^m \kappa^s, \kappa := \frac{q_1(1-q_0)}{q_0(1-q_1)}. \tag{11}$$

The posterior probability and the odds ratio satisfy

$$\mathbb{P}(H^+ \mid S = s) = \frac{\mathrm{OR}(s)}{1 + \mathrm{OR}(s)}. \tag{12}$$

The function $x \mapsto x/(1+x)$ is strictly increasing on $x > 0$, and

$$\kappa > 1 \iff q_1(1-q_0) > q_0(1-q_1) \iff q_1 > q_0.$$

Thus the sign of $q_1 - q_0$ determines the posterior direction:

1. If $q_1 > q_0$, then $\kappa > 1$. Both the posterior odds ratio and the posterior probability of correctness are strictly increasing in s.
2. If $q_1 = q_0$, then $\kappa = 1$. The support count changes neither the posterior odds ratio nor the posterior probability of correctness.
3. If $q_1 < q_0$, then $0 < \kappa < 1$. Both the posterior odds ratio and the posterior probability of correctness are strictly decreasing in s.

Moreover, Eq. (11) shows that the posterior odds ratio depends on s through the exponential factor κ^s. □

Corollary A.2 (Posterior convergence at a fixed support ratio)**.** *Under the binomial model in Theorem A.1, let m be the number of independent evaluators, and let $s = \lfloor \eta m \rfloor$, where $\eta \in [0, 1]$ is fixed. If $q_1 \neq q_0$, then there exists a unique threshold $\eta^\star$ between q_0 and q_1. As $m \to \infty$, any fixed support ratio on the side biased toward q_1 makes the posterior probability of correctness converge exponentially fast to 1, while any fixed support ratio on the other side makes it converge exponentially fast to 0. If $q_1 = q_0$ or $\eta = \eta^\star$, no exponential decision is obtained.*

Proof. Let the prior be $\rho = \mathbb{P}(H^+) \in (0,1)$, and write $\kappa := \frac{q_1(1-q_0)}{q_0(1-q_1)}$. Let $\delta_m = s - \eta m$. Since $s = \lfloor \eta m \rfloor$, we have $|\delta_m| \le 1$. By Eq. (11),

$$\begin{aligned}
\log \mathrm{OR}(s) &= \log \frac{\rho}{1-\rho} + m \log \frac{1-q_1}{1-q_0} + s \log \kappa \\
&= \log \frac{\rho}{1-\rho} + m \left[\log \frac{1-q_1}{1-q_0} + \eta \log \kappa \right] + \delta_m \log \kappa \\
&= \log \frac{\rho}{1-\rho} + m \left[\log \frac{1-q_1}{1-q_0} + \eta \log \frac{q_1(1-q_0)}{q_0(1-q_1)} \right] + O(1) \\
&= \log \frac{\rho}{1-\rho} + m \left[\eta \log \frac{q_1}{q_0} + (1-\eta) \log \frac{1-q_1}{1-q_0} \right] + O(1) \\
&= \log \frac{\rho}{1-\rho} + mD(\eta) + O(1).
\end{aligned}$$

Here

$$D(\eta) := \eta \log \frac{q_1}{q_0} + (1-\eta) \log \frac{1-q_1}{1-q_0}. \tag{13}$$

If $q_1 = q_0$, then $D(\eta) \equiv 0$. We now assume $q_1 \neq q_0$. By Eq. (13),

$$D(q_0) = -\,\mathrm{KL}\left(\mathrm{Bern}(q_0) \| \,\mathrm{Bern}(q_1)\right) < 0, \qquad D(q_1) = \mathrm{KL}\left(\mathrm{Bern}(q_1) \| \,\mathrm{Bern}(q_0)\right) > 0.$$

Thus D is a nonconstant linear function, and its unique zero lies between q_0 and q_1. Solving $D(\eta^\star) = 0$ in Eq. (13) gives

$$\eta^\star := \frac{\log\{(1-q_0)/(1-q_1)\}}{\log \kappa}.$$

Finally, by Eq. (12),

$$\mathbb{P}(H^+ \mid S = s) = \frac{1}{1 + \exp\{-\log \mathrm{OR}(s)\}} = \frac{1}{1 + \exp\{-mD(\eta) + O(1)\}}.$$

Moreover, $D'(\eta) = \log r$. We distinguish three cases:

1. If $q_1 > q_0$, then D is strictly increasing. Hence, when $\eta > \eta^\star$, $D(\eta) > 0$, and the posterior probability of correctness converges exponentially fast to 1. When $\eta < \eta^\star$, $D(\eta) < 0$, and the posterior probability of correctness converges exponentially fast to 0.
2. If $q_1 < q_0$, then D is strictly decreasing, so the sign direction is reversed. Hence, when $\eta < \eta^\star$, the posterior probability of correctness converges exponentially fast to 1, and when $\eta > \eta^\star$, it converges exponentially fast to 0.
3. If $\eta = \eta^\star$, then $D(\eta) = 0$, and the posterior odds remain only at constant order. No exponential decision is obtained.

This proves the claim. □

A.2 Analysis of Output-Consensus-Based Cluster Selection

For a clean analysis, we study the tie-breaking rule of Section 2.4 on the tied codes $\mathcal{C}_{\mathrm{high}}$ under idealized assumptions, namely i.i.d. sampling and no execution failures; the rule remains effective in our experiments.

Proposition A.3 (Correctness of largest-cluster selection)**.** *In the failure-free case (deterministic codes, no failed executions), let $c_1, \ldots, c_n$ be i.i.d. from a distribution Q over $\mathcal{C}_{\mathrm{high}}$ and $z_1, \ldots, z_R$ i.i.d. from the distribution $\mathcal{D}$ of $\pi(\mathsf{input}; \mathcal{P})$, shared by all candidates and independent of them, and cluster the candidates by identical signatures $\sigma(c) = (c(z_1), \ldots, c(z_R))$. A* behavior *is a class of codes that agree on $\mathcal{D}$-almost every input; suppose there are finitely many, the correct one C^+ with mass ρ and incorrect ones $K_1, \ldots, K_L$ with masses $q_1, \ldots, q_L$. Codes with the same behavior share a signature almost surely, whereas codes with different behaviors share one only if they happen to agree on all R inputs. Let $\gamma := \max_\ell q_\ell$, let $\beta < 1$ be the largest probability that two different*

behaviors agree on a random input, and let Success *be the event that clustering succeeds: the correct candidates form a cluster of their own, with no incorrect code mixed in, and every other cluster is strictly smaller. Whenever* Success *occurs, the rule of Section 2.4 selects this cluster, since it picks the largest cluster in the failure-free case, and thus submits a correct code.*

(a) *If $\rho > \gamma$, then*

$$\mathbb{P}(\text{Success}) \;\geq\; 1 - \binom{L+1}{2}\beta^R - L\,e^{-n(\rho-\gamma)^2/2} \;\xrightarrow{n,R\to\infty}\; 1.$$

(b) *If $q_\ell > \rho$ for some ℓ, then, uniformly in R,*

$$\mathbb{P}(\text{Success}) \;\leq\; e^{-n(q_\ell-\rho)^2/2} \;\xrightarrow{n\to\infty}\; 0.$$

Proof. Let

$$N^+ := \big|\{i : c_i \in C^+\}\big|, \qquad N_\ell := \big|\{i : c_i \in K_\ell\}\big|, \qquad B := \bigcup_{\substack{i<j:\ c_i,c_j \text{ in} \\ \text{different behaviors}}} \big\{\sigma(c_i) = \sigma(c_j)\big\},$$

where B is the event of a collision. Two different behaviors collide only if they agree on all R independent inputs, which has probability at most β^R; a union bound over the $\binom{L+1}{2}$ pairs of behaviors gives

$$\mathbb{P}(B) \;\leq\; \binom{L+1}{2}\beta^R. \tag{14}$$

Without a collision, the clusters are exactly the behaviors present, of sizes N^+ and N_ℓ; if moreover $N^+ > N_\ell$ for all ℓ, the correct cluster is the unique largest one. Hence

$$\{\text{no collision}\} \;\cap\; \bigcap_{\ell=1}^{L}\{N^+ > N_\ell\} \;\subseteq\; \text{Success},$$

where the inclusion may be strict, so (a) is only a lower bound. The counts are related by

$$N^+ - N_\ell = \sum_{i=1}^{n}\big(\mathbf{1}\{c_i \in C^+\} - \mathbf{1}\{c_i \in K_\ell\}\big),$$

a sum of n i.i.d. terms in $[-1,1]$ with mean $\rho - q_\ell$. Since $\{N_\ell \geq N^+\} = \{N^+ - N_\ell \leq 0\}$, Hoeffding's inequality gives, whenever $\rho > q_\ell$,

$$\mathbb{P}\big(N_\ell \geq N^+\big) \;\leq\; e^{-n(\rho-q_\ell)^2/2} \;\leq\; e^{-n(\rho-\gamma)^2/2}. \tag{15}$$

A union bound then yields

$$\mathbb{P}(\text{Success}) \;\geq\; 1 - \mathbb{P}(B) - \sum_{\ell=1}^{L}\mathbb{P}\big(N_\ell \geq N^+\big),$$

which with Eqs. (14) and (15) gives (a).

For (b), we show that selecting the correct cluster requires the correct candidates to outnumber K_ℓ, which is unlikely because K_ℓ is more common. First, the candidates in K_ℓ share a behavior and hence a signature, so they form one cluster with at least N_ℓ members that is not the correct cluster. Since Success requires the correct cluster to be strictly larger than every other cluster,

$$\text{Success} \implies N^+ \;>\; \big|\text{cluster containing } K_\ell\big| \;\geq\; N_\ell, \qquad \text{i.e.} \qquad \text{Success} \subseteq \{N^+ > N_\ell\}.$$

Second, since $q_\ell > \rho$, the count N_ℓ exceeds N^+ on average: the difference

$$N_\ell - N^+ = \sum_{i=1}^{n}\big(\mathbf{1}\{c_i \in K_\ell\} - \mathbf{1}\{c_i \in C^+\}\big)$$

is a sum of n i.i.d. terms in $[-1,1]$ with mean $n(q_\ell - \rho) > 0$. The event $\{N^+ > N_\ell\}$ requires this sum to fall below zero, i.e., more than $n(q_\ell - \rho)$ below its mean, and Hoeffding's inequality bounds the probability of such a deviation:

$$\mathbb{P}(\text{Success}) \;\leq\; \mathbb{P}\big(N^+ > N_\ell\big) \;\leq\; e^{-n(q_\ell-\rho)^2/2}.$$

Both steps only count candidates and never use the inputs $z_1, \ldots, z_R$; more inputs separate different behaviors but cannot split K_ℓ, so the bound holds for every R. □

B Detailed Algorithm Design

Algorithm 1 The CoSPlay Framework

Input: Problem $\mathcal{P}$; frozen LLM π; pool sizes N_c, N_t; number of random inputs R; maximum rounds $T_{\max}$
Output: Selected code c^*

UT sampling $t \sim \pi_{\mathrm{UT}}(\cdot)$: sample a valid input $x \sim \pi(\mathsf{input};\, \mathcal{P}, \cdot)$, then sample $y \sim \pi(\mathsf{output};\, \mathcal{P}, x)$ four times
and accept $t = (x, y)$ only if at least three samples agree (output self-consistency); resample otherwise.
Refresh $\mathbf{M}$: execute all Code-UT pairs, with $c_i(x) = \perp$ on runtime errors or timeouts, and set $\mathbf{M}_{ij} = \mathbb{I}[c_i(x_j) = y_j]$,
$p_j^{\mathrm{UT}} = \sum_i \mathbf{M}_{ij}$, $p_i^{\mathrm{code}} = \sum_j \mathbf{M}_{ij}$, and the selections $\mathcal{C}^0, \mathcal{T}^0, \mathcal{T}^{\mathrm{all}}, t^{\pm}, \mathcal{C}^{\pm}$ in Eq. (3).

// Stage 1: Exploration-Attack-Guided Initialization

$\mathcal{P} \xrightarrow{\text{hint}} \mathcal{H}$ (hints) $\xrightarrow{\text{expand subsets } \le 2} \mathcal{S}$ (solution plans) $\xrightarrow{\text{attack}} \mathcal{A}$ (UT ideas)

$\mathcal{H} \sim \pi(\mathsf{hint};\, \mathcal{P})$
$\mathcal{S} \leftarrow \{s \sim \pi(\mathsf{plan};\, \mathcal{P}, \omega) \mid \omega \subseteq \mathcal{H},\ |\omega| \le 2\}$
$\mathcal{A} \leftarrow \{a \sim \pi(\mathsf{attack};\, \mathcal{P}, s) \mid s \in \mathcal{S}\}$
$\mathcal{C} \leftarrow \{c_i \sim \pi(\mathsf{code};\, \mathcal{P}, s_i)\}_{i=1}^{N_c}$, with s_i taken once each from the shuffled $\mathcal{S}$
$\mathcal{T} \leftarrow \{t_j\}_{j=1}^{N_t}$: half $t_j \sim \pi_{\mathrm{UT}}(\varnothing)$ and half $t_j \sim \pi_{\mathrm{UT}}(a_j)$, with a_j from the shuffled $\mathcal{A}$
Refresh $\mathbf{M}$ and its statistics

// Stage 2: Execution-Matrix-Driven Iterative Self-Play

Loop rules: Keep pool sizes fixed and skip steps without eligible targets. After each step, refresh $\mathbf{M}$ and its statistics, and terminate self-play immediately if all $\mathbf{M}_{ij} = 1$.

for $T = 1$ **to** $T_{\max}$ **while** some $\mathbf{M}_{ij} = 0$ **do**
  // Step 1: Code Cleaning
  $\mathcal{C} \leftarrow (\mathcal{C} \setminus \mathcal{C}^0) \cup \{c_k' \sim \pi(\mathsf{code};\, \mathcal{P}, s_k) \mid s_k \in \mathcal{S}\}_{k=1}^{|\mathcal{C}^0|}$
  // Step 2: Breaking Spurious Code-UT Coupling
  $\mathcal{T} \leftarrow (\mathcal{T} \setminus \{t^-\}) \cup \{t' \sim \pi_{\mathrm{UT}}(a,\, t^-,\, \mathcal{C}^-)\},\ a \in \mathcal{A}$
    (t^- and $\mathcal{C}^-$ serve only as a negative reference marked "may contain errors")
  // Step 3: Code Fixing
  $\mathcal{C} \leftarrow (\mathcal{C} \setminus \mathcal{C}^+) \cup \{c_i' \sim \pi(\mathsf{fix};\, \mathcal{P}, c_i, t^+, c_i(x^+)) \mid c_i \in \mathcal{C}^+\}$
  // Step 4: Replacing Zero-Discrimination UTs
  $\mathcal{T} \leftarrow \big(\mathcal{T} \setminus (\mathcal{T}^0 \cup \mathcal{T}^{\mathrm{all}})\big) \cup \{t_k' \sim \pi_{\mathrm{UT}}(a_k) \mid a_k \in \mathcal{A}\}_{k=1}^{|\mathcal{T}^0 \cup \mathcal{T}^{\mathrm{all}}|}$
end for

// Stage 3: Output-Consensus-Based Cluster Selection

$\mathcal{C}_{\mathrm{high}} \leftarrow \{c_i \in \mathcal{C} \mid p_i^{\mathrm{code}} = \max_\ell p_\ell^{\mathrm{code}}\}$
Draw $z_1, \ldots, z_R \sim \pi(\mathsf{input};\, \mathcal{P})$ and compute $\sigma(c_i) = (c_i(z_1), \ldots, c_i(z_R))$ for all $c_i \in \mathcal{C}_{\mathrm{high}}$
for each $c_i \in \mathcal{C}_{\mathrm{high}}$ in a fixed order **do**
  Add c_i to the first cluster G_m whose members c_j all satisfy $c_i(z_r) = c_j(z_r)$ whenever $c_i(z_r), c_j(z_r) \neq \perp$;
  otherwise open a new cluster $\{c_i\}$
end for
$w(c_i) \leftarrow \sum_{c_j \in G_m,\, j \neq i} \sum_{r=1}^{R} \mathbb{I}[c_i(z_r) = c_j(z_r) \neq \perp]$ for each $c_i \in G_m$
$G^* \leftarrow \arg\max_{G_m} \sum_{c_i \in G_m} w(c_i), \quad c^* \leftarrow \arg\max_{c_i \in G^*} w(c_i)$
return c^*

C Related Work

C.1 Fine-tuning Methods for Code Generation

For code generation, most existing training-based methods rely heavily on external ground-truth data, approaches like μCode (Jain et al., 2025a) and AceCoder (Zeng et al., 2025) employ Supervised Fine-Tuning using ground-truth or synthetic datasets, while Focused DPO (Zhang et al., 2025a) and CTRL (Xie et al., 2025) utilize RLVR by optimizing rewards design. Similarly, O1-Coder (Zhang et al., 2024) and UTGen (Prasad et al., 2025) fine-tune models specifically for UT generation by leveraging ground-truth code. Some recent approaches aim to co-evolve these capabilities using RLVR but often retain critical dependencies: UTRL (Lee et al., 2025) requires ground-truth code, and CURE (Wang et al., 2025b) relies on ground-truth UT. While Absolute-Zero (Zhao et al., 2025a) achieves purely self-supervised learning without external data, it demands prohibitive computational resources for RLVR training.

C.2 TTS for LLM Code Generation

Test-Time Scaling (TTS) strategies typically begin with repeated sampling, such as Best-of-N (BoN), but they often struggle to identify the best candidate without GT data. To improve selection, consensus-based methods such as CodeT (Chen et al., 2023) and MPSC (Huang et al., 2024) use self-generated UTs for voting, although CodeT is limited by the potential unreliability of generated tests and MPSC incurs substantial overhead from extensive sampling. More broadly, sampling-based inference-time methods such as PowerSampling improve reasoning by drawing from sharpened sequence-level distributions without additional training, suggesting that stronger performance can sometimes be elicited purely through better test-time sampling (Karan & Du, 2025). Beyond simple resampling, a growing line of work treats code generation as a structured search problem. SFS formulates code generation as black-box optimization in code space and improves exploration diversity to avoid local optima (Light et al., 2025), while CodeTree organizes planning, generation, and debugging within a unified tree structure guided by both execution feedback and agent feedback (Li et al., 2025b). ThinkCoder further decomposes test-time computation into a thorough exploration stage followed by an optimal refinement stage, improving efficiency by first broadening the solution space and then sharpening promising candidates (Zhang et al., 2025b). Finally, execution-guided approaches leverage runtime feedback for iterative improvement. Methods such as ORPS (Yu et al., 2025b), S* (Li et al., 2025a), LDB (Zhong et al., 2024), and Reflexion (Shinn et al., 2023) use execution signals for process supervision or refinement. Another paper (Chen et al., 2025) further shows that self-generated tests can support iterative debugging, while also highlighting the bias and reliability issues that arise when such tests are used naively.

C.3 Self-Play in LLM

The self-play paradigm, popularized by AlphaZero (Silver et al., 2017) and AlphaGo (Silver et al., 2016), has recently been adapted to optimize LLMs via RL. However, most existing approaches in the coding domain remain dependent on external supervision and heavy training. For instance, CURE (Wang et al., 2025b) relies on GT UT to guide the co-evolution of generation capability of code and UT, while UTRL (Lee et al., 2025) necessitates ground-truth code for improving model performance. Although Absolute-Zero (Zhao et al., 2025a) achieves a fully self-supervised setup without human data, it yields limited performance gains despite the prohibitive training costs of RLVR training. Crucially, these methods are exclusively training-time optimizations. In contrast, CoSPlay explores inference-time self-play for GT-free code generation, leveraging the pass count in execution metrix as signals to iteratively refine and filter candidates without parameter updates.

C.4 Search in Natural Language Space

To enhance the reasoning capabilities of LLMs, recent studies have proposed Chain-of-Thought (Wei et al., 2022) strategies into structured search processes within the natural language space. Frameworks such as Tree of Thoughts (Yao et al., 2023) and Reasoning via Planning (Hao et al., 2023) treat reasoning steps as a search tree, exploring various intermediate thoughts to reach a solution. In the specific domain of code generation, methods like PlanSearch (Wang et al., 2025a) demonstrate that

generating high-level natural language plans before implementation significantly outperforms direct code generation. The primary benefit of such natural language search lies in its ability to introduce structural diversity, preventing the model from converging to a narrow set of repetitive solutions—a common issue in direct code resampling. However, most existing methods for code generation remain static, generating plans without leveraging dynamic execution feedback or iterative self-play to refine.

D Detailed baseline setting

RL Baselines. Detailed data usage and base models are listed in Table 5. We follow the original prompts and LLM parameters as provided in each method's code or paper.

TTS Baselines. For a fair comparison, we control the total code and UT generation budgets to be comparable across methods, and all UTs are generated by the model itself without external supervision. Specific settings are as follows:

(1) SFS (Light et al., 2025): num_seed = 10, max_iters = 40 (total budget of 50) and we scale to 80, generated UT = 6; we use the `Role` seed theme, the best-performing variant reported in the original paper. The sampling temperature is 0.2.

(2) CodeTree (Li et al., 2025b): full `agent` setting with max budget = 50, generated UT = 6, max depth = 5, search width = 10. The scaled-up setting with max budget = 1000, generated UT = 16, max depth = 20, search width = 100. The sampling temperature is 0.0.

(3) MPSC (Huang et al., 2024): solution = 60, specification = 50, test case = 100; we use `MPSC-label` mode, the best-performing mode for code generation reported in the original paper. The scaled-up setting with solution = 300, specification = 250, test case = 600. The sampling temperature is 0.8.

(4) PowerSampling (Karan & Du, 2025): n_samples = 16, mcmc_steps = 10, code sampling and UT generation temperature = 0.6.

(5) ThinkCoder (Zhang et al., 2025b): $k = 20$ per iteration, each candidate prompted to generate 3 UTs, temperature is 1.0. budget n to 2, and we scale to 5/10/20

(6) S* (Li et al., 2025a): we use the local S* pipeline with generated UT = 16, $N = 16$ code samples per round, `num_round` $= 3$ and we scale to 6. The sampling temperature is 0.7.

(7) BoN: we set N = 16/64/128/256 for both code and UT.

Table 5: Training Data Source and Base Model for baselines.

Model	Base model	Data	training
AceCoder-Rule-7B	Qwen2.5-7B-Coder-Instruct	22k code data	Yes
AceCoder-RM-7B	Qwen2.5-7B-Coder-Instruct	22k code data + RM(307k data)	Yes
Absolute-Zero-Coder-7B	Qwen-2.5-7B-Coder	**0**	Yes
CURE-7B	Qwen2.5-7B-Instruct	4.5k CodeContests data	Yes
CoSPlay-7B	Qwen2.5-7B-Instruct	**0**	**No**
Absolute-Zero-Coder-14B	Qwen-2.5-14B-Coder	**0**	Yes
CURE-14B	Qwen2.5-14B-Instruct	4.5k CodeContests data	Yes
CoSPlay-14B	Qwen2.5-14B-Instruct	**0**	**No**
AceCoder-RM-7B	Qwen2.5-Coder-7B-Instruct	307k preference pairs	Yes
AceCoder-RM-32B	Qwen2.5-Coder-32B-Instruct	307k preference pairs	Yes

E Additional Implementation Details

Execution environment and time limits. All code-UT executions run in an isolated CPU sandbox. Following the CURE benchmark suite (Wang et al., 2025b), a single execution is limited to 8 seconds on LiveBench and LiveCodeBench and to 1 second on CodeContests and CodeForces. The limit

Table 6: **Detailed TTS computational cost breakdown on Qwen2.5-7B-Instruct [3×200].** The columns are defined as follows: **Method** indicates the evaluated baseline or our proposed method; **Code** and **UT** denote the average numbers of generated code candidates and unit tests per problem, respectively; **Avg. Exec.** is the average number of code-UT executions per problem, i.e. the number of (code, unit test) pairs actually run in the sandbox; **Avg. Call** and **Avg. Token** measure the average number of LLM calls and generated tokens across all evaluated problems; **Succ. Call** and **Succ. Token** report the corresponding averages over successfully solved problems. Finally, **Acc** is the accuracy of the single program each method submits; for CoSPlay variants, it is reported after applying the cluster step. **Bold** means the best performance.

Method	Code	UT	Avg. Exec.	Avg. Call	Avg. Token	Succ. Call	Succ. Token	Acc
Qwen2.5-7B-Instruct								
Direct	1.0	0.0	0.0	1.0	628.3	1.0	515.6	22.7
BoN (N=16)	16.0	16.0	256.0	32.0	24,478.1	32.0	20,160.3	26.5
BoN (N=64)	64.0	64.0	4,096.0	128.0	90,952.9	128.0	72,583.1	24.8
BoN (N=128)	128.0	128.0	16,384.0	256.0	344,674.5	256.0	260,191.6	27.2
BoN (N=256)	256.0	256.0	65,536.0	512.0	745,385.6	512.0	555,398.1	22.5
SFS (Round 20)	24.1	6.0	144.6	43.7	17,501.6	33.2	11,634.4	31.0
SFS (Round 40)	37.3	6.0	223.8	69.0	28,677.3	43.4	15,823.4	31.8
SFS (Round 80)	55.2	6.0	331.2	122.0	50,951.9	100.8	45,871.3	32.0
S* (Round 3)	48.0	16.0	767.7	70.7	23,973.3	67.8	15,707.0	20.5
S* (Round 6)	96.0	16.0	1,536.0	121.7	42,945.7	118.8	29,509.3	19.2
ThinkCoder (Round 2)	39.4	118.2	307.8	40.4	116,646.3	39.2	96,088.0	25.3
ThinkCoder (Round 5)	97.1	291.3	783.4	99.6	297,219.4	95.6	253,272.6	25.5
ThinkCoder (Round 10)	189.6	568.7	1,577.0	192.7	565,558.7	176.0	476,425.6	25.5
ThinkCoder (Round 20)	382.9	1148.6	3,169.3	386.9	1,156,593.2	361.8	971,276.7	27.8
CodeTree	15.6	5.8	90.7	38.2	13,173.5	35.3	9,068.0	15.7
CodeTree (Scale)	907.3	15.5	14,111.8	1,849.5	638,268.8	1,868.7	402,823.6	14.3
MPSC	60.0	99.5	100,861.1	6.8	104,709.6	6.9	97,773.8	25.3
MPSC (Scale)	300.0	478.4	2,516,750.0	18.8	661,763.8	18.9	612,221.5	23.5
PowerSampling	21.2	0.0	0.0	49.3	14,762.1	41.2	9,957.4	25.0
PowerSampling + BoN	338.7	338.7	256.0	1,477.2	416,238.5	1,264.3	303,952.7	27.5
CoSPlay (Ours)	38.5	206.0	6,912.0	776.1	682,292.9	603.4	479,437.9	**37.2**
w/o self-cons.	41.0	74.9	6,912.0	395.3	243,800.4	374.4	208,419.8	34.2
w/o self-cons. & UT-iteration	41.4	24.0	3,072.0	198.4	90,291.4	183.8	68,641.2	32.5

Table 7: **Detailed TTS cost breakdown on Qwen2.5-14B-Instruct [3×200].** The column definitions follow Table 6.

Method	Code	UT	Avg. Exec.	Avg. Call	Avg. Token	Succ. Call	Succ. Token	Acc
Qwen2.5-14B-Instruct								
Direct	1.0	0.0	0.0	1.0	568.0	1.0	472.8	26.5
BoN (N=16)	16.0	16.0	256.0	32.0	22,102.9	32.0	19,505.9	36.8
BoN (N=64)	64.0	64.0	4,096.0	128.0	86,830.2	128.0	77,246.0	38.8
BoN (N=128)	128.0	128.0	16,384.0	256.0	328,727.6	256.0	293,841.5	40.5
BoN (N=256)	256.0	256.0	65,536.0	512.0	710,314.1	512.0	635,587.2	40.3
SFS (Round 20)	26.8	6.0	160.8	50.6	19,341.6	47.1	15,658.6	36.3
SFS (Round 40)	42.8	6.0	256.8	76.3	31,629.9	68.1	24,228.1	37.9
SFS (Round 80)	60.7	6.0	364.2	131.9	56,802.1	157.7	59,382.7	38.7
S* (Round 3)	48.0	16.0	767.8	51.3	23,787.8	51.1	18,855.8	31.7
S* (Round 6)	96.0	16.0	1,527.3	118.3	45,672.6	114.6	35,259.8	31.9
ThinkCoder (Round 2)	39.3	117.8	315.4	39.3	64,189.4	39.3	57,505.0	35.8
ThinkCoder (Round 5)	96.8	290.5	792.7	96.8	165,598.4	93.8	148,618.3	36.2
ThinkCoder (Round 10)	190.8	572.5	1,576.6	190.8	379,948.4	184.1	370,315.7	38.5
ThinkCoder (Round 20)	375.9	1127.6	3,129.6	375.9	786,704.7	361.6	798,329.0	38.7
CodeTree	14.4	5.9	85.7	33.9	12,056.6	23.4	7,078.5	22.8
CodeTree (Scale)	842.0	15.9	13,380.5	1711.4	525,068.9	1552.2	402,579.9	25.6
MPSC	60.0	93.8	95,013.1	6.5	126,328.7	6.8	142,208.8	27.7
MPSC (Scale)	300.0	412.7	2,361,252.5	18.4	710,164.0	18.9	647,139.1	31.7
PowerSampling	18.2	0.0	0.0	43.1	11,451.3	37.2	8,488.4	27.2
PowerSampling + BoN	282.6	282.6	256.0	1273.2	320,064.1	1,084.4	242,852.4	34.8
CoSPlay (Ours)	36.2	171.3	6,912.0	763.7	838,745.7	664.8	748,581.2	**45.0**
w/o self-cons.	39.0	83.7	6,912.0	328.4	254,158.5	314.8	232,620.7	43.7
w/o self-cons. & UT-iteration	41.8	24.0	3,072.0	202.9	111,134.0	184.8	89,815.0	41.5

applies to every individual code-input execution, covering self-generated UTs and ground-truth tests alike. A program that exceeds its limit is terminated and recorded as a Timeout Error, which counts as a failed execution.

Ground-truth evaluation. When measuring true correctness, each code is executed against at most eight ground-truth tests per problem. The same cap applies to every method, backbone, and dataset, so all reported comparisons remain on a common footing.

Decoding configuration. Open-weight backbones are served with vLLM (Kwon et al., 2023) on two NVIDIA H100 (80GB) GPUs, with temperature 0.8, top-p 0.95, top-k 40. Gemini-2.0-Flash is accessed through its official API, where temperature and top-p are kept identical and top-k is left at the provider default because it is not exposed.

Non-trivial UT refinement. At each self-play round, we compute the pass rate of every valid UT and exclude trivial UTs with pass rates of 0 or 1. We then identify the non-trivial UTs with the lowest pass rate. If multiple UTs are tied at this minimum, we resample only one of them. Such a tie may indicate that one code is more reliable than the others, although one of the tied UTs may still be noisy or problematic. Resampling a single tied candidate therefore provides a minimal diagnostic intervention: it tests whether the UT itself should be corrected while preserving the code-quality signal and avoiding the unnecessary replacement of multiple potentially valid UTs.

Pool sizes and budgets. We set the code pool size $N_c = 16$, the UT pool size $N_t = 16$, the maximum number of self-play rounds $T_{\max} = 5$, and the number of random clustering inputs $R = 16$. Within the subset space Ω, we keep at most four singleton subsets ($|\omega| = 1$), while the number of size-two subsets and the number of attack ideas derived from each plan are left unconstrained and determined by the model. Half of the initial UT inputs are drawn from the attack idea pool $\mathcal{A}$ and half are random valid inputs taken directly from the problem statement. For every UT input we sample four expected outputs and keep the UT only when at least three agree.

Failure handling. When a response contains no parsable program, we resample up to five times before recording the candidate as failed. Runtime errors, non-zero exit codes, and timeouts are all mapped to a failed execution and are never treated as passes.

Backbone-specific deviation for OCR-Nemotron-1.1-7B. OCR-Nemotron-1.1-7B produces long chain-of-thought traces, so the full two-stage UT construction is prohibitively expensive on it: every expected output would require four independent long-form generations for the self-consistency check. For this backbone only, we generate each UT in a single pass directly from the problem and the attack idea, without the separate expected-output stage and without the self-consistency filter. All other components, pool sizes, and budgets are unchanged.

F Analysis of the cost and performance balance for TTS methods

Table 8 reports the per-dataset Acc results of each TTS method, while Table 6 and Table 7 summarize their computational costs on Qwen2.5-7B-Instruct and Qwen2.5-14B-Instruct, respectively, where **Avg. Exec.** denotes the average number of code-UT pairs actually executed per problem. Overall, CoSPlay achieves the best Acc accuracy on both model scales, improving the strongest competing TTS baseline from 32.0% to 37.2% on 7B and from 40.5% to 45.0% on 14B. More importantly, this gain is not obtained by simply sampling more programs. On 7B, several scaled baselines use comparable or even larger token budgets than CoSPlay but achieve much lower accuracy: BoN (N=256) uses 745K tokens and obtains 22.5%, MPSC (Scale) uses 662K tokens and obtains 23.5%, CodeTree (Scale) uses 638K tokens and obtains 14.3%, and ThinkCoder (Round 20) uses 1.16M tokens but only reaches 27.8%. In contrast, CoSPlay uses 682K tokens and reaches 37.2%, showing that its advantage comes from more effective use of test-time computation rather than brute-force scaling.

The execution counts lead to the same conclusion along a third cost axis. Baselines relying on exhaustive cross-execution scale poorly: BoN executes N^2 code-UT pairs, growing from 256 executions at N=16 to 65,536 at N=256 with no monotonic gain, while MPSC (Scale), despite a token budget close to CoSPlay's on 7B (662K vs. 682K), performs 2.52M executions—roughly $364\times$ more—yet its Acc drops to 23.5%. In contrast, CoSPlay attains the best accuracy on both backbones with a small fixed budget of 6,912 executions, whereas the zero-execution baselines (Direct and

Table 8: **Performance comparison of TTS methods [3×200]**. **Acc.** reports the accuracy of the final submitted solution. **Code** and **UT** denote the average numbers of sampled code candidates and sampled unit tests, respectively. **Avg.** represents the average performance across all four datasets. **Bold** means the best performance.

Method	Venue	LB			LCB			CC			CF			Avg.		
		Acc.	#Code	#UT	Acc.	#Code	#UT	Acc.	#Code	#UT	Acc.	#Code	#UT	Acc.	#Code	#UT
Qwen2.5-7B-Instruct																
Direct	-	36.0	1.0	0.0	27.1	1.0	0.0	21.3	1.0	0.0	6.5	1.0	0.0	22.7	1.0	0.0
BoN (N=16)	-	38.0	16.0	16.0	32.7	16.0	16.0	25.3	16.0	16.0	10.0	16.0	16.0	26.5	16.0	16.0
BoN (N=64)	-	37.3	64.0	64.0	27.3	64.0	64.0	26.0	64.0	64.0	8.7	64.0	64.0	24.8	64.0	64.0
BoN (N=128)	-	44.0	128.0	128.0	30.0	128.0	128.0	24.7	128.0	128.0	10.0	128.0	128.0	27.2	128.0	128.0
BoN (N=256)	-	36.0	256.0	256.0	24.0	256.0	256.0	24.0	256.0	256.0	6.0	256.0	256.0	22.5	256.0	256.0
MPSC	ACL24 Long	39.3	60.0	99.2	30.7	60.0	99.2	23.3	60.0	99.6	8.0	60.0	100.0	25.3	60.0	99.5
MPSC (Scale)	ACL24 Long	41.3	300.0	392.7	29.3	300.0	406.5	18.7	300.0	546.6	4.7	300.0	567.6	23.5	300.0	478.4
CodeTree	NAACL25 Findings	18.0	15.0	5.9	14.0	16.3	5.9	22.7	16.1	5.8	8.0	15.0	5.8	15.7	15.6	5.8
CodeTree (Scale)	NAACL25 Findings	15.0	973.8	15.7	13.3	933.6	15.7	20.7	924.6	15.4	8.1	797.2	15.3	14.3	907.3	15.5
SFS (Round 20)	ICLR25 Poster	38.0	21.4	6.0	38.0	19.7	6.0	33.3	25.8	6.0	14.7	29.4	6.0	31.0	24.1	6.0
SFS (Round 40)	ICLR25 Poster	38.0	31.9	6.0	40.0	27.6	6.0	34.7	41.0	6.0	14.7	48.6	6.0	31.8	37.3	6.0
SFS (Round 80)	ICLR25 Poster	38.0	33.5	6.0	40.0	32.0	6.0	34.7	69.8	6.0	15.3	85.5	6.0	32.0	55.2	6.0
S* (Round 3)	EMNLP 2025 Findings	27.3	48.0	16.0	26.7	48.0	16.0	20.7	48.0	16.0	7.3	48.0	16.0	20.5	48.0	16.0
S* (Round 6)	EMNLP 2025 Findings	26.0	96.0	16.0	26.0	96.0	16.0	20.0	96.0	16.0	4.7	96.0	16.0	19.2	96.0	16.0
ThinkCoder (Round 2)	ACL25 Findings	36.7	39.2	117.6	32.7	38.9	116.8	27.3	39.5	118.4	4.7	40.0	120.0	25.3	39.4	118.2
ThinkCoder (Round 5)	ACL25 Findings	36.7	96.4	289.2	33.3	94.4	283.2	26.7	97.6	292.8	5.3	100.0	300.0	25.5	97.1	291.3
ThinkCoder (Round 10)	ACL25 Findings	38.0	191.1	573.2	34.0	185.7	557.2	24.7	181.5	544.4	5.3	200.0	600.0	25.5	189.6	568.7
ThinkCoder (Round 20)	ACL25 Findings	38.7	378.9	1136.8	36.0	366.9	1100.8	29.3	387.6	1162.8	7.3	398.0	1194.0	27.8	382.9	1148.6
PowerSampling	ICLR26 Oral	37.3	20.6	0.0	31.3	23.6	0.0	22.0	20.3	0.0	9.3	20.3	0.0	25.0	21.2	0.0
+ BoN	ICLR26 Oral	38.0	308.0	308.0	34.0	332.9	332.9	27.3	345.2	345.2	10.7	368.6	368.6	27.5	338.7	338.7
CoSPlay	-	**51.3**	28.9	194.2	**41.3**	29.6	193.6	36.0	36.7	208.7	**18.0**	58.9	227.7	36.7	38.5	206.0
+ Cluster	-	**51.3**	28.9	194.2	**41.3**	29.6	193.6	**38.0**	36.7	208.7	**18.0**	58.9	227.7	**37.2**	38.5	206.0
w/o self cons.	-	48.0	30.5	70.8	38.0	30.1	67.2	30.7	46.2	74.6	16.7	57.2	86.9	33.3	41.0	74.9
+ Cluster	-	50.0	30.5	70.8	39.3	30.1	67.2	30.7	46.2	74.6	16.7	57.2	86.9	34.2	41.0	74.9
w/o self cons. & UT-iteration	-	43.3	30.9	24.0	34.0	29.8	24.0	33.3	41.5	24.0	16.7	63.4	24.0	31.8	41.4	24.0
+ Cluster	-	44.7	30.9	24.0	34.7	29.8	24.0	34.0	41.5	24.0	16.7	63.4	24.0	32.5	41.4	24.0
Qwen2.5-14B-Instruct																
Direct	-	41.0	1.0	0.0	34.2	1.0	0.0	22.9	1.0	0.0	7.9	1.0	0.0	26.5	1.0	0.0
BoN (N=16)	-	56.0	16.0	16.0	44.0	16.0	16.0	31.3	16.0	16.0	16.0	16.0	16.0	36.8	16.0	16.0
BoN (N=64)	-	58.7	64.0	64.0	46.7	64.0	64.0	31.3	64.0	64.0	18.7	64.0	64.0	38.8	64.0	64.0
BoN (N=128)	-	60.7	128.0	128.0	45.3	128.0	128.0	34.7	128.0	128.0	21.3	128.0	128.0	40.5	128.0	128.0
BoN (N=256)	-	60.0	256.0	256.0	45.3	256.0	256.0	36.0	256.0	256.0	20.0	256.0	256.0	40.3	256.0	256.0
MPSC	ACL24 Long	42.0	60.0	99.2	33.3	60.0	88.6	23.3	60.0	99.3	12.0	60.0	88.0	27.7	60.0	93.8
MPSC (Scale)	ACL24 Long	44.7	300.0	389.8	42.7	300.0	347.0	26.7	300.0	451.7	12.7	300.0	462.4	31.7	300.0	412.7
CodeTree	NAACL25 Findings	28.7	10.7	6.0	31.3	11.7	6.0	24.7	14.0	5.9	6.7	21.4	5.9	22.8	14.4	5.9
CodeTree (Scale)	NAACL25 Findings	33.3	839.7	15.8	34.2	846.4	15.9	26.4	829.2	15.9	8.6	852.9	15.9	25.6	842.0	15.9
SFS (Round 20)	ICLR25 Poster	40.0	26.5	6.0	44.7	23.3	6.0	38.0	28.2	6.0	22.7	29.2	6.0	36.3	26.8	6.0
SFS (Round 40)	ICLR25 Poster	42.7	42.5	6.0	46.0	35.0	6.0	38.7	45.5	6.0	24.0	48.0	6.0	37.9	42.8	6.0
SFS (Round 80)	ICLR25 Poster	44.7	43.9	6.0	46.7	37.7	6.0	38.7	75.5	6.0	24.7	85.8	6.0	38.7	60.7	6.0
S* (Round 3)	EMNLP 2025 Findings	44.7	48.0	16.0	34.7	48.0	16.0	32.7	48.0	16.0	14.7	48.0	16.0	31.7	48.0	16.0
S* (Round 6)	EMNLP 2025 Findings	45.0	96.0	16.0	34.7	96.0	16.0	33.0	96.0	16.0	14.7	96.0	16.0	31.9	96.0	16.0
ThinkCoder (Round 2)	ACL25 Findings	53.3	38.8	116.4	46.0	39.5	118.4	29.3	39.2	117.6	14.7	39.6	118.8	35.8	39.3	117.8
ThinkCoder (Round 5)	ACL25 Findings	52.0	94.3	282.8	46.0	97.9	293.6	30.7	96.8	290.4	16.0	98.4	295.2	36.2	96.8	290.5
ThinkCoder (Round 10)	ACL25 Findings	55.3	186.3	558.8	46.7	189.3	568.0	34.7	191.3	574.0	17.3	196.4	589.2	38.5	190.8	572.5
ThinkCoder (Round 20)	ACL25 Findings	55.3	366.9	1100.8	46.0	369.2	1107.6	34.0	377.1	1131.2	19.3	390.3	1170.8	38.7	375.9	1127.6
PowerSampling	ICLR26 Oral	41.3	15.3	0.0	36.7	16.1	0.0	21.3	22.0	0.0	9.3	19.2	0.0	27.2	18.2	0.0
PowerSampling + BoN	ICLR26 Oral	46.0	252.2	252.2	44.0	263.9	263.9	35.3	316.1	316.1	14.0	298.4	298.4	34.8	282.6	282.6
CoSPlay	-	58.7	25.7	158.0	52.7	27.7	156.8	**41.5**	35.5	171.0	26.0	56.0	199.5	44.7	36.2	171.3
+ Cluster	-	**59.0**	25.7	158.0	**53.0**	27.7	156.8	**41.5**	35.5	171.0	**26.7**	56.0	199.5	**45.0**	36.2	171.3
w/o self-cons.	-	57.3	28.2	82.8	48.7	28.7	79.7	42.0	39.3	83.1	25.7	59.6	89.1	43.4	39.0	83.7
+ Cluster	-	58.0	28.2	82.8	48.7	28.7	79.7	42.0	39.3	83.1	26.0	59.6	89.1	43.7	39.0	83.7
w/o self-cons. & UT-iteration	-	56.0	29.4	24.0	49.3	32.3	24.0	32.7	40.8	24.0	26.0	64.5	24.0	41.1	41.8	24.0
+ Cluster	-	56.7	29.4	24.0	50.0	32.3	24.0	33.3	40.8	24.0	26.0	64.5	24.0	41.5	41.8	24.0

PowerSampling) lack execution-based verification and lag far behind. This shows that performance is driven not by execution volume but by spending a modest budget on co-evolving reliable codes and UTs; since executions run in a cheap, parallelizable CPU sandbox, this axis also adds little overhead to the dominant LLM cost.

The ablation results further show a favorable efficiency–performance trade-off. The full CoSPlay pipeline includes two token-consuming components: self-consistency validation for UT generation and iterative UT updating during self-play. Removing self-consistency reduces the 7B token cost from 682K to 244K while still achieving 34.2%, which remains higher than all competing TTS baselines. Further removing UT-iteration reduces the cost to 90K tokens, 198 calls, and 3,072 executions, yet

Table 9: **Incremental ablation of CoSPlay components on Qwen2.5-7B-Instruct [3×200].** Starting from direct sampling, we add one component at a time on the 200-problem benchmark (averaged over three seeds) and report the BoN, the average output tokens per problem, the marginal performance gain (Δperf.) and marginal token cost (Δtokens) over the previous row. Steps 1–4 refer to the self-play steps of Stage 2; all rows report accuracy without cluster selection except the last row, which adds it.

Setting	Output Tokens	BoN	Δperf.	Δtokens
Direct sampling (single)	628.3	22.7	–	–
BoN (N=16)	24,478.1	26.5	+3.8	23,849.8
exp-atk only	103,302.6	30.2	+3.7	78,824.5
+ self-consis.	124,109.9	33.0	+2.8	20,807.4
+ Step 1	192,468.9	34.2	+1.2	68,359.0
+ Step 1, 3 (w/o UT-iter.)	280,743.8	35.2	+1.0	88,274.9
+ Step 1, 2, 3	339,223.1	35.8	+0.6	58,479.3
+ Step 1, 2, 3, 4 (Full)	676,614.9	36.7	+0.9	337,391.8
+ Cluster	682,292.9	37.2	+0.5	5,678.1

the method still achieves 32.5%, slightly outperforming the strongest baseline SFS (Round 80) at 32.0%. A similar trend holds on 14B: the lightweight variant reduces the token cost from 839K to 111K and the number of calls from 763.7 to 202.9, while still reaching 41.5%, above the strongest competing baseline. These results suggest that even when the most expensive modules are removed, the core execution-guided design of CoSPlay remains more efficient than existing TTS strategies under comparable budgets.

This comparison is especially important under our GT-free setting. Several prior TTS methods were originally designed to benefit from reliable public tests or external feedback, but in our setting they must rely only on self-generated UTs. Without guaranteed-correct tests, their search or refinement signals can become noisy, and scaling them does not necessarily improve performance. For example, on 7B, S* drops from 20.5% at Round 3 to 19.2% at Round 6, CodeTree and CodeTree (Scale) remain below direct sampling, and scaling MPSC from 60 code candidates to 300 reduces Acc from 25.3% to 23.5%. These results indicate that naively increasing the number of generated codes, UTs, or search steps can amplify noisy self-generated feedback rather than improve selection.

CoSPlay addresses this issue by spending additional computation on improving the quality of both sides of the execution matrix. During UT generation, exploration-attack ideas produce more targeted tests for potential failure cases. During self-play, low-value codes are pruned, suspicious low-support UTs are refreshed, buggy codes are refined using high-support non-trivial UTs, and zero-discrimination UTs are replaced. Thus, the extra computation is used to strengthen the verification signal itself, rather than merely increasing the sample count. This explains why the full method outperforms scaled TTS baselines at similar or larger budgets, while its lightweight variants still preserve strong accuracy with substantially lower token usage.

Finally, all competing baselines are evaluated under generous budgets. Methods with built-in stopping criteria, such as SFS (Light et al., 2025) and CodeTree (Li et al., 2025b), are allowed to run until convergence. For methods without automatic stopping, we assign large sampling or search budgets to avoid under-exploration. Therefore, the comparisons in Table 8, Table 6, and Table 7 should be understood as comparisons against strong scaled configurations of existing TTS baselines. Detailed hyperparameter and budget configurations are provided in Appendix D.

G Detailed data of incremental ablation

Table 9 and Figure 11 reports the accuracy and cumulative output tokens at each stage of the incremental ablation, together with the accuracy gain, additional token cost, and gain per 100K additional tokens relative to the preceding stage. Three points are visible in the table but hard to read off the curve. First, marginal efficiency decreases within the self-play stage, from 1.8 to 1.1, 1.0, and 0.3 percentage points per 100K additional tokens along the

tested path. The full pipeline is not monotonic in efficiency: self-consistency and Cluster selection achieve 13.5 and 8.8 percentage points per 100K additional tokens, respectively. Second, Step 3 adds 1.0 percentage point when Step 1 is already enabled (34.2% to 35.2%), and Step 2 adds another 0.6 percentage points on top of Steps 1 and 3 (35.2% to 35.8%). The resulting Steps 1, 2, and 3 configuration gains 1.6 percentage points over Step 1 alone and 2.8 percentage points over the configuration without self-play (33.0%). This path establishes positive conditional contributions but does not isolate a synergistic interaction between Steps 2 and 3. Third, Step 4 has the largest marginal token cost (337,391.8 tokens) for a gain of 0.9 percentage points, whereas Cluster selection has the smallest (5,678.1 tokens) for a gain of 0.5 percentage points. Step 4 should not be interpreted as simply increasing UT accuracy. In the separate ablation in Table 13, retaining all-passing UTs (*w/o resample-UT 100%*) yields higher average UT accuracy than Full (83.9% vs. 78.7%) but lower BoN (34.3% vs. 36.7%). This contrast is consistent with a trade-off between raw UT accuracy and discriminative value.

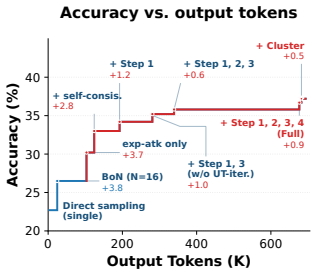


Figure 11: Incremental ablation of CoSPlay components on Qwen2.5-7B-Instruct. **[3×200]**

H Detailed data about generalization of our method

As shown in Table 10, we provide a detailed breakdown of CoSPlay on ten backbones spanning instruction-tuned, RLVR-tuned, and long-CoT models. These evaluations examine whether the proposed Code–UT co-evolution mechanism generalizes across different backbones and training paradigms. Averaged over the ten evaluated backbones, CoSPlay with cluster selection improves BoN from 45.8% to 52.2%, corresponding to an absolute gain of 6.5 percentage points.

The table shows three main trends. First, CoSPlay brings large gains on most mid-sized models. For instance, Seed-Coder-8B-Instruct improves from 32.3% to 42.3%, and DeepSeek-Coder-V2-Lite-Instruct-16B improves from 30.8% to 41.3%. Similar improvements are observed on RLVR-tuned 7B models: Absolute-Zero-7B increases from 24.0% to 30.2%, AceCoder-Rule-7B from 29.0% to 39.5%, and AceCoder-RM-7B from 29.3% to 40.5%. These results indicate that CoSPlay is not limited to one specific backbone type, but can serve as an inference-time complement to both instruction tuning and RLVR-style post-training.

Second, CoSPlay also improves performance on Gemini-2.0-Flash, DeepSeek-V3.2-685B, and long-CoT backbones, although the magnitude varies across models. Gemini-2.0-Flash improves its average BoN from 53.8% to 55.3%, with gains on LiveCodeBench (62.0% to 65.3%), CodeContests (48.7% to 51.3%), and CodeForces (36.7% to 38.7%). DeepSeek-V3.2-685B improves its average BoN from 65.7% to 68.2%, with gains in UT accuracy (74.0% to 79.5%) and code accuracy (56.6% to 63.8%), despite a slight decrease in Verifier Acc. (40.0% to 39.2%). Among long-CoT models, Qwen3-4B improves from 64.5% to 67.0%, CURE-4B from 65.0% to 66.7%, and OCR-Nemotron-1.1-7B from 63.2% to 71.3%.

Third, improvements in average performance do not imply gains on every benchmark. For example, on LiveBench, Gemini-2.0-Flash decreases from a baseline BoN of 68.0% to 66.0% with CoSPlay and cluster selection. Absolute-Zero-7B also decreases from 11.3% to 10.0% on CodeForces, despite its average BoN increasing from 24.0% to 30.2%. These exceptions show that the benefits of Code–UT co-evolution depend on the backbone–benchmark combination. The aggregate gains therefore coexist with localized regressions, motivating further analysis of when self-generated execution feedback supports reliable refinement and selection.

Overall, these detailed results suggest that CoSPlay is most effective when the initial Code-UT signal is useful but not saturated. In such settings, execution feedback can guide code and UTs to improve each other. When the baseline is already near-optimal on an easier benchmark, the extra self-play process may introduce unnecessary perturbations; when the initial model is too weak on a difficult benchmark, the signal may be too noisy to bootstrap reliable co-evolution. Detailed analyses are provided in Section 4

Table 10: Generalization capability of CoSPlay across different **base models** and **RL-tuned models [3×200]**. **UT** and **Code** represent the average accuracy of generated unit tests and code solutions, respectively. Verif.: BoN accuracy when the generated UTs select from a fixed pool of Qwen2.5-7B-Ins. code candidates, isolating the discriminative power of corresponding UTs. **BoN** reports the accuracy of the final selected code. The **baseline** evaluated here represents the model's standard **BoN** performance. **Bold** means the best performance.

Method	LiveBench				LiveCodeBench				CodeContests				CodeForces				Average			
	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN
Base Models																				
Seed-Coder-8B-Instruct																				
Baseline	48.0	26.8	40.7	50.7	32.0	26.5	32.4	38.7	28.7	21.4	16.6	26.7	10.0	13.5	6.7	13.3	29.7	22.1	24.1	32.3
CoSPlay (Ours)	**52.0**	**63.3**	**47.5**	**56.0**	**42.7**	**75.9**	**42.2**	**52.7**	**34.0**	**77.4**	**29.5**	**35.3**	**17.3**	**93.1**	**13.5**	**25.3**	**36.5**	**77.4**	**33.2**	**42.3**
DeepSeek-Coder-V2-Lite-Instruct-16B																				
Baseline	48.7	27.4	30.8	38.7	38.0	39.9	28.4	34.7	**34.0**	51.9	22.8	34.0	12.7	27.3	7.9	16.0	33.3	36.6	22.5	30.8
CoSPlay (Ours)	**53.3**	**68.7**	**42.0**	**51.3**	**43.3**	**86.6**	**43.5**	**48.7**	31.3	**82.8**	**35.8**	**41.3**	16.7	**92.9**	**15.4**	**24.0**	36.2	**82.7**	**34.2**	**41.3**
DeepSeek-V3.2-685B																				
Baseline	**58.7**	**82.1**	68.5	**77.3**	45.3	**85.1**	75.7	**82.7**	**38.0**	76.3	58.8	63.3	**18.0**	52.4	23.2	39.3	**40.0**	74.0	56.6	65.7
CoSPlay (Ours)	57.3	74.0	**76.9**	76.0	**46.7**	83.2	**78.4**	80.0	36.0	**76.9**	**60.2**	**66.7**	16.7	**83.7**	**39.4**	**50.0**	39.2	**79.5**	**63.8**	**68.2**
Gemini-2.0-Flash																				
Baseline	**56.7**	50.3	59.8	**68.0**	42.0	50.2	56.6	62.0	30.7	36.2	44.9	48.7	**14.7**	26.4	33.0	36.7	**36.0**	40.8	48.6	53.8
CoSPlay (Ours)	53.3	**76.2**	**63.2**	66.0	**43.3**	**83.5**	**61.5**	**65.3**	**32.7**	**67.0**	**48.7**	**51.3**	14.0	**69.3**	**35.3**	38.7	35.8	**74.0**	**52.2**	**55.3**
RL-tuned Models																				
Absolute-Zero-7B																				
Baseline	44.0	19.3	11.6	36.7	32.7	31.0	11.6	24.7	28.0	28.5	8.9	23.3	**11.3**	15.7	4.8	**11.3**	29.0	23.6	9.2	24.0
CoSPlay (Ours)	**52.0**	**80.4**	**33.0**	**47.3**	**36.0**	**84.9**	**28.1**	**36.7**	29.3	**88.9**	**21.7**	**26.7**	9.3	**84.9**	**6.8**	10.0	**31.7**	**84.8**	**22.4**	**30.2**
AceCoder-Rule-7B																				
Baseline	44.0	13.9	40.7	46.0	29.3	15.9	34.0	36.0	27.3	5.3	23.3	25.3	10.0	3.0	6.9	8.7	27.7	9.5	26.2	29.0
CoSPlay (Ours)	**52.7**	**64.6**	**45.2**	**54.7**	**44.7**	**69.7**	**40.0**	**51.3**	**34.0**	**72.7**	**25.5**	33.3	**17.3**	**87.4**	**9.6**	18.7	**37.2**	**73.6**	**30.1**	39.5
AceCoder-RM-7B																				
Baseline	43.3	15.1	40.9	48.7	29.3	17.2	34.0	38.0	27.3	8.5	22.4	25.3	8.0	7.1	5.8	5.3	27.0	12.0	25.8	29.3
CoSPlay (Ours)	**56.7**	**61.3**	**47.0**	56.0	**44.0**	**70.7**	**39.8**	**50.0**	**32.6**	**73.5**	**25.8**	**34.7**	**17.3**	**85.2**	**10.4**	**21.3**	**37.7**	**72.7**	**30.8**	**40.5**
Long-Cot Models																				
Qwen3-4B																				
Baseline	54.7	63.5	74.8	79.3	42.7	67.6	76.7	76.7	**36.7**	66.5	55.5	**61.3**	16.7	62.1	31.1	40.7	37.7	64.9	59.5	64.5
CoSPlay (Ours)	**56.7**	**79.3**	**77.0**	**82.7**	**46.0**	**84.8**	**78.8**	**81.3**	**36.7**	**74.8**	**57.2**	**61.3**	**17.3**	**68.9**	**35.0**	**42.7**	**39.2**	**77.0**	**62.0**	**67.0**
CURE-4B																				
Baseline	**58.0**	86.2	75.2	**81.3**	44.7	88.6	74.7	80.7	**36.0**	78.3	55.7	61.3	16.0	70.6	32.5	36.7	**38.7**	80.9	59.5	65.0
CoSPlay (Ours)	55.3	**87.2**	**78.0**	**81.3**	**46.0**	**93.8**	**79.1**	**81.3**	**36.0**	**87.1**	**57.5**	**63.3**	**16.7**	**82.3**	**35.5**	**40.7**	38.5	**87.6**	**62.5**	**66.7**
OCR-Nemotron-1.1-7B																				
Baseline	55.3	27.8	63.1	71.3	43.3	29.8	64.2	68.7	35.3	32.9	54.3	**66.0**	13.3	23.6	26.8	46.7	36.8	28.6	52.1	63.2
CoSPlay (Ours)	58.0	**68.0**	**77.4**	**82.7**	46.0	**73.9**	**77.2**	**80.0**	**36.0**	**80.5**	**61.2**	63.3	**18.0**	**77.6**	**45.0**	**59.3**	39.5	**75.0**	**65.2**	**71.3**

I CoSPlay under Partially Available GT UTs

Table 11 evaluates CoSPlay when each problem comes with a small set of trustworthy but insufficient GT UTs. Following the main protocol, we sample 50 problems from each of the four datasets with seeds 41, 42, and 43, forming three independent 200-problem benchmarks, and report the averaged BoN on Qwen2.5-7B-Instruct. For each problem, we retain the original $N_t = 16$ self-generated UTs and add four GPT-generated UTs whose input-output pairs have been verified as ground truth, resulting in 20 UTs in total. The four GT UTs are treated as trusted and non-replaceable throughout self-play and are therefore never refreshed or replaced. In Step 3, if at least one GT UT is not passed by all current code candidates, we prioritize the highest-support non-trivial GT UT for code fixing; otherwise, we follow the standard non-trivial-best UT selection rule over the self-generated UTs. In Step 2, GT UTs are excluded from the candidate set used to identify the lowest-support non-trivial UT, ensuring that trusted GT UTs are never treated as potentially spurious Code–UT couplings.

Table 11: **CoSPlay with partially available GT UTs on Qwen2.5-7B-Instruct [3 × 200].** For each problem, four verified GT UTs are added to the original 16 self-generated UTs, resulting in 20 UTs in total. The GT UTs form a trusted and non-replaceable subset, while the self-generated UTs supplement the remaining coverage. *Oracle* denotes pass@16, i.e., the oracle ceiling of selecting from the candidate code pool.

Setting	LiveBench	LiveCodeBench	CodeContests	CodeForces	Average
Baseline	38.0	32.7	25.3	10.0	26.5
w/ 4× GT	**60.0**	**50.0**	**32.7**	**15.3**	**39.5**
Oracle	64.0	50.0	42.0	15.3	42.8
CoSPlay	51.3	41.3	38.0	18.0	37.2
w/ 4× GT	**57.3**	**53.3**	**43.3**	**18.7**	**43.2**
Oracle	64.7	55.3	44.7	18.7	45.8

Table 12: **Verifier Acc. (%) on the full datasets [Full].** Candidate pools contain 16 codes per problem generated by Qwen2.5-7B-Instruct. Baseline uses 16 generated UTs to select the code with the highest test pass rate. CoSPlay uses its final-round UTs with clustering-based selection. Reward models select the code with the highest predicted score. **#Data** denotes the number of examples used for additional training. Average is weighted by dataset size.

Verifier	#Data	LiveBench	LiveCodeBench	CodeContests	CodeForces	Average
Qwen2.5-7B-Ins.	–	36.2	31.2	24.1	7.1	22.1
CoSPlay-7B	0	**50.3**	**43.7**	32.6	**13.6**	**31.9**
AceCodeRM-7B	300k	39.1	36.4	30.1	7.7	25.6
AceCodeRM-32B	300k	43.8	38.7	**33.5**	8.8	27.9

J Effectiveness of CoSPlay for Code Verification versus Reward Model

Table 12 evaluates how well different verifiers identify correct code from the same candidate pool. For each problem in the full datasets, Qwen2.5-7B-Instruct generates 16 candidate codes, and each verifier selects one of them. The baseline and CoSPlay select candidates by the pass rates on their UTs, while the reward models select by predicted scores. Notably, AceCodeRM-7B and AceCodeRM-32B are trained on about 300k synthetic preference pairs. Without any training or GT supervision, CoSPlay improves average Verifier Acc. from 22.1% to 31.9%, outperforming AceCodeRM-7B and AceCodeRM-32B by 6.3 and 4.0 percentage points, respectively. Together with the UT accuracy gains in Table 1, these results show that CoSPlay's generated UTs are not only more accurate but also more discriminative, making them effective for code verification.

K Dataset Overview

LiveBench. LiveBench is a dynamic and contamination-limited benchmark designed for broad LLM evaluation, with tasks spanning multiple domains such as math, coding, reasoning, language, instruction following, and data analysis (White et al., 2025). A key feature of LiveBench is that it is updated frequently and emphasizes objective, automatically checkable evaluation, making it well suited for reducing benchmark contamination and maintaining difficulty over time. In our work, we use its coding subset (128 problems) as a challenging evaluation setting for measuring code reasoning and functional correctness under a more up-to-date benchmark distribution.

LiveCodeBench. We use LiveCodeBench v2 (Jain et al., 2025b), a contamination-controlled benchmark for evaluating code generation and related code abilities of LLMs. It contains 511 problems released between May 2023 and May 2024, collected from recent programming contests on LeetCode, AtCoder, and CodeForces. Since each problem is tagged with its original release time, the benchmark supports temporally controlled evaluation and helps reduce train-test leakage. The problems are

Table 13: Ablation study of CoSPlay-7B **[3×200]**.

Model Variant	LiveBench				LiveCodeBench				CodeContests				CodeForces				Average			
	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN
Full	54.7	71.9	41.3	51.3	40.7	72.7	31.6	41.3	34.0	77.4	23.7	36.0	16.7	92.8	9.4	18.0	36.5	78.7	26.5	36.7
w/o exp-atk	42.0	29.4	41.7	48.0	26.7	32.5	30.4	32.7	27.3	42.1	25.7	30.7	7.3	26.2	7.6	10.0	25.8	32.5	26.3	30.3
w/o self-play	49.3	53.0	30.9	50.0	36.7	60.4	24.3	40.7	32.7	59.4	18.8	29.3	17.3	50.0	5.6	12.0	34.0	55.7	19.9	33.0
w/o random UT	51.3	65.3	41.8	52.7	40.7	74.3	31.9	40.0	34.7	76.4	24.5	31.3	17.3	88.0	8.2	18.0	36.0	76.0	26.6	35.5
w/o non-trivial best	53.3	71.7	42.1	50.7	42.0	72.7	32.3	37.3	32.7	77.7	25.4	32.7	16.7	91.6	10.4	19.3	36.2	78.4	27.6	35.0
w/o self-consis.	51.3	55.4	41.9	48.0	40.0	64.4	30.9	38.0	35.3	62.6	23.0	30.7	16.0	72.2	8.2	16.7	35.7	63.7	26.0	33.3
w/o self-play step2	50.7	63.1	42.1	49.3	38.0	72.4	31.7	37.3	34.7	74.7	24.9	37.3	16.7	86.8	8.0	16.0	35.0	74.2	26.7	35.0
w/o self-play step3	52.0	63.0	39.7	52.7	41.3	76.0	29.8	40.0	32.7	70.9	22.1	32.0	17.3	85.8	8.6	18.7	35.8	73.9	25.1	35.8
w/o resample-UT 0%	48.7	33.6	42.2	46.7	38.0	48.1	31.9	38.0	32.7	43.5	24.4	28.0	14.7	30.7	9.0	16.7	33.5	38.9	26.9	32.3
w/o resample-UT 100%	52.7	77.6	41.3	47.3	42.0	88.0	32.0	40.0	33.3	83.8	24.3	33.3	16.7	86.4	8.7	16.7	36.2	83.9	26.6	34.3
w/o resample-Code 0%	51.3	77.2	33.8	48.0	42.7	80.7	25.5	38.0	31.3	73.4	22.4	32.7	16.7	91.0	7.0	14.7	35.5	80.6	22.2	33.3
w/o UT-iteration	49.3	50.3	40.4	51.3	35.3	61.5	29.2	38.0	32.0	53.6	23.1	32.7	16.7	47.7	9.0	18.7	33.3	53.3	25.4	35.2
w/o self-play & self-consis.	49.3	38.3	33.3	46.7	38.7	49.6	21.3	36.0	32.0	30.5	18.7	24.7	14.7	30.3	5.3	13.3	33.7	37.2	19.7	30.2
baseline(Direct resample)	38.0	8.2	36.0	38.0	32.7	13.9	27.1	32.7	25.3	15.1	21.3	25.3	10.0	12.6	6.5	10.0	26.5	12.5	22.7	26.5

drawn from real contests and have been validated by large numbers of participants, which helps ensure high data quality.

CodeContests. CodeContests is a competitive-programming dataset introduced with AlphaCode (Li et al., 2022). It is constructed from Codeforces, Description2Code, and APPS. In our experiments, we use a subset of 239 problems sampled from the original dataset provided by CURE (Wang et al., 2025b).

CodeForces. We use the `open-r1/codeforces` dataset (Penedo et al., 2025), which is built from CodeForces, one of the most popular competitive-programming platforms. The full dataset contains more than 10k unique problems collected from contests spanning the earliest rounds to 2025. In our experiments, we use a randomly sampled subset of 467 problems with difficulty levels below 2 provided by CURE (Wang et al., 2025b). A key feature of this dataset is its emphasis on verifiability: for problems with multiple valid outputs, it provides custom checker programs, and it also includes additional challenging test cases beyond the truncated examples typically visible on the contest platform. These properties make it well suited for evaluating code reasoning and functional correctness on realistic competitive-programming tasks.

Sampled evaluation protocol. Running every experiment on all 1,345 problems is computationally infeasible, so only Table 1, Figure 1, and Table 14 use the full suite. Every other result uses a 3*200-problem sample. For each of the seeds 41, 42, and 43 we draw 50 problems from each of the four datasets, which gives three independent 200-problem benchmarks. We report the mean over the three seeds. Because each dataset contributes the same number of problems, the reported average weights the four datasets equally, whereas the average on the full suite is weighted by dataset size. Numbers obtained under the two protocols are therefore not directly comparable in absolute terms.

L Detailed Ablation Results

L.1 Ablation Variant Descriptions

We evaluate the contribution of each component through the following ablation variants. **w/o exp-atk** removes Stage 1 entirely, generating codes and UTs directly without solution-aware idea exploration. **w/o self-play** removes Stage 2 entirely, skipping all iterative refinement and selecting directly from the initial pool. **w/o random UT** initializes the UT pool using only failure-oriented inputs, removing the random-valid-input half of the hybrid initialization strategy. **w/o non-trivial best** replaces the targeted non-trivial Best UT selection in Step3 with a randomly chosen UT for code fixing. **w/o self-consis.** disables self-consistency during UT output validation, using a single-sample output as the expected value instead of majority voting. **w/o self-play step2** removes Step 2 (breaking spurious Code-UT coupling) while retaining all other self-play steps. **w/o self-play step3** removes Step 3 (code fixing) while retaining all other self-play steps. **w/o resample-UT 0%** disables replacement of zero-pass-rate UTs in Step 4, retaining UTs that all candidates fail. **w/o resample-UT 100%** disables replacement of full-pass-rate UTs in Step 4, retaining UTs that all candidates pass. **w/o**

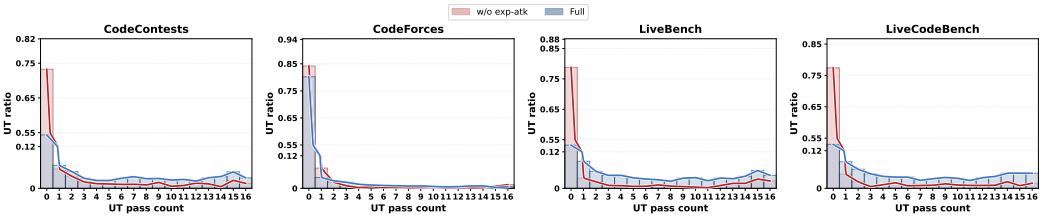


Figure 12: UT pass count (number of code candidates passing each UT) distributions at the UT initialization stage, comparing the full method (blue) against the variant w/o exp-atk (red).

resample-Code 0% disables code resampling in Step 1, retaining all-failing code candidates instead of replacing them. **w/o UT-iteration** removes all UT-side update steps within self-play (Steps 2 and 4), keeping the UT pool fixed throughout all rounds while still performing code cleaning and fixing. **w/o self-play & self-consis.** removes both Stage 2 and self-consistency, serving as a lower-bound baseline. **Baseline (Direct resample)** applies no CoSPlay mechanisms, directly sampling a large pool of codes and selecting via BoN with self-generated UTs.

L.2 Analysis

As shown in Table 13, removing the first stage leads to lower BoN performance even though raw code accuracy may increase, indicating that idea exploration mainly improves the discriminative quality of generated UTs rather than naive code quality. Ablations on the self-play stage show a consistent pattern: CoSPlay works because it jointly improves *UT quality* and *code quality*. On the UT side, removing random UT initialization, self-consistency, or UT regeneration all hurts final BoN. In particular, keeping 0%-pass UTs causes severe degradation, while keeping 100%-pass UTs also reduces BoN as such tests are trivial and non-discriminative. On the code side, removing the refinement step or not resampling 0%-pass codes degrades both code accuracy and BoN. Finally, replacing the non-trivial best UT with a random UT weakens refinement, confirming that targeted UT selection provides a stronger refinement signal. Overall, these results support our design that effective self-play requires both progressively refined UTs and targeted code refinement.

M Detailed data of UT pass count distribution about Idea-Level Exploration vs. direct sample

Figure 12 presents a detailed comparison of UT pass count distributions between the full method and the variant without Exploration-Attack stage (w/o exp-atk), evaluated at the UT initialization stage before self-play stage. In all of the UT pass count plots, the w/o exp-atk variant shows a pronounced concentration of zero-pass UTs across all four datasets, indicating that a large proportion of directly sampled UTs are not passed by any code candidate–these are highly likely to be incorrect or noisy. In contrast, the full method substantially reduces this spike and shifts the distribution toward moderate pass counts (roughly 2-10). This more uniform spread suggests that UTs generated via our proposed Exploration-Attack stage are not only more accurate (fewer zero-pass outliers) but also more discriminative: instead of trivially failing or trivially passing all candidates, they meaningfully differentiate among code solutions. These distributions confirm that this exploration improves the quality of initial UT pool in a complementary manner: it produces UTs that are more accurate and more evenly discriminative, which in turn provides a stronger foundation for the subsequent self-play stage.

N Scalability Analysis of Random Input Size for Clustering

To further evaluate the robustness of our execution-consensus-based cluster selection strategy, we analyze how the number of random valid inputs used for clustering affects the final performance. Specifically, we use Qwen2.5-7B-Instruct with BoN ($N = 16$) and vary the number of clustering inputs k from 0 to 64.

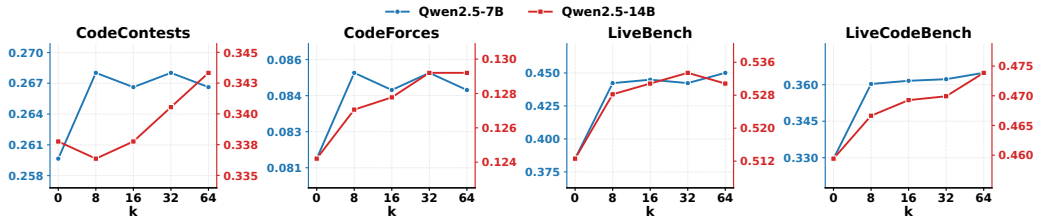


Figure 13: Effect of the number of random valid inputs used for execution-consensus clustering. We vary the number of clustering inputs from 0 to 64 and report the results on CodeContests, CodeForces, LiveBench, and LiveCodeBench. Increasing the number of inputs provides a more reliable execution signature for grouping functionally equivalent candidates.

As shown in Figure 13, increasing k generally improves or stabilizes performance across the four datasets. This is because more random inputs provide a more reliable execution signature, making it easier to group functionally equivalent candidates and separate spurious agreements. The gains are especially visible on LiveBench and LiveCodeBench, suggesting that execution-consensus selection benefits more when candidate behaviors are diverse and difficult to distinguish with only a few inputs.

The improvement is not strictly monotonic, since the randomly generated inputs may vary in discriminative strength. Nevertheless, the overall trend shows that using more clustering inputs leads to more stable selection, while performance does not collapse under larger k. This confirms that the cluster-selection stage scales favorably with the number of random inputs and can effectively exploit additional test-time execution budget.

O VARIANCE ANALYSIS

Table 14 reports the mean performance and population standard deviation over three independent runs. Each cell reports the mean, with the corresponding standard deviation in parentheses. The generally small standard deviations across most datasets and metrics suggest that the observed improvements are stable rather than driven by random sampling variation.

Table 14: Mean accuracy and population standard deviation over three runs **[Full]**. Each cell reports the mean accuracy and the corresponding standard deviation, both expressed as percentages. The standard deviation is computed with normalization by the number of runs.

Model	LiveBench				LiveCodeBench				CodeContests				CodeForces				Average			
	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN	Verif.	UT	Code	BoN
7B Models																				
Qwen2.5-7B-Instruct	36.2 (2.9)	9.0 (0.6)	33.1 (0.8)	36.2 (2.9)	31.2 (1.0)	14.4 (0.2)	27.0 (0.1)	31.2 (1.0)	24.1 (1.1)	21.3 (0.9)	21.5 (0.6)	24.1 (1.1)	7.1 (0.2)	12.9 (0.7)	5.0 (0.0)	7.1 (0.2)	22.1 (0.6)	14.6 (0.3)	19.0 (0.1)	22.1 (0.6)
Qwen2.5-7B-Coder	40.6 (1.3)	12.1 (1.9)	3.8 (0.1)	19.0 (2.7)	33.4 (1.6)	16.0 (0.6)	4.1 (0.3)	17.7 (1.0)	25.2 (0.9)	21.3 (1.8)	2.6 (0.2)	12.8 (1.4)	6.8 (0.4)	6.7 (1.8)	1.8 (0.8)	3.5 (1.2)	23.4 (0.5)	13.3 (0.5)	3.0 (0.1)	12.0 (0.7)
Qwen2.5-7B-Coder-Ins.	42.7 (1.6)	12.5 (0.3)	36.3 (0.3)	47.7 (1.1)	33.7 (0.4)	13.4 (1.3)	28.8 (0.7)	34.4 (2.9)	24.0 (0.7)	12.7 (0.6)	20.9 (0.2)	23.2 (0.4)	6.9 (0.4)	5.3 (0.8)	5.4 (0.1)	7.9 (0.6)	23.5 (0.5)	10.4 (0.3)	20.0 (0.2)	24.5 (1.0)
Seed-Coder-8B-Instruct	43.5 (1.3)	26.9 (1.3)	37.6 (0.7)	45.1 (1.9)	36.1 (0.2)	30.0 (0.7)	32.5 (0.1)	40.8 (0.5)	25.0 (0.9)	31.8 (0.0)	18.3 (0.2)	26.9 (1.0)	8.4 (1.0)	17.0 (1.1)	6.4 (0.2)	10.5 (0.5)	25.2 (0.6)	25.5 (0.4)	21.4 (0.0)	28.2 (0.3)
AceCoder-7B-Rule	43.5 (0.4)	17.2 (0.8)	36.1 (0.2)	45.8 (1.0)	34.1 (0.2)	17.5 (0.7)	30.5 (0.6)	35.9 (0.6)	24.4 (2.6)	18.8 (1.5)	20.1 (0.6)	23.2 (0.9)	6.4 (0.6)	5.9 (0.3)	5.4 (0.1)	7.7 (0.6)	23.6 (0.3)	13.7 (0.2)	20.5 (0.3)	24.8 (0.4)
AceCoder-7B-RM	43.0 (1.7)	17.7 (0.6)	37.5 (0.5)	48.4 (1.7)	33.7 (1.1)	19.9 (0.7)	30.3 (0.4)	37.1 (0.2)	25.0 (0.5)	20.0 (0.9)	20.6 (0.4)	25.0 (0.5)	6.6 (0.3)	6.6 (1.0)	5.2 (0.1)	7.8 (0.4)	23.6 (0.5)	15.1 (0.7)	20.6 (0.1)	25.8 (0.4)
AZR-7B-Coder	44.5 (1.1)	23.1 (0.7)	12.5 (0.4)	34.9 (2.2)	37.1 (0.6)	31.7 (1.1)	12.4 (0.3)	29.8 (1.0)	29.1 (0.7)	39.0 (1.2)	13.7 (6.3)	24.4 (2.9)	7.6 (0.7)	16.9 (0.5)	5.3 (0.7)	8.4 (0.3)	26.1 (0.4)	27.0 (0.5)	10.2 (1.3)	21.9 (0.7)
CURE-7B	52.3 (1.7)	54.6 (1.3)	37.4 (0.1)	51.6 (2.3)	42.1 (0.3)	58.4 (0.9)	31.2 (0.3)	42.7 (0.7)	33.9 (0.7)	63.7 (0.9)	25.5 (0.3)	34.5 (1.1)	11.5 (0.4)	46.6 (1.9)	7.5 (0.2)	16.1 (1.0)	31.0 (0.4)	54.9 (0.5)	22.5 (0.1)	32.9 (0.3)
Qwen2.5-7B-Ins. + CoSPlay	50.3 (1.0)	66.7 (1.3)	38.8 (0.6)	49.7 (1.6)	43.7 (0.8)	77.0 (1.3)	33.2 (0.1)	43.4 (1.4)	32.6 (0.7)	74.4 (2.4)	26.0 (0.4)	34.6 (1.8)	13.6 (0.3)	85.0 (0.6)	6.8 (0.5)	16.8 (1.3)	31.9 (0.3)	78.3 (0.9)	23.3 (0.2)	33.2 (0.2)
CURE-7B + CoSPlay	53.6 (1.9)	66.2 (2.6)	45.4 (0.9)	52.3 (0.6)	44.7 (0.2)	75.6 (0.9)	37.4 (0.3)	48.6 (1.1)	35.6 (0.7)	76.6 (1.5)	29.8 (0.1)	40.0 (0.9)	14.4 (0.2)	85.6 (0.2)	11.1 (0.3)	23.1 (0.5)	33.4 (0.2)	78.3 (0.4)	27.7 (0.3)	38.6 (0.5)
14B Models																				
Qwen2.5-14B-Instruct	47.7 (2.3)	26.4 (1.1)	39.2 (1.1)	51.0 (0.7)	41.4 (0.8)	38.0 (1.1)	34.5 (0.3)	46.2 (0.6)	30.4 (0.2)	43.2 (0.6)	25.1 (0.3)	34.6 (1.3)	9.9 (1.0)	22.6 (1.3)	7.3 (0.2)	13.3 (1.5)	29.1 (0.3)	32.4 (0.9)	23.9 (0.1)	33.2 (0.5)
Qwen2.5-14B-Coder	42.2 (1.3)	13.9 (0.9)	10.8 (0.4)	30.2 (3.0)	36.1 (0.2)	20.7 (0.5)	9.7 (0.2)	26.7 (0.2)	28.3 (0.5)	25.6 (2.7)	6.8 (0.3)	21.3 (1.2)	10.2 (0.3)	16.7 (0.8)	2.1 (0.2)	9.3 (0.5)	26.3 (0.1)	19.5 (0.3)	6.7 (0.2)	20.1 (0.6)
Qwen2.5-14B-Coder-Ins.	48.2 (0.4)	37.8 (0.3)	47.2 (0.4)	53.9 (0.6)	39.5 (0.6)	40.6 (1.0)	41.0 (0.2)	49.4 (0.6)	27.5 (1.9)	34.8 (1.1)	25.1 (0.2)	31.4 (0.7)	7.9 (0.2)	18.1 (0.7)	5.6 (0.0)	9.6 (0.6)	27.2 (0.1)	31.5 (0.6)	26.5 (0.1)	32.8 (0.4)
DeepSeek-Coder-V2-16B	45.8 (1.3)	31.0 (0.8)	37.4 (0.3)	38.3 (1.1)	39.3 (0.7)	42.1 (3.6)	25.7 (6.9)	36.7 (2.6)	33.6 (1.4)	51.6 (6.7)	22.1 (6.8)	35.1 (2.6)	13.8 (1.4)	37.2 (6.1)	6.5 (0.8)	16.8 (1.1)	30.1 (1.1)	41.0 (4.1)	19.5 (3.7)	29.6 (0.9)
AZR-14B-Coder	39.8 (2.3)	22.2 (0.4)	38.9 (0.5)	43.5 (0.4)	34.1 (3.2)	29.7 (0.6)	35.3 (0.2)	39.8 (0.3)	25.8 (1.7)	35.3 (1.5)	26.4 (0.5)	30.4 (1.5)	8.5 (1.0)	18.9 (0.5)	9.5 (0.1)	16.2 (1.0)	24.3 (2.0)	26.2 (0.1)	25.1 (0.1)	30.3 (0.3)
CURE-14B	53.6 (0.7)	77.3 (0.5)	47.7 (0.2)	59.9 (1.8)	45.1 (0.2)	86.1 (0.8)	41.2 (0.2)	50.5 (0.6)	34.9 (0.7)	87.0 (1.1)	32.5 (0.3)	44.1 (0.5)	14.9 (0.1)	77.5 (1.1)	12.0 (0.2)	26.1 (0.5)	33.6 (0.2)	82.4 (0.4)	30.1 (0.1)	41.8 (0.6)
Qwen2.5-14B-Ins. + CoSPlay	56.5 (1.3)	70.1 (2.5)	47.9 (0.2)	57.6 (1.0)	46.2 (1.3)	74.4 (1.0)	43.5 (0.3)	54.8 (1.2)	34.0 (0.9)	71.7 (0.7)	31.5 (0.6)	40.0 (1.3)	14.3 (0.2)	86.1 (0.7)	11.8 (0.2)	24.6 (1.2)	34.0 (0.4)	77.6 (0.5)	30.8 (0.2)	42.0 (0.9)
CURE-14B + CoSPlay	54.9 (1.3)	71.5 (2.6)	52.4 (0.6)	60.7 (0.7)	45.5 (0.9)	76.2 (2.1)	46.9 (0.2)	55.3 (0.7)	34.9 (0.8)	75.1 (2.9)	36.8 (0.3)	45.3 (2.3)	15.3 (0.1)	89.4 (1.9)	19.1 (0.3)	32.6 (1.5)	34.1 (0.2)	80.1 (2.0)	36.0 (0.2)	46.2 (0.7)

P Detailed Evolution of Execution Statistics

P.1 Detailed Evolution of Cluster-Size Statistics for Correct and Incorrect Codes

Figures 14–21 show the evolution of cluster-size distributions for correct and wrong code candidates during self-play. Overall, correct codes tend to appear in larger execution-signature clusters, while wrong codes are more frequently distributed over smaller clusters. This supports the use of cluster size as a GT-free proxy for functional correctness.

The separation is most evident on medium-difficulty datasets such as CodeContests and LiveCodeBench. On these benchmarks, correct solutions often concentrate in large clusters, while wrong solutions remain relatively fragmented. For harder datasets such as CodeForces, the distributions overlap more, indicating that candidate behaviors are harder to distinguish by cluster size alone. These results suggest that execution-consensus clustering is especifically effective when the model can generate a non-trivial number of correct or near-correct candidates, allowing correct behaviors to form stable clusters.

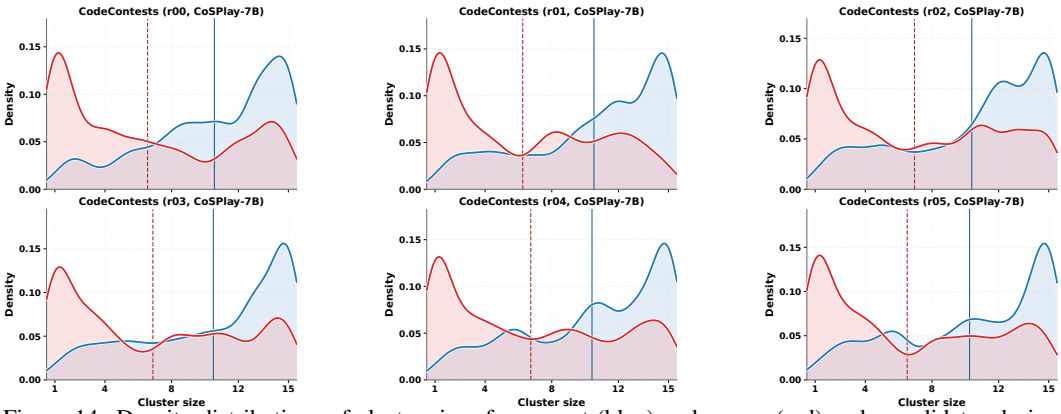


Figure 14: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **CodeContests** with the **7B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

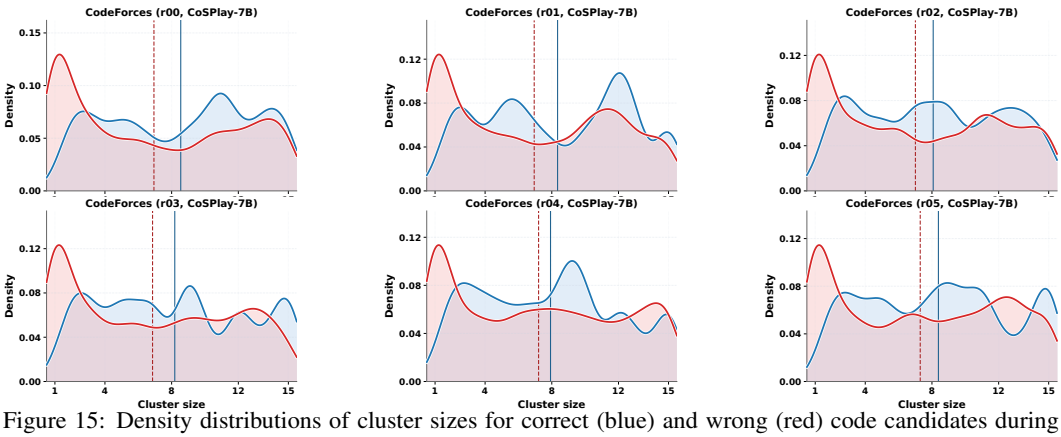


Figure 15: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **CodeForces** with the **7B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

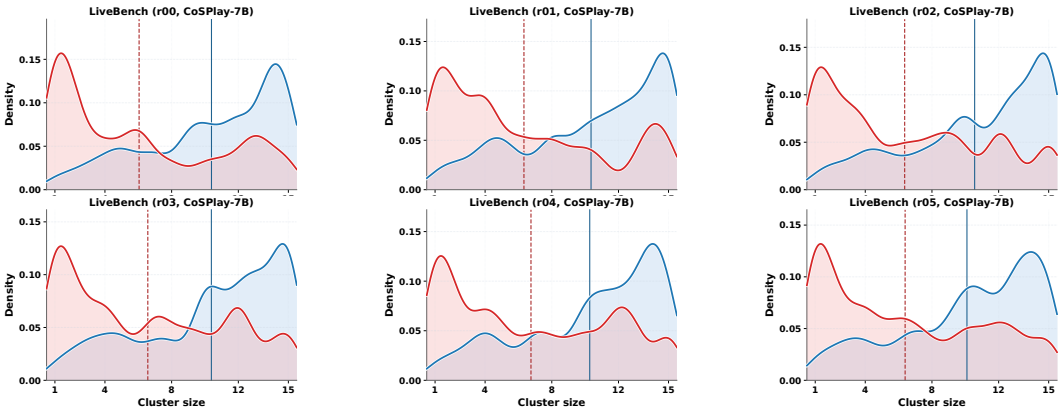


Figure 16: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **LiveBench** with the **7B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

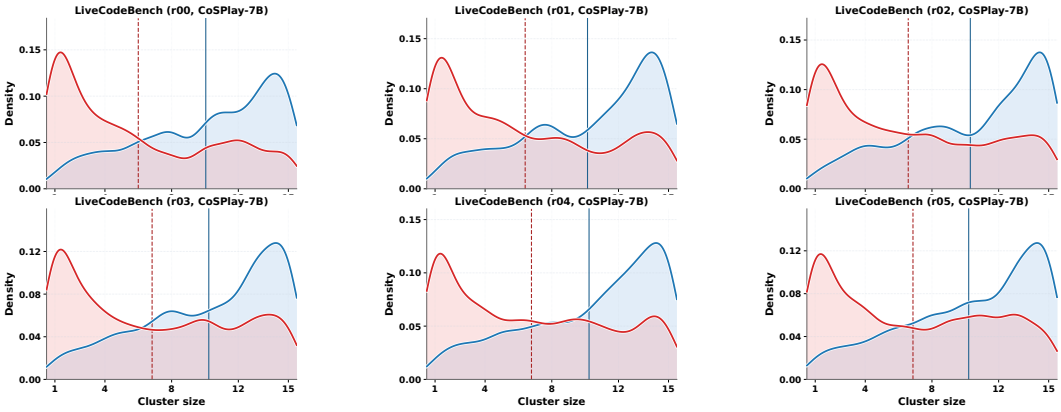


Figure 17: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **LiveCodeBench** with the **7B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

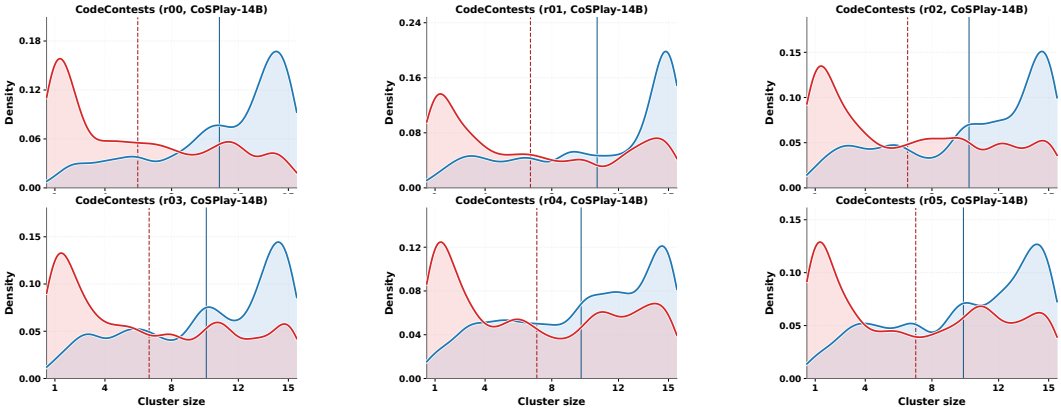


Figure 18: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **CodeContests** with the **14B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

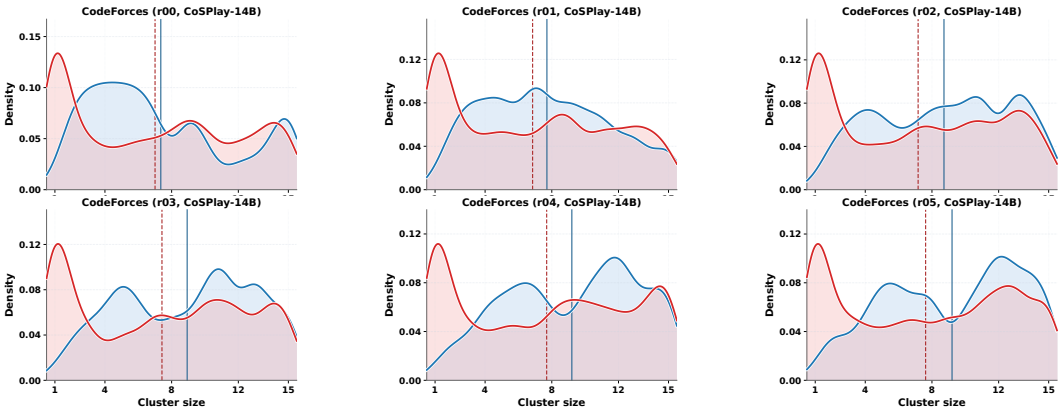


Figure 19: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **CodeForces** with the **14B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

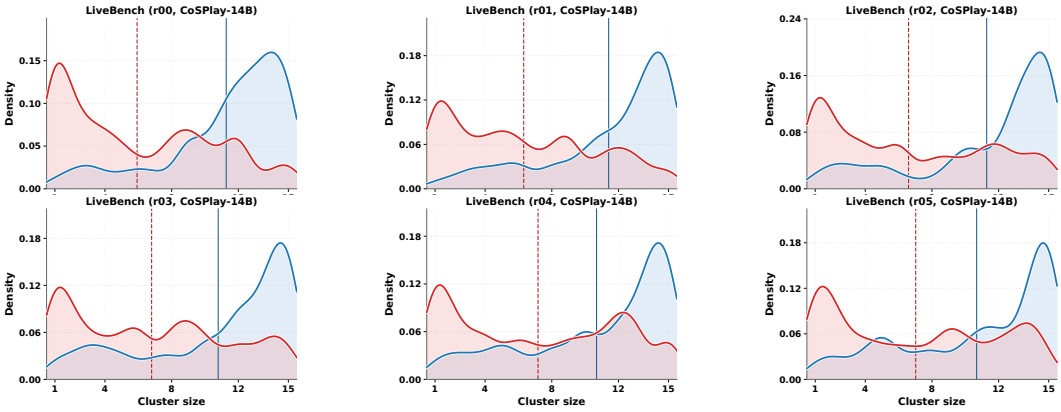


Figure 20: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **LiveBench** with the **14B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

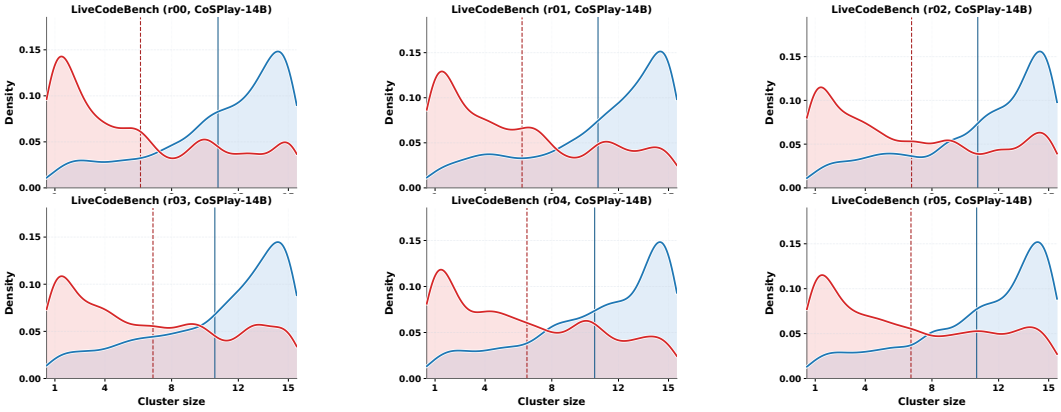


Figure 21: Density distributions of cluster sizes for correct (blue) and wrong (red) code candidates during self-play on **LiveCodeBench** with the **14B model**. The top row shows Round 0–2, and the bottom row shows Round 3–5. Vertical lines denote average values.

P.2 Detailed Evolution of UT Pass-Count Density

Figures 22-29 provide per-dataset and per-round visualizations of the UT pass-count density for both 7B and 14B models. The UT pass count measures how many code candidates pass a generated UT. Across datasets, the density mass gradually shifts from extremely low pass-count regions toward higher pass-count regions as self-play proceeds. This indicates that CoSPlay progressively reduces overly hard, noisy, or weakly validated UTs that are passed by few candidates, while increasing the proportion of UTs that are semantically more consistent with the code pool.

Since UT pass count is positively correlated with true UT correctness, the rightward shift suggests that the UT pool becomes more reliable over self-play rounds. Meanwhile, the continued replacement of zero-discrimination UTs prevents the pool from collapsing into trivial tests, allowing the evolved UTs to remain useful for both refinement and selection.

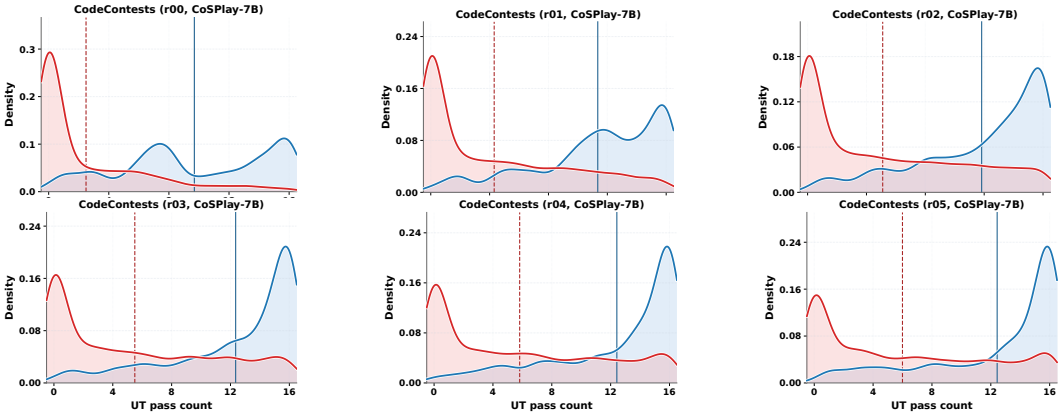


Figure 22: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **CodeContests** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

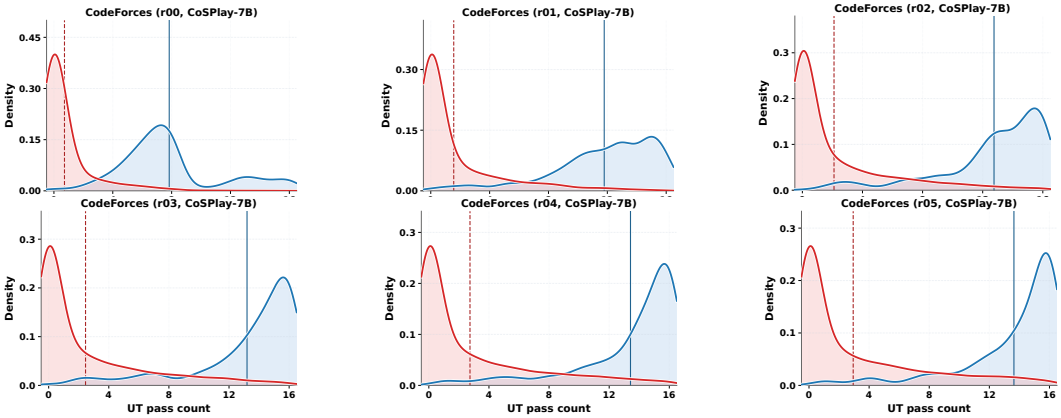


Figure 23: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **CodeForces** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

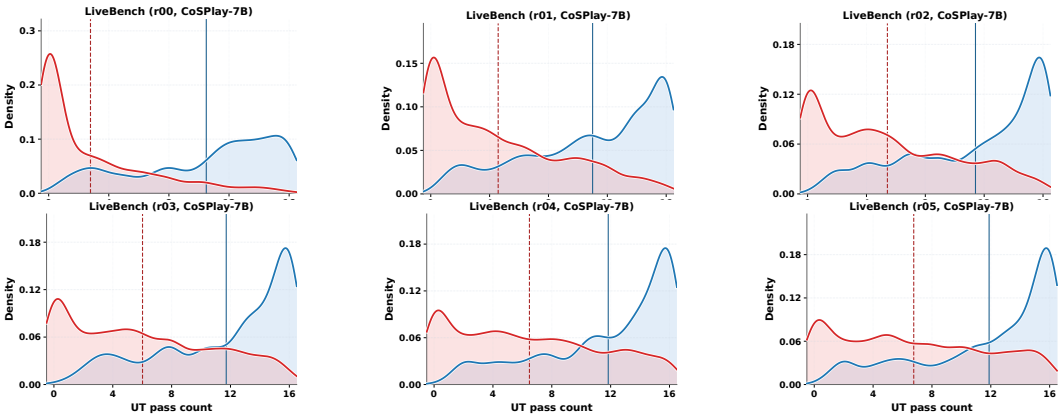


Figure 24: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **LiveBench** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

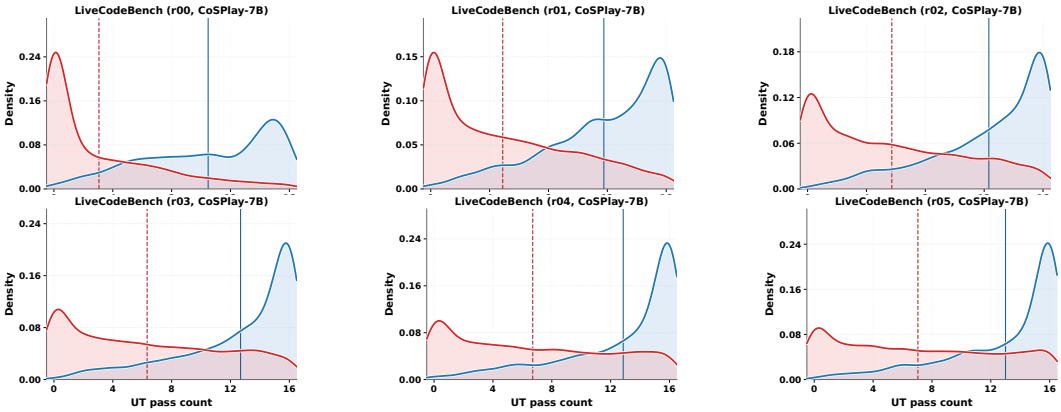


Figure 25: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **LiveCodeBench** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

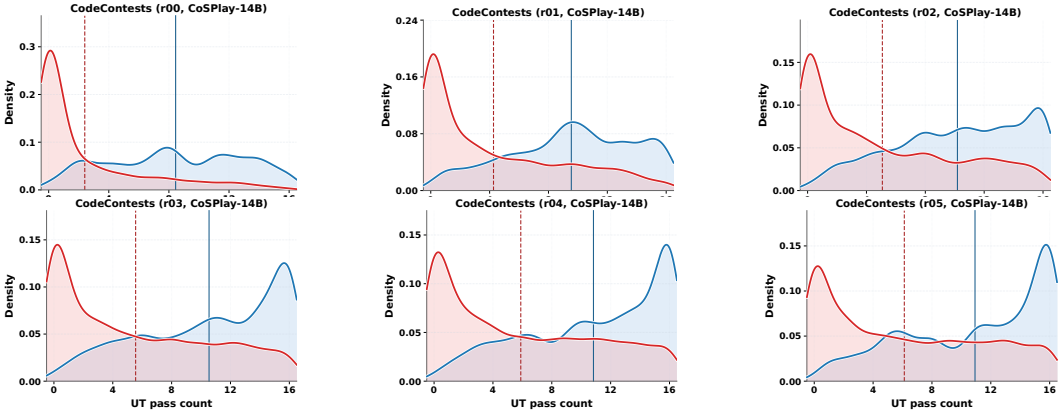


Figure 26: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **CodeContests** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

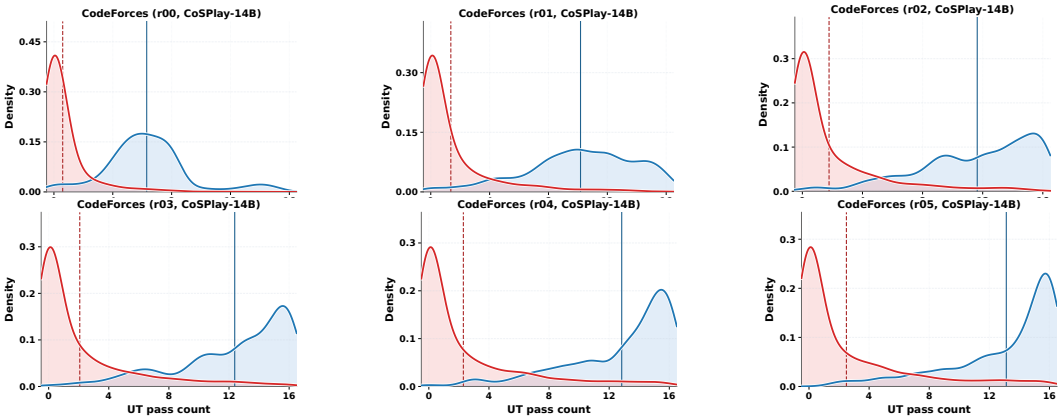


Figure 27: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **CodeForces** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

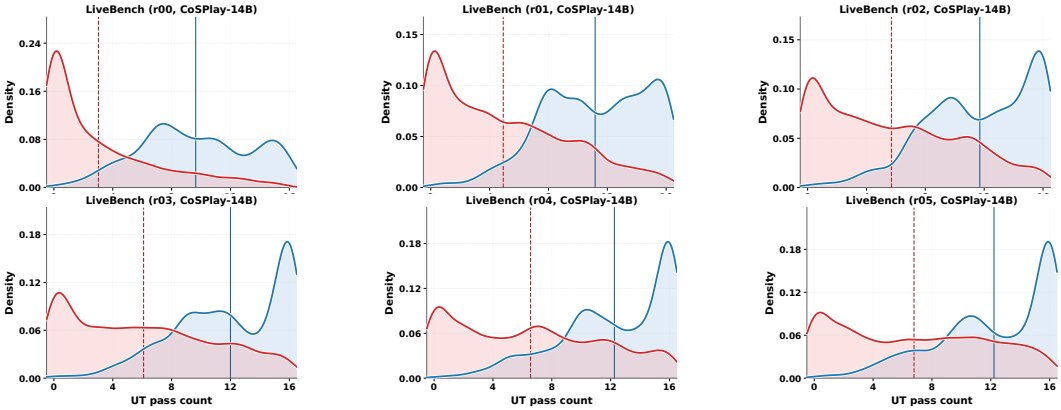


Figure 28: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **LiveBench** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

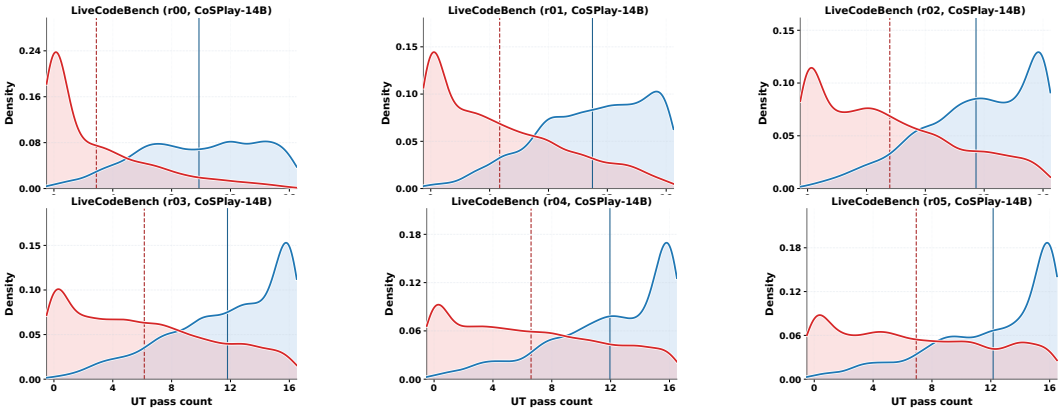


Figure 29: Density distributions of UT pass counts for correct (blue) and wrong (red) UT candidates during self-play on **LiveCodeBench** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

P.3 Detailed Evolution of Code Pass-Count Density

Figures 30-37 show the per-dataset and per-round plots of code pass-count density. The code pass count measures how many generated UTs a code candidate passes. Across datasets and model scales, the density progressively moves away from low pass-count regions and shifts toward higher pass-count regions, indicating that more code candidates satisfy a larger fraction of the evolved UT pool after self-play.

This pattern provides fine-grained evidence for the effectiveness of the iterative code cleaning and code fixing steps. All-failing or weak candidates are removed or refined, while the improved UT pool provides increasingly useful supervision for subsequent refinement. Together with the positive correlation between code pass count and true code correctness, these density shifts show that CoSPlay improves not only the UT quality but also the quality distribution of the code pool, which ultimately supports stronger BoN selection.

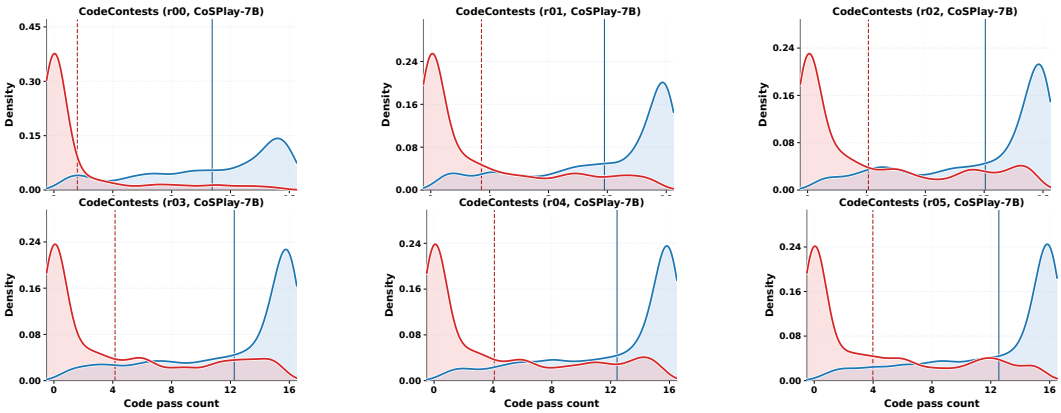


Figure 30: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **CodeContests** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

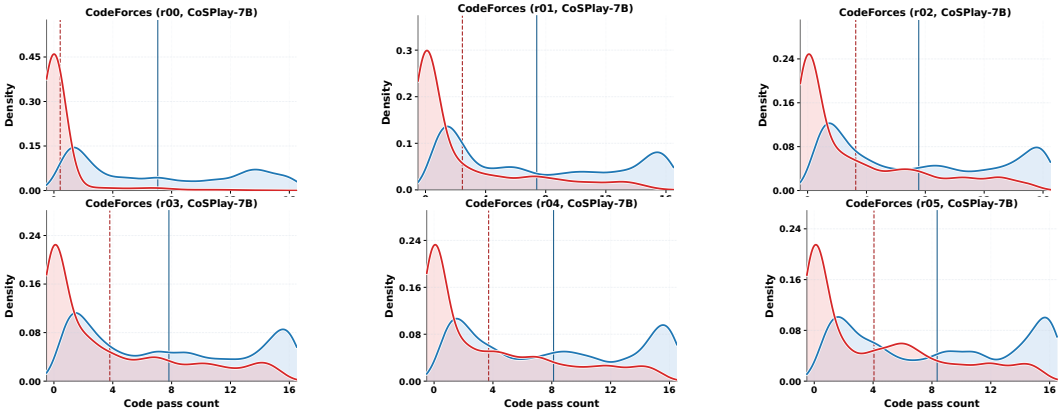


Figure 31: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **CodeForces** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

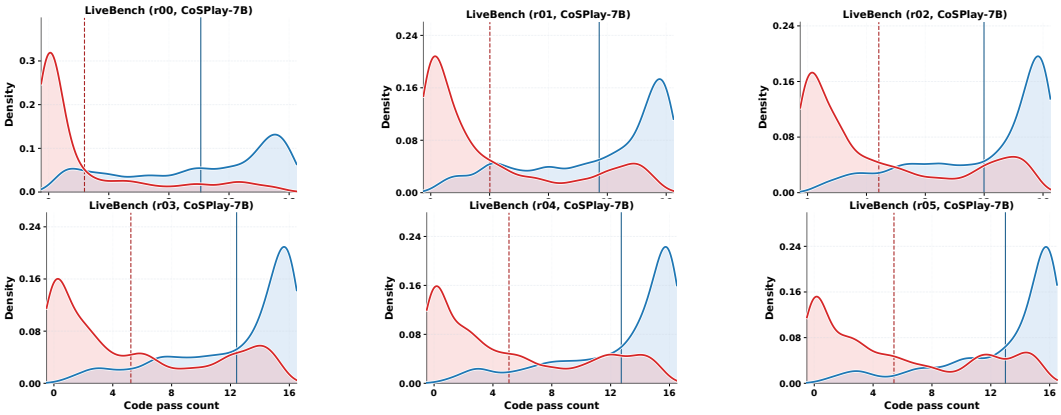


Figure 32: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **LiveBench** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

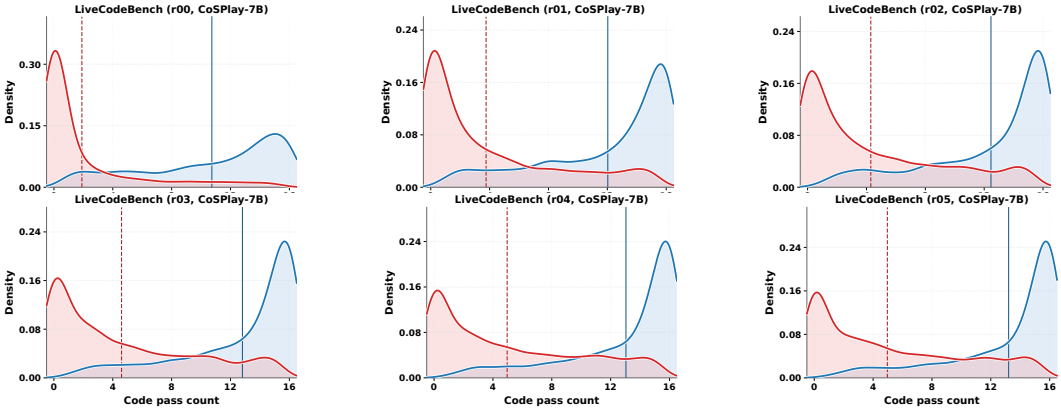


Figure 33: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **LiveCodeBench** with the **7B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

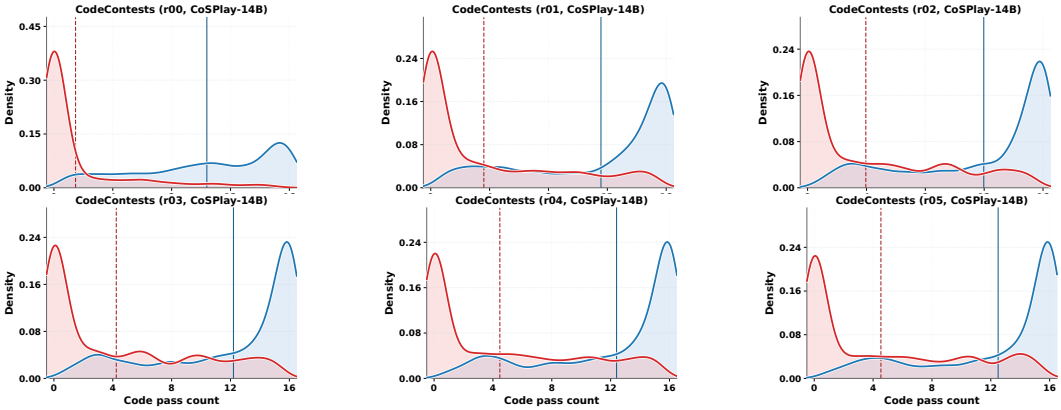


Figure 34: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **CodeContests** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

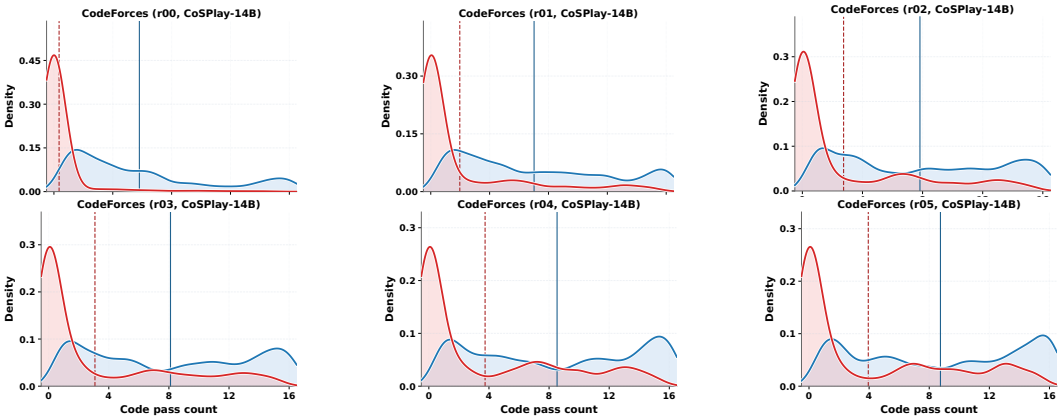


Figure 35: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **CodeForces** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

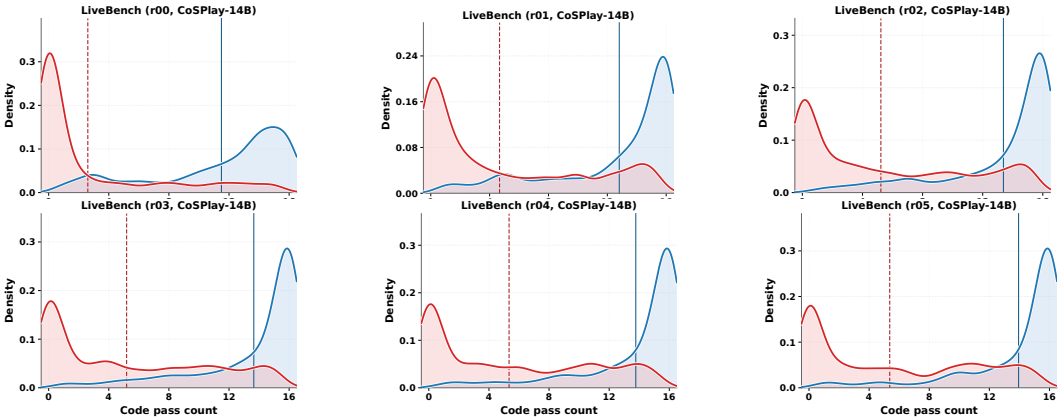


Figure 36: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **LiveBench** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

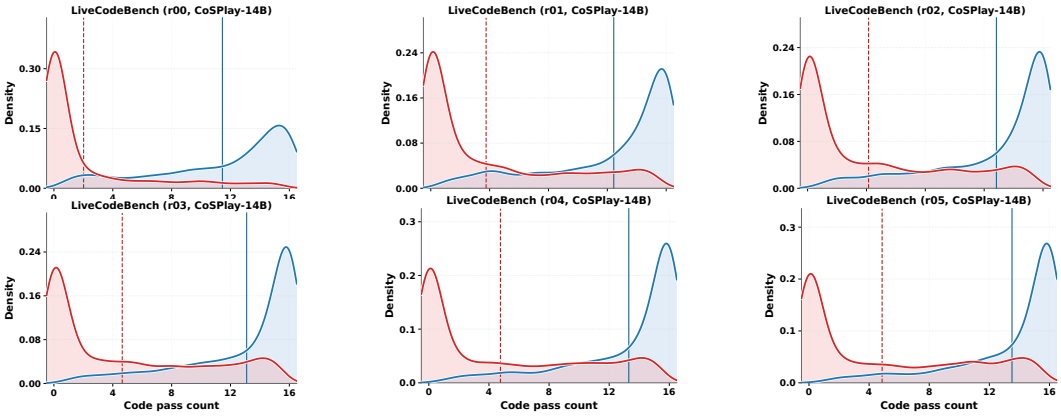


Figure 37: Density distributions of code pass counts for correct (blue) and incorrect (red) code candidates during self-play on **LiveCodeBench** with the **14B model**. The top row shows Round 0-2, and the bottom row shows Round 3-5. Vertical lines mean the average values.

Q Detailed Breakdown of the Relationship between Execution Statistics and True Accuracy

Q.1 Detailed Breakdown of the Relationship between Cluster Size and Code True Accuracy

Figures 38–41 provide per-dataset and per-round results for the relationship between cluster size and average true code accuracy. These plots complement the dataset-level analysis in Figure 10 (right). Larger clusters often exhibit higher correctness, but the strength and shape of this relationship vary across datasets, model scales, and self-play rounds. In particular, the CodeForces curves exhibit non-monotonic behavior, showing that increasing cluster size does not always improve accuracy. These results support output consensus as an empirical signal for code selection while illustrating the limitations of relying on cluster size alone.

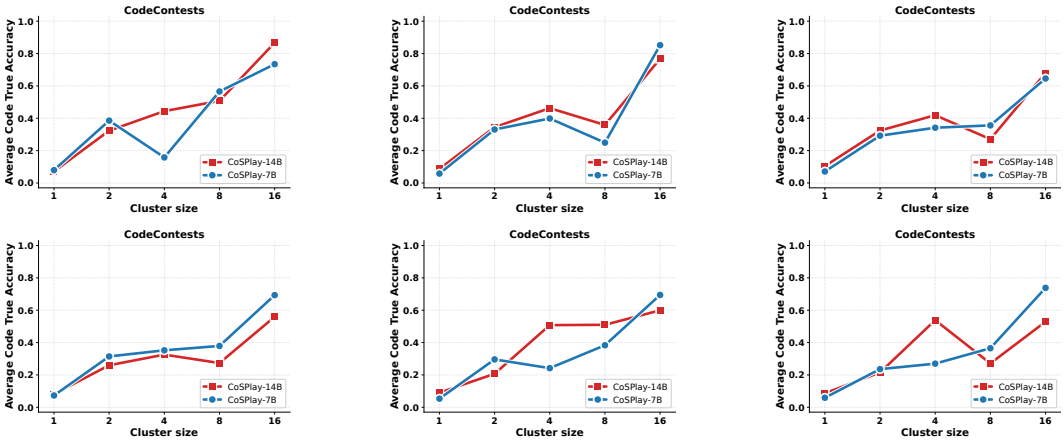


Figure 38: The relationship between cluster size and average code true accuracy during self-play on CodeContests for both 7B and 14B models. The top row shows Round 0-2, and the bottom row shows Round 3-5.

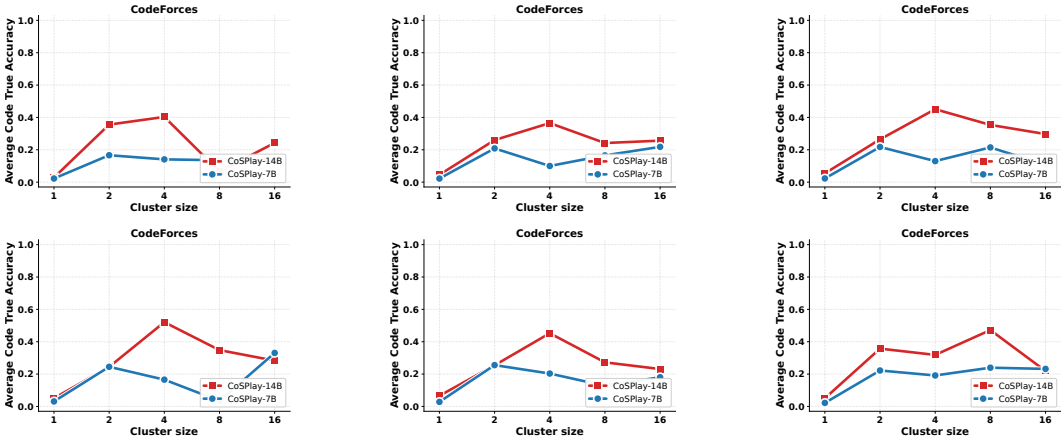


Figure 39: The relationship between cluster size and average code true accuracy during self-play on CodeForces for both 7B and 14B models. The top row shows Round 0-2, and the bottom row shows Round 3-5.

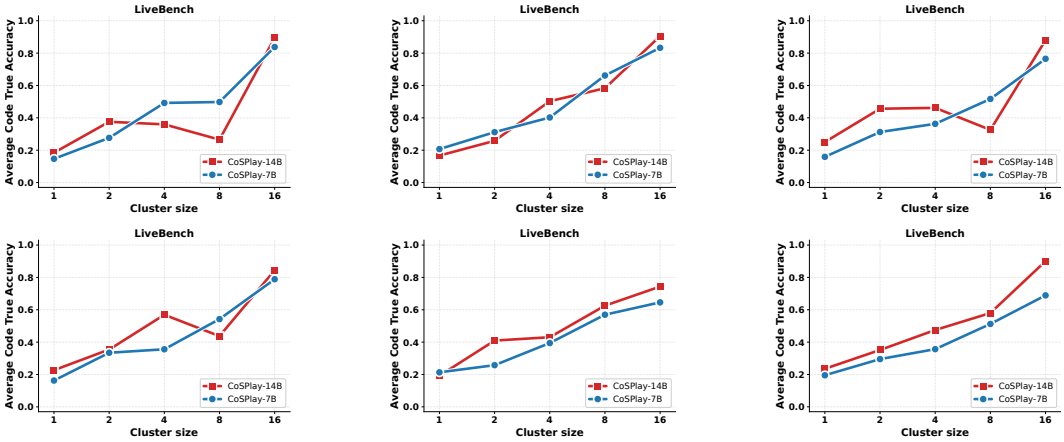


Figure 40: The relationship between cluster size and average code true accuracy during self-play on LiveBench for both 7B and 14B models. The top row shows Round 0-2, and the bottom row shows Round 3-5.

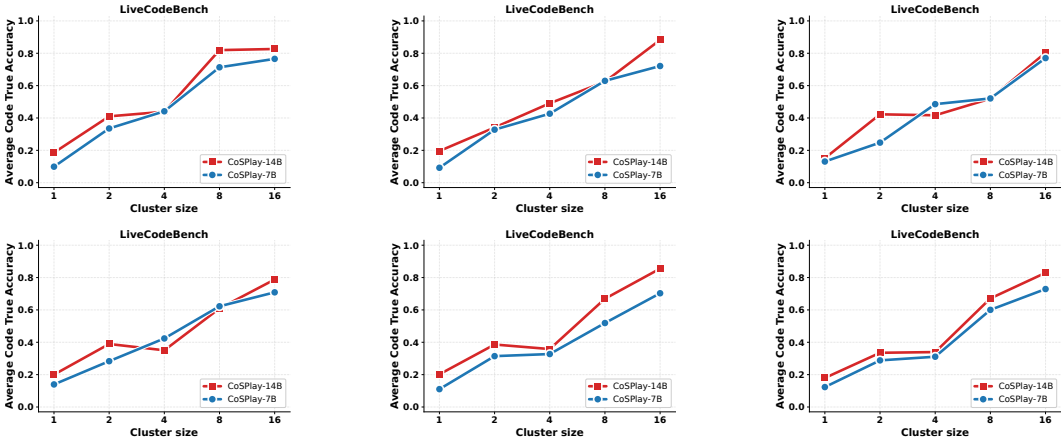


Figure 41: The relationship between cluster size and average code true accuracy during self-play on LiveCodeBench for both 7B and 14B models. The top row shows Round 0-2, and the bottom row shows Round 3-5.

Q.2 Detailed Breakdown of the Relationship between UT Pass Count and True Accuracy

This section provides the detailed data supporting the correlation analysis presented in the main text. Figure 42-45 display the relationship between UT pass counts and their true accuracy for each dataset individually.

We observe that the strong positive correlation discussed in the main text is not an artifact of aggregation; rather, it holds consistently across all benchmarks. As the pass count increases, the reliability of the generated UTs improves uniformly across all tasks. Additionally, the performance gap between the 7B and 14B models is evident in these detailed plots, where the 14B model consistently achieves higher accuracy rates for the same pass counts. This detailed view reinforces the stability of using pass count as a filtering metric across diverse coding challenges.

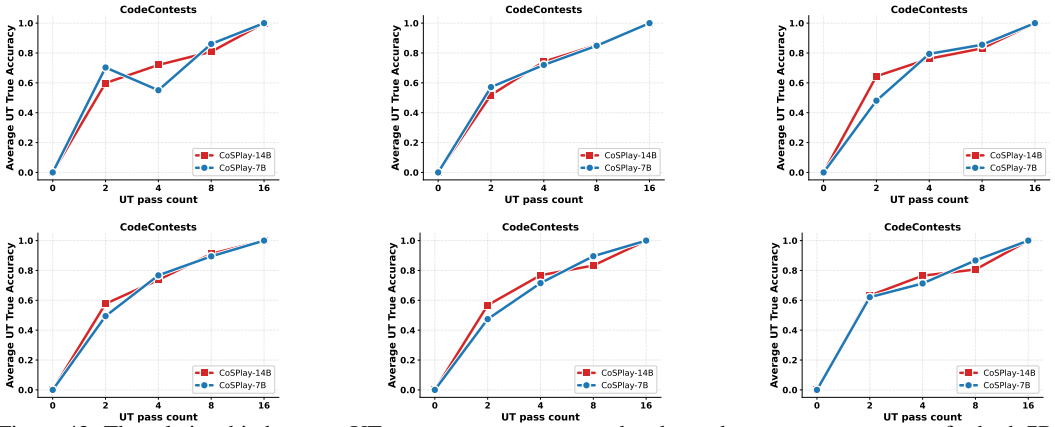


Figure 42: The relationship between UT pass counts on generated codes and average true accuracy for both 7B and 14B models on CodeContests. The top row shows Round 0-2, and the bottom row shows Round 3-5.

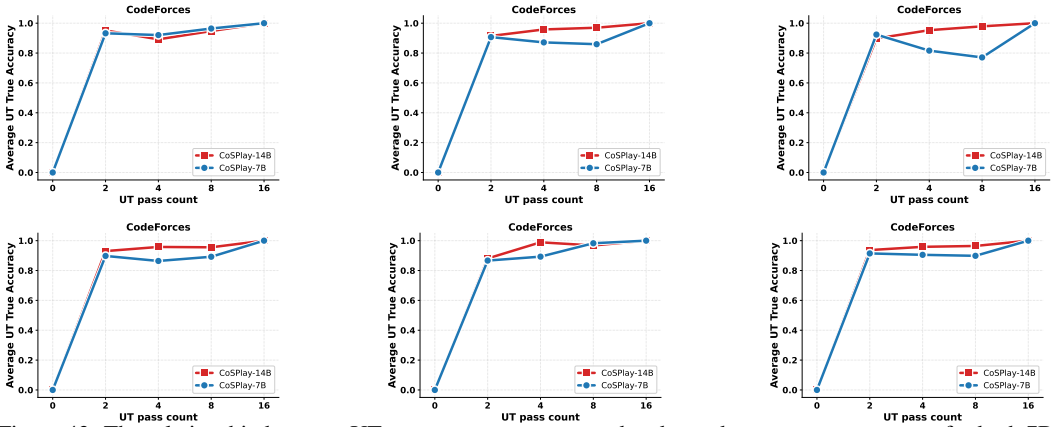


Figure 43: The relationship between UT pass counts on generated codes and average true accuracy for both 7B and 14B models on CodeForces. The top row shows Round 0-2, and the bottom row shows Round 3-5.

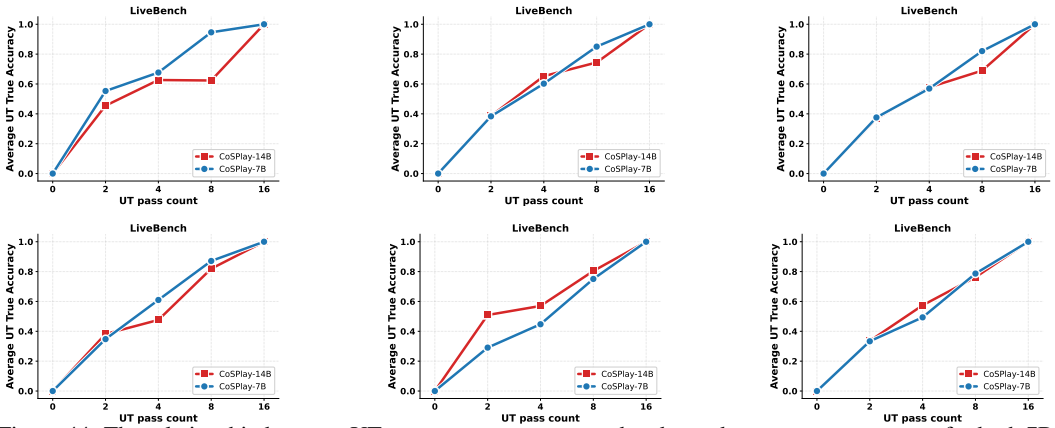


Figure 44: The relationship between UT pass counts on generated codes and average true accuracy for both 7B and 14B models on LiveBench. The top row shows Round 0-2, and the bottom row shows Round 3-5.

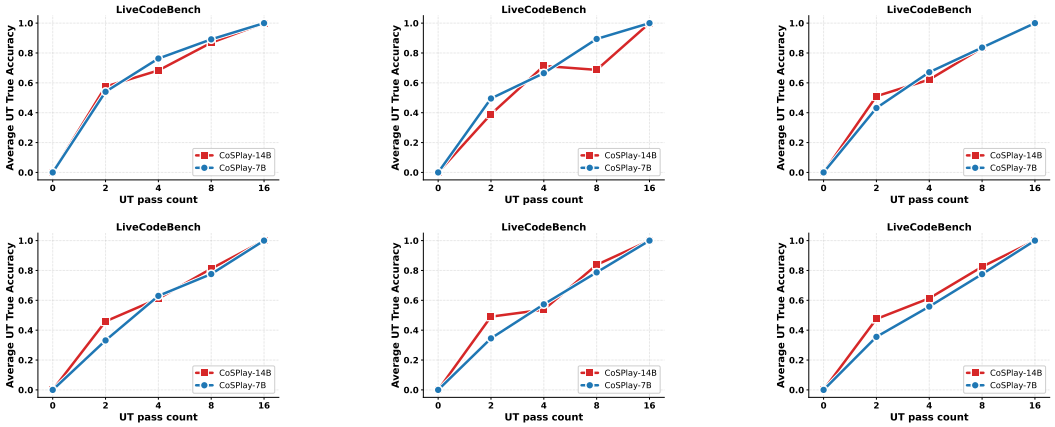


Figure 45: The relationship between UT pass counts on generated codes and average true accuracy for both 7B and 14B models on LiveCodeBench. The top row shows Round 0-2, and the bottom row shows Round 3-5.

Q.3 Detailed Breakdown of the Relationship between Code Pass Count and True Accuracy

Figures 46–49 provide per-dataset and per-round results for the relationship between code pass count and true code accuracy for the 7B and 14B models.

This relationship need not be strictly monotonic, and the relative accuracy of the 7B and 14B models can vary with pass count. These results support using the number of generated UTs passed by a code candidate as a GT-free signal for ranking candidate quality.

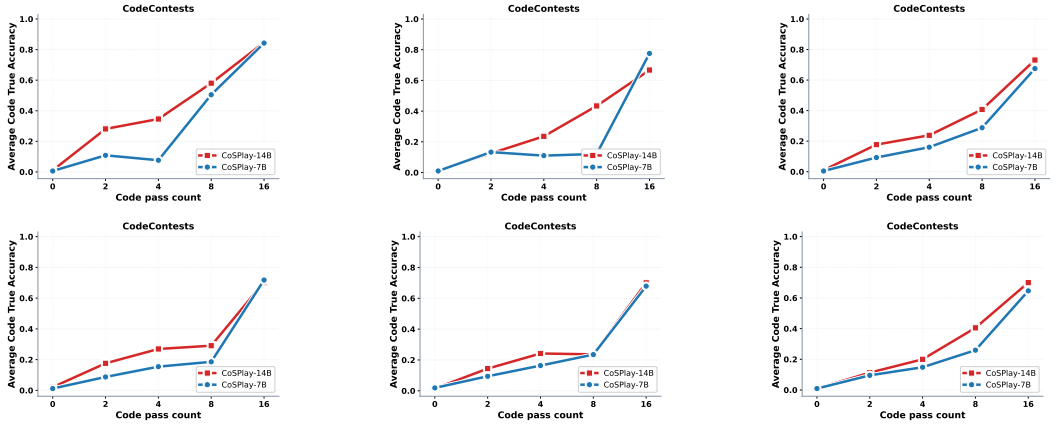


Figure 46: The relationship between code pass counts and average true accuracy for both 7B and 14B models on CodeContests. The top row shows Round 0-2, and the bottom row shows Round 3-5.

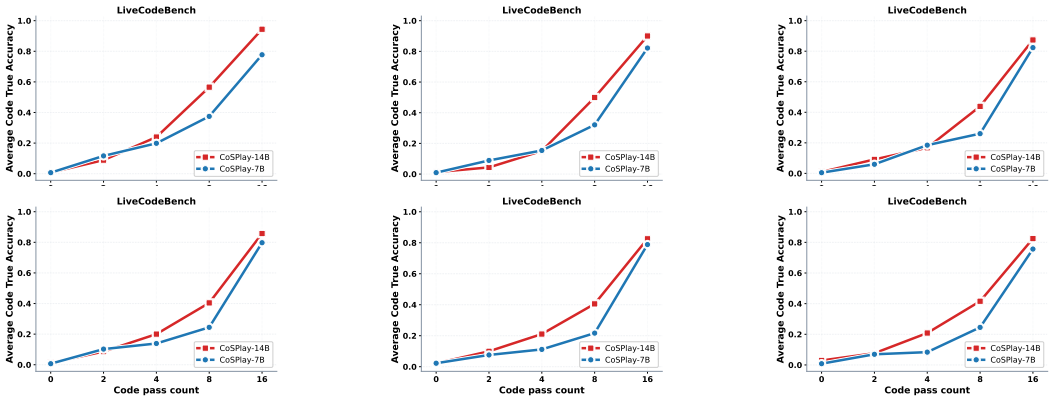


Figure 49: The relationship between code pass counts and average true accuracy for both 7B and 14B models on LiveCodeBench. The top row shows Round 0-2, and the bottom row shows Round 3-5.

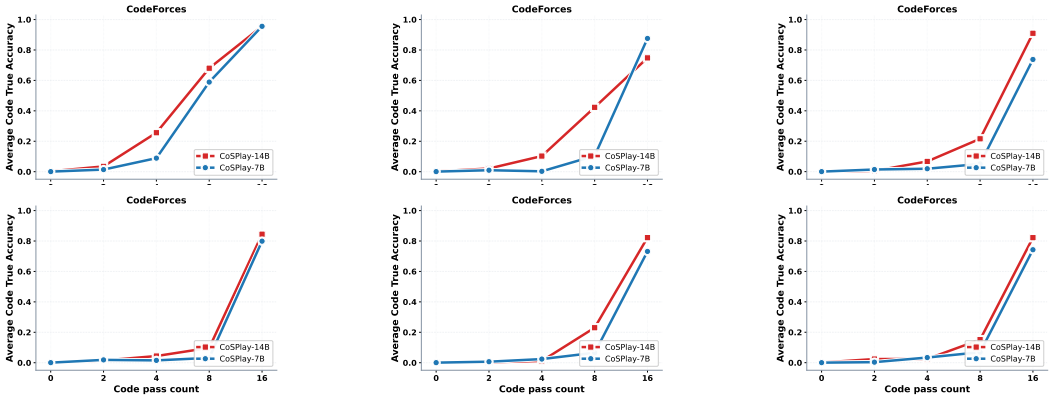


Figure 47: The relationship between code pass counts and average true accuracy for both 7B and 14B models on CodeForces. The top row shows Round 0-2, and the bottom row shows Round 3-5.

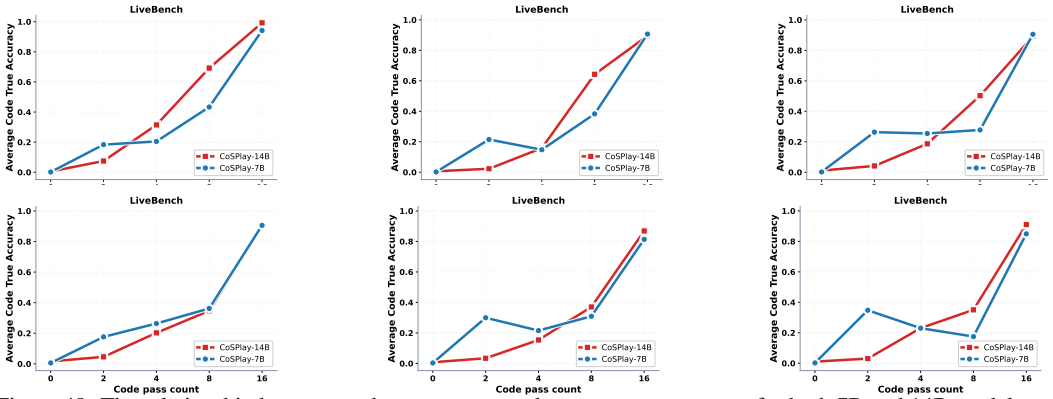


Figure 48: The relationship between code pass counts and average true accuracy for both 7B and 14B models on LiveBench. The top row shows Round 0-2, and the bottom row shows Round 3-5.

R T-SNE VISUALIZATION OF CLUSTERING STRATEGY

To further investigate the effectiveness of our clustering-based selection, Figure 50 provide t-SNE visualizations based on the execution output consistency of generated code candidates. By mapping the output similarity into a low-dimensional space, we observe that correct solutions (blue dots) consistently gravitate toward the largest cluster (indicated by dashed blue circles).



Figure 50: t-SNE visualization of CoSPlay-7B and CoSPlay-14B output signatures.

This observation confirms the core hypothesis of our strategy: functionally equivalent correct solutions yield identical outputs for the same input, thereby exhibiting high consistency. In contrast, wrong solutions (red dots) are either sparsely scattered across the space or form significantly smaller, isolated clusters, reflecting the diverse and unpredictable nature of their failure patterns. Consequently, the visualization demonstrates that the cluster size is a useful empirical proxy for code correctness—the larger the cluster, the higher the output consistency, and thus the greater the likelihood that the candidates within it are correct. This empirical evidence strongly supports our use of clustering to identify and select robust solutions from the candidate pool.

S AVERAGE EVOLUTION OF THE UT AND CODE PASS-COUNT DISTRIBUTIONS ACROSS ITERATIVE SELF-PLAY ROUNDS

Figures 51-54 provide per-dataset results for the evolution of UT and code pass-count distributions at the 7B and 14B scales. Consistent with the averaged trends in the main text, self-play gradually shifts probability mass from low pass-count regions to high pass-count regions, indicating that the execution matrix becomes more structured over iterations. For UTs, the reduction of low pass-count density suggests that self-play removes or refreshes overly hard, invalid, or spuriously coupled tests that provide little reliable supervision. The increased density in high pass-count regions shows that more UTs become broadly satisfiable by the evolving code pool, making them more useful for verification and selection. For codes, the rightward shift indicates that more candidates pass a larger number of UTs after iterative refinement and replacement. This reflects a positive feedback loop between the two pools: better UTs provide more informative execution feedback, while stronger code candidates help validate and refine the UT pool. The consistent trend across datasets supports that CoSPlay improves both candidate quality and selection reliability during self-play.

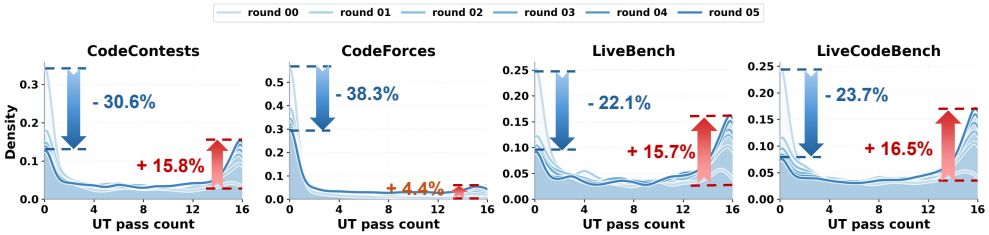


Figure 51: Evolution of UT pass-count distributions during self-play with the 7B model. Curves show per-round density changes across four benchmarks.

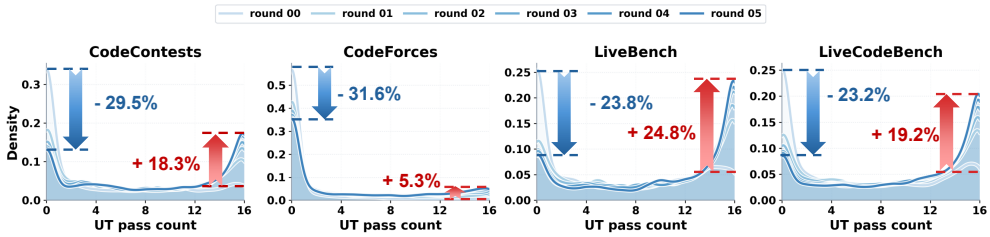


Figure 52: Evolution of UT pass-count distributions during self-play with the 14B model. Curves show per-round density changes across four benchmarks.

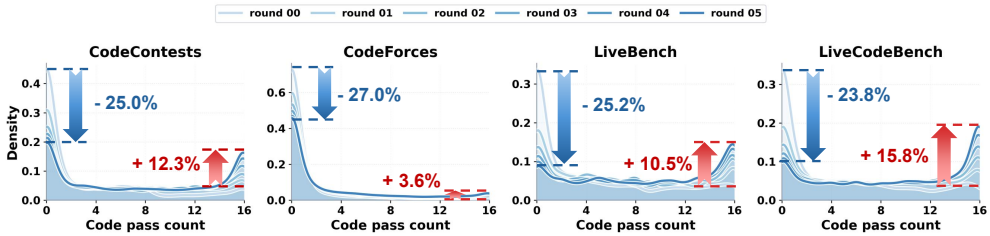


Figure 53: Evolution of code pass-count distributions during self-play with the 7B model. Curves show per-round density changes across four benchmarks.

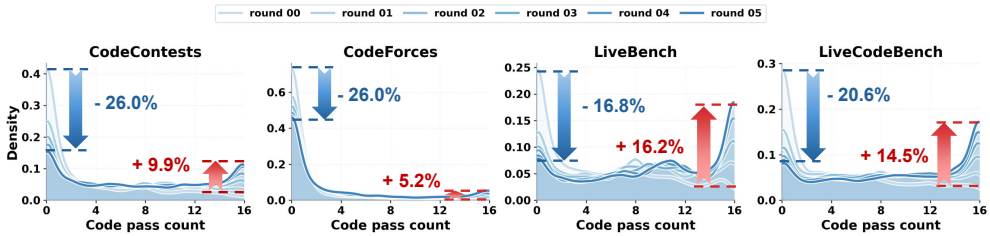


Figure 54: Evolution of code pass-count distributions during self-play with the 14B model. Curves show per-round density changes across four benchmarks.

T Detailed metrics evolution during self-play stage

T.1 Detailed Verifier Accuracy Evolution during Self-Play Rounds

Figure 55 illustrates the evolution of Verifier Accuracy, defined as the BoN accuracy of CoSPlay-generated UTs when used to select Qwen2.5-7B-Instruct-generated code candidates. Both the 7B and 14B exhibit a generally upward trend across all datasets, confirming that iterative self-play progressively enhances the discriminative power of the evolved UT pool. Introducing cluster-based selection (+ Cluster) consistently yields higher Verifier Acc. than the BoN-only variant, demonstrating that cluster selection further improves the reliability of UT-guided candidate selection. The 14B model achieves higher Verifier Accuracy than the 7B model across all datasets and rounds, reflecting the benefit of greater model capacity in generating more accurate and discriminative UTs. Among datasets, *CodeForces* remains the most challenging, showing the lowest absolute Verifier Acc. values throughout.

T.2 Detailed BoN evolution during self-play rounds

Figure 56 illustrates the evolution of BoN Accuracy across self-play rounds. Both the 7B and 14B models show a consistent upward trend across all four datasets, confirming that iterative self-play steadily improves the quality of the code candidate pool. The most notable gains occur in the early rounds, after which performance continues to grow at a more gradual pace. Applying cluster-based selection (+ Cluster) consistently improves BoN Accuracy over the plain BoN variant, validating the effectiveness of execution-consensus-based selection as a complementary mechanism. As expected, the 14B model outperforms the 7B model across all datasets and rounds. *CodeForces* remains the most challenging benchmark with the lowest absolute BoN scores, while *LiveBench* and *LiveCodeBench* yield comparatively higher accuracy for both models.

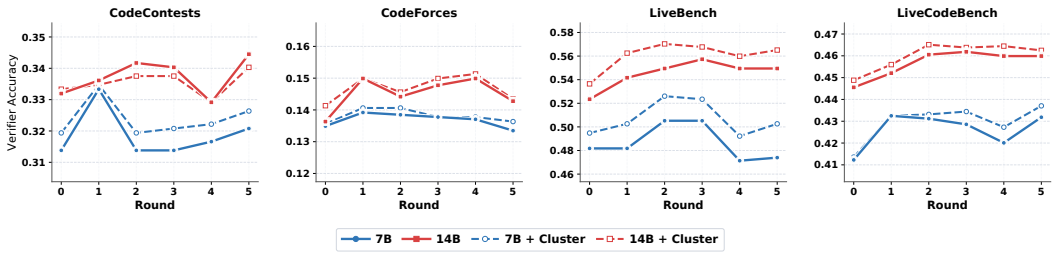


Figure 55: Evolution of **Verifier Accuracy** across iterative self-play rounds.

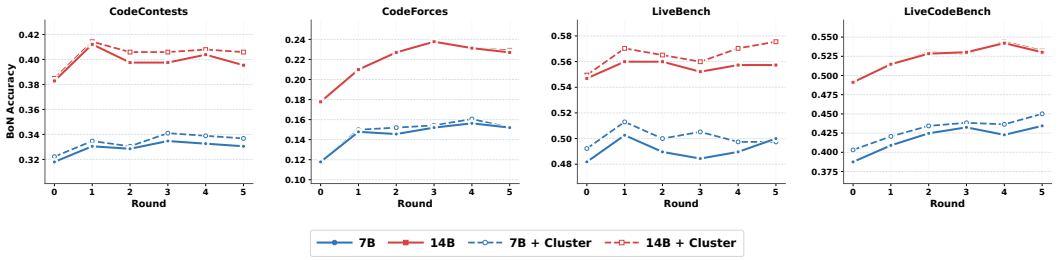


Figure 56: Evolution of **Best-of-N (BoN) accuracy** evaluated on four benchmarks during self-play rounds.

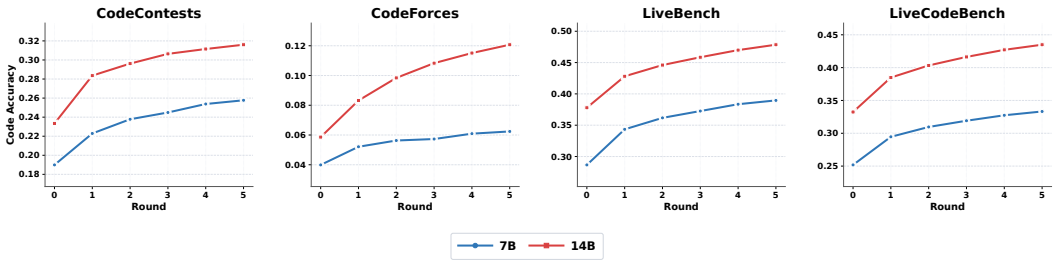


Figure 57: Evolution of **Code Accuracy** across iterative self-play rounds.

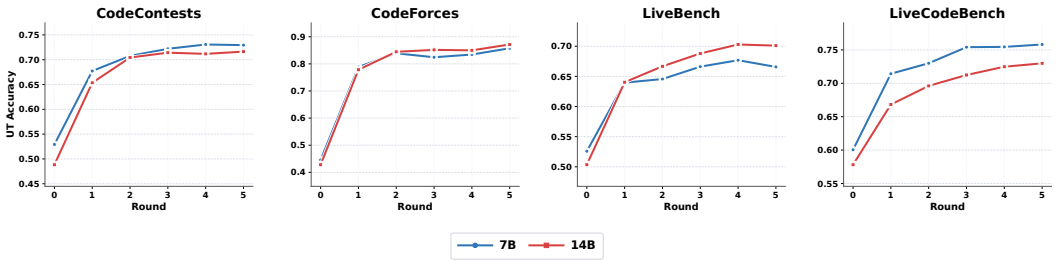


Figure 58: Evolution of **Unit Test (UT) Accuracy** across iterative self-play rounds

T.3 Detailed Code accuracy evolution during self-play rounds

Figure 57 illustrates the trajectory of Code Accuracy. Both the 7B and 14B models show a consistent upward trend across all datasets, confirming the effectiveness of iterative self-play. The most substantial gain occurs between Round 0 and Round 1, followed by steady, gradual growth. Notably, both models achieve their highest accuracy on *LiveBench* and *LiveCodeBench*, while *CodeForces* remains the most challenging with the lowest overall scores. As expected, the 14B model consistently outperforms the 7B model across all stages.

T.4 Detailed UT accuracy evolution during self-play rounds

Figure 58 presents the evolution of Unit Test (UT) Accuracy. The defining characteristic across all datasets is a dramatic surge from Round 0 to Round 1, indicating that our self-play stage can filter

Table 15: **Contamination-controlled evaluation on the newest LiveCodeBench v6 window (25.02–25.05). Verif.** denotes the verifier accuracy of the generated unit tests on a fixed code pool. **UT** and **Code** report the average accuracy of generated unit tests and code solutions, respectively. **BoN** reports the accuracy of the final selected code. CoSPlay uses cluster selection. Bold scores indicate the best performance within each model. Δ denotes the percentage-point gain of CoSPlay over Baseline.

Model	Method	LiveCodeBench v6 (25.02–25.05)			
		Verif.	UT	Code	BoN
Qwen2.5-7B-Instruct	Baseline	21.4	20.6	18.9	21.4
	CoSPlay (Ours)	**28.2**	**78.1**	**22.6**	**29.0**
	Δ vs. Baseline	**+6.8**	**+57.5**	**+3.7**	**+7.6**
Qwen2.5-14B-Instruct	Baseline	26.0	44.8	24.0	29.0
	CoSPlay (Ours)	**29.0**	**71.4**	**27.7**	**32.8**
	Δ vs. Baseline	**+3.0**	**+26.6**	**+3.7**	**+3.8**
Qwen3-4B	Baseline	**29.8**	71.8	31.6	42.0
	CoSPlay (Ours)	**29.8**	**97.0**	**36.2**	**45.0**
	Δ vs. Baseline	**+0.0**	**+25.2**	**+4.6**	**+3.0**

Table 16: **Token-budget scaling of CoSPlay on Qwen3-4B [3×200].** Each column corresponds to one self-play round (R0–R5) and the token budget it consumes. **Budget** is the number of tokens per problem in millions. **Baseline** spends the same budget on sampling more codes and UTs and executes the full Code-UT matrix. **Acc.** is the BoN accuracy in percent; Δ is the gain over the baseline in percentage points. **Exec.** is the number of code-UT executions per problem; **Exec. reduction** is the relative change of CoSPlay over the baseline, where negative values indicate fewer executions. Bold marks the best accuracy at each budget.

Round	R0	R1	R2	R3	R4	R5
Budget (M)	0.871	1.255	1.542	1.803	2.051	2.294
Baseline Acc.	64.67	65.83	65.00	**65.83**	65.17	65.17
CoSPlay Acc.	**64.83**	**67.00**	**66.17**	**65.83**	**66.67**	**67.67**
Δ vs. Baseline	**+0.17**	**+1.17**	**+1.17**	**0.00**	**+1.50**	**+2.50**
Baseline Exec.	2,809	5,929	8,836	12,100	15,625	16,384
CoSPlay Exec.	**512**	**1,792**	**3,072**	**4,352**	**5,632**	**6,912**
Exec. reduction (%)	**-81.80**	**-69.80**	**-65.20**	**-64.00**	**-64.00**	**-57.80**

wrong UT and inject correct UT into the UT pool. This leap is most prominent on *CodeForces*, which jumps from the lowest to the highest accuracy. Following this initial surge, performance largely plateaus with only marginal fluctuations. The overall trajectories are remarkably similar for both models, though the 14B model achieves slightly higher peak values.

U CoSPlay on the Latest LiveCodeBench v6

The gains of CoSPlay are not an artifact of benchmark contamination. A potential concern for GT-free test-time scaling is that a contaminated backbone may reproduce memorized tests rather than generate genuinely discriminative ones. To control for this, we follow the official Qwen evaluation protocol and evaluate on the latest LiveCodeBench v6 window (25.02–25.05). This window is adopted officially precisely because it postdates the data used to build these model families, making it a standard contamination-controlled setting. As shown in Table 15, CoSPlay improves BoN on all three backbones, by 4.8% on average, which is consistent with the gains observed on our main

benchmarks. The improvement on the UT side is particularly large: UT accuracy rises from 20.6% to 78.1% on Qwen2.5-7B-Instruct and from 71.8% to 97.0% on Qwen3-4B, together with consistent gains in Verifier Acc. and code accuracy. Therefore, these improvements in UT quality stem from exploration-attack idea generation and execution-driven co-evolution. These results suggest that CoSPlay improves UT and code quality through its verification mechanism rather than through contamination, and that its effectiveness holds on the most recent problems.

V CoSPlay under Equal Token Budgets on Reasoning Model.

CoSPlay is complementary to extended reasoning and remains effective under matched token budgets. Reasoning models scale test-time compute along a different axis, generating longer chains of thought before answering. To examine whether CoSPlay remains useful along this axis, we apply it to Qwen3-4B in thinking mode and match the token budget of each self-play round with a baseline that spends the same budget on the reasoning model's own sampling. As shown in Table 16, CoSPlay matches or exceeds the baseline accuracy at every tested budget, although both its accuracy and its margin over the baseline fluctuate across intermediate budgets. At the highest tested budget (R5), it reaches its best accuracy of 67.7% and its largest advantage of 2.5 percentage points. CoSPlay also uses 57.8% to 81.8% fewer executions than the budget-matched baseline at every round. These results suggest that CoSPlay provides an effective form of test-time scaling for reasoning models, achieving better accuracy at the highest matched token budget with fewer executions.

W CASE STUDY

W.1 CASE STUDY OF SUCCESSFUL CODE FIXING

W.1.1 COSPLAY-7B

Question for CoSPlay-7B

A balance scale tips to the left **if** $L > R$, where L **is** the total weight of the masses on the left pan **and** R **is** the total weight of the masses on the right pan. Similarly, it balances **if** $L = R$, **and** tips to the right **if** $L < R$. Takahashi placed a mass of weight A **and** a mass of weight B on the left pan of a balance scale, **and** placed a mass of weight C **and** a mass of weight D on the right pan. Print `Left` **if** the balance scale tips to the left; **print** `Balanced` **if** it balances; **print** `Right` **if** it tips to the right.

Constraints

- $1 \leq A, B, C, D \leq 10$
- All **input** values are integers.

Input

Input **is** given **from** Standard Input **in** the following **format**:

A B C D

Output

Print `Left` **if** the balance scale tips to the left; **print** `Balanced` **if** it balances; **print** `Right` **if** it tips to the right.

Examples:
Input
3 8 7 1

Output
Left

Input
3 4 5 2

Output
Balanced

Input
1 7 6 4

Output
Right

Buggy code from CoSPlay-7B

```
# Read input from standard input
A, B, C, D = map(int, input().split())

# Calculate the sum of reciprocals for the left and right pans
left_sum = 1/A + 1/B
right_sum = 1/C + 1/D
```

```
# Determine the result based on the comparison of the sums
if left_sum > right_sum:
    print("Left")
elif left_sum == right_sum:
    print("Balanced")
else:
    print("Right")
```

Generated UT from CoSPlay-7B

Attack idea:
This case targets buggy solutions that compare an incorrect quantity **for** each side of the balance scale instead of comparing the total weights. With $A = 10$ **and** $B = 10$, the left pan has total weight 20. With $C = 1$ **and** $D = 1$, the right pan has total weight 2. Therefore, the correct answer **is** `Left`. The original buggy code compares reciprocal sums, which reverses the decision on this **input and** produces `Right`.

Input:
10 10 1 1

Expected output:
Left

Buggy-code output:
Right

Fixed-code output:
Left

UT Explanation:
The total weight on the left pan **is** $A + B = 10 + 10 = 20$, **while** the total weight on the right pan **is** $C + D = 1 + 1 = 2$. Since $20 > 2$, the balance scale tips to the left, so the correct output **is** `Left`. This UT exposes the buggy implementation because comparing reciprocal sums gives $1/10 + 1/10 = 0.2$ on the left **and** $1/1 + 1/1 = 2$ on the right, which incorrectly leads the buggy code to output `Right`.

Fixed Code from CoSPlay-7B

```
# Read input from standard input
A, B, C, D = map(int, input().split())

# Calculate the total weight for the left and right pans
left_sum = A + B
right_sum = C + D

# Determine the result based on the comparison of the sums
if left_sum > right_sum:
    print("Left")
elif left_sum == right_sum:
    print("Balanced")
else:
    print("Right")
```

Fixed-Code Explanation

The failure **type is** a logical error. The original code runs successfully but produces the wrong answer because it computes `1/A + 1/B` **and** `1/C + 1/D` instead of comparing the actual total weights. The correct logic **is** to compute `A + B` **for** the left pan **and** `C + D` **for** the right pan, then compare these two values. On the UT below, the fixed code outputs `Left`, matching the correct expected output.

W.1.2 CoSPlay-14B

Question for CoSPlay-14B

Ball **is** the teacher **in** Paperfold University. The seats of his classroom are arranged **in** 2 rows **with** m seats each.

Ball **is** teaching $a+b+c$ monkeys, **and** he wants to assign **as** many monkeys to a seat **as** possible. Ball knows that a of them only want to sit **in** row 1, b of them only want to sit **in** row 2, **and** c of them have no preference. Only one monkey may sit **in** each seat, **and** each monkey's preference must be followed if it is seated.

What is the maximum number of monkeys that Ball can seat?

Input format:
The first line contains an integer t ($1 \le t \le 10^4$) – the number of test cases.

Each test case contains four integers m, a, b, and c ($1 \le m, a, b, c \le 10^8$).
Output format:
For each test case, output the maximum number of monkeys you can seat.
Example Input 0:

```
5
10 5 5 10
3 6 1 1
15 14 12 4
1 1 1 1
420 6 9 69
```

Example Output 0:

```
20
5
30
2
84
```

Notes:
In the second test case, 6 monkeys want to sit in the front row, but only 3 seats are available. The monkeys that have no preference and the monkeys who prefer sitting in the second row can sit in the second row together. Thus, the answer is $3+2=5$.

Buggy Code from CoSPlay-14B

```
def max_seats(m, a, b, c):
    # Limit monkeys preferring row 1 and row 2 to the available seats
    a = min(a, m)
    b = min(b, m)
```

```
    # Remaining seats after seating the monkeys with preferences
    remaining_seats = m - (a + b)

    # Monkeys with no preference can be seated in either row
    seated_monkeys = a + b

    # Add as many no-preference monkeys as possible
    if remaining_seats >= c:
        seated_monkeys += c
    else:
        seated_monkeys += remaining_seats

    return seated_monkeys

t = int(input())
for _ in range(t):
    m, a, b, c = map(int, input().split())
    print(max_seats(m, a, b, c))
```

Generated UT from CoSPlay-14B

Attack Idea:
If c **is** larger than m but a **and** b are both smaller than m, the algorithm might incorrectly assume **all** seats can be filled without considering the actual number of available seats.

Input:

```
1
10 4 4 11
```

Test output:
19

Buggy-code output:
10

Fixed-code output:
19

UT Explanation:

1. We have 1 test case **with** the **input** values $m = 10$, $a = 4$, $b = 4$, **and** $c = 11$.
2. There are 10 seats **in** each of the 2 rows, so a total of 20 seats.
3. a monkeys (4 monkeys) want to sit **in** row 1.
4. b monkeys (4 monkeys) want to sit **in** row 2.
5. c monkeys (11 monkeys) have no preference.
6. We need to maximize the number of monkeys seated, following their seating preferences.
7. We first seat the a monkeys **in** row 1 **and** b monkeys **in** row 2, which takes up 4 seats **in** each row.
8. There are $10 - 4 = 6$ seats left **in** row 1 **and** $10 - 4 = 6$ seats left **in** row 2.
9. We now seat the c monkeys, starting **with** those **with** no preference:
    - We can seat up to 6 of these monkeys **in** row 1 since there are 6 seats left.
    - After seating 6 monkeys **in** row 1, we have $11 - 6 = 5$ monkeys left.
    - We can then seat the remaining 5 monkeys **in** row 2 since there are 6 seats left.

10. Therefore, we have seated 4 monkeys **in** row 1, 4 monkeys **in** row 2, **and** 6 monkeys **in** row 1 plus 5 monkeys **in** row 2, which totals to $4+4+6+5=19$ monkeys.

Fixed Code from CoSPlay-14B

Analysis:
The error **is** a logical error (wrong answer), **as** the code runs but produces the wrong output. Let's trace the logic to find the divergence from the expected output.

Given:
- The input is `10 4 4 11`, where `m = 10`, `a = 4`, `b = 4`, and `c = 11`.
- We are trying to maximize the number of seated monkeys by considering monkeys ' preferences **and** available seats.

The code fails because it does **not** correctly handle the scenario where there are more no-preference monkeys (c) than the remaining seats after seating monkeys **with** preferences. Specifically, the logic does **not** account **for** the optimal distribution of no-preference monkeys to fill the available seats efficiently.

Let's walk through the correct logic:
1. If `a > m`, then the number of monkeys that can sit in row 1 is limited to `m`.

2. If `b > m`, then the number of monkeys that can sit in row 2 is limited to `m`.

3. The remaining seats after seating monkeys with preferences are `2 * m - (a + b)`.
4. If there are more no-preference monkeys than remaining seats, only the available seats can be filled.

Fixed Code

```
def max_seats(m, a, b, c):
    # Limit monkeys preferring row 1 and row 2 to the available seats
    a = min(a, m)
    b = min(b, m)

    # Calculate the remaining seats after seating monkeys with preferences
    remaining_seats = 2 * m - (a + b)

    # Monkeys with no preference can be seated in either row
    seated_monkeys = a + b

    # Add as many no-preference monkeys as possible
    if remaining_seats >= c:
        seated_monkeys += c
    else:
        seated_monkeys += remaining_seats

    return seated_monkeys

t = int(input())
for _ in range(t):
    m, a, b, c = map(int, input().split())
    print(max_seats(m, a, b, c))
```

Fixed-Code Explanation

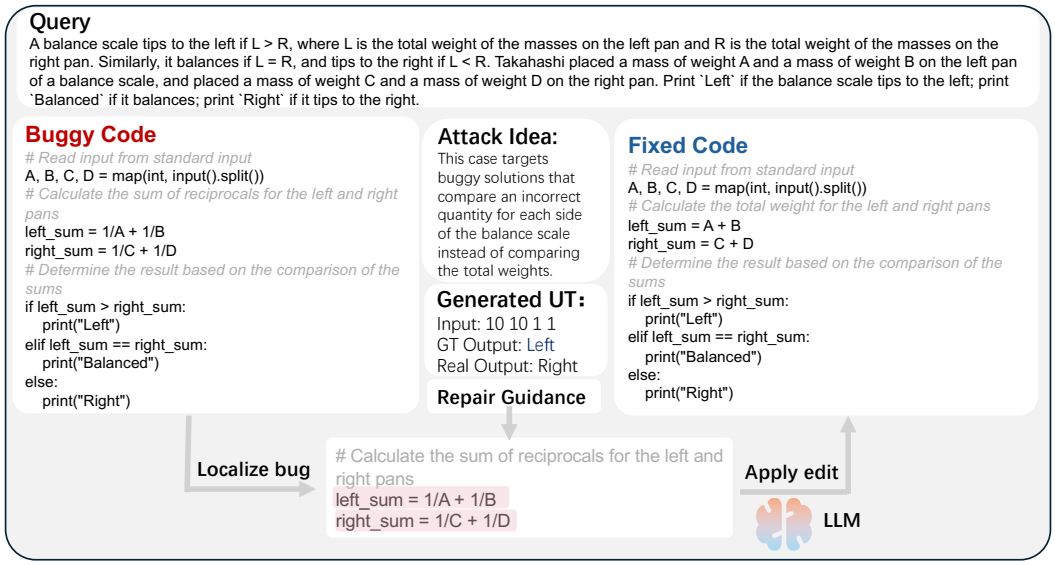
```
- We first limit `a` and `b` to the number of seats in their respective rows.
- Calculate the total remaining seats after seating `a` and `b` as `
remaining_seats = 2 * m - (a + b)`.
- Add the maximum number of no-preference monkeys that can be seated based on
the remaining seats.

This should correctly handle the given test cases and others as well.
```

Figure 59 presents two concrete examples of successful code refinement driven by the non-trivial best UT selected in Step 3.

Example 1: Balance Scale Comparison. The buggy code attempts to solve a simple balance-scale problem, where the correct decision should be based on comparing the total weights on the two sides, i.e., `A + B` versus `C + D`. However, the implementation mistakenly compares the sums of reciprocals, `1/A + 1/B` and `1/C + 1/D`, which can lead to an incorrect ordering for many inputs. The generated UT directly targets this semantic error with `A = 10`, `B = 10`, `C = 1`, and `D = 1`. The expected output is `Left`, since the left side has total weight `20` while the right side has total weight `2`. Nevertheless, the buggy code outputs `Right`, because its reciprocal-based comparison gives `0.2` on the left and `2` on the right. The refinement guidance correctly localizes this erroneous comparison and replaces the reciprocal sums with direct weight sums, producing a semantically correct fixed code.

Example 2: Two-Row Monkey Seating. The buggy code correctly caps the row-preference groups by their row capacities, but then miscomputes the remaining capacity as `m - (a + b)`, effectively treating the classroom as if it had only one row. The generated UT exposes this bug with `m = 10`, `a = 4`, `b = 4`, and `c = 11`, where the expected answer is `19` because there are `20` total seats and all `11` no-preference monkeys can still be seated after placing the `4` row-1 and `4` row-2 monkeys. However, the buggy code outputs `10`, since it incorrectly believes only `2` seats remain. The refinement guidance fixes the capacity calculation by replacing `m - (a + b)` with `2 * m - (a + b)`, producing a semantically correct fixed code.

Summary. Both examples show that Step 3 can provide precise refinement signals when the selected UT exposes a concrete semantic mismatch between the buggy implementation and the problem requirement. In the balance-scale example, the UT reveals that the code compares reciprocal values instead of total weights; in the monkey-seating example, the UT reveals that the code computes remaining capacity using only one row instead of two. In both cases, the failure is not caused by formatting or invalid inputs, but by a targeted test that activates the exact erroneous logic. This allows the LLM to perform a localized correction—replacing the wrong comparison formula in the first case and the wrong capacity formula in the second—rather than rewriting the entire solution from scratch, demonstrating the effectiveness of Step 3's targeted refinement strategy.



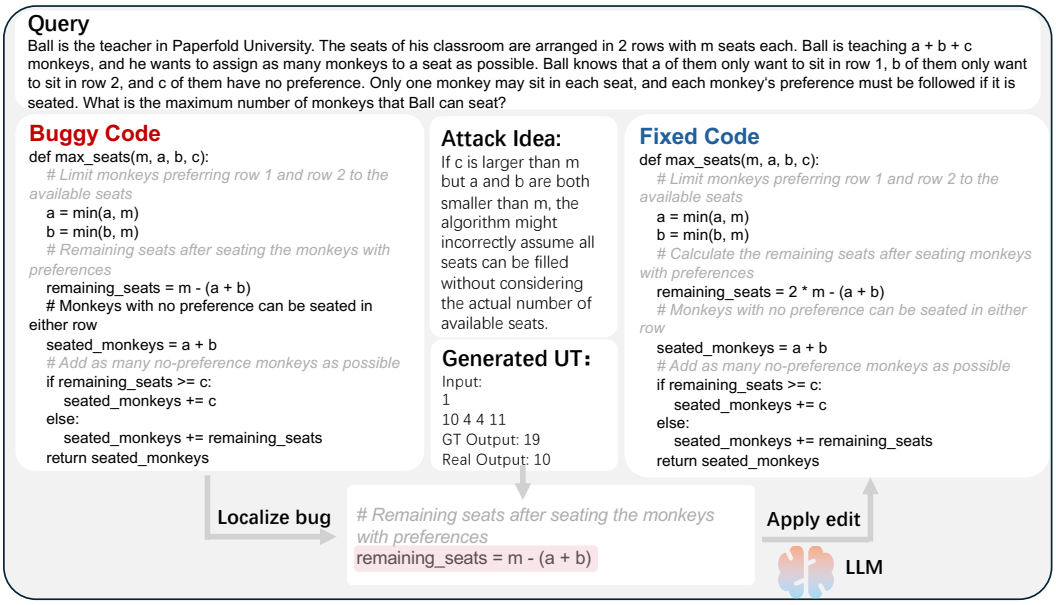


Figure 59: Case study of successful code fix.

W.2 Case study of successful locating Code–UT Coupling

W.2.1 CoSPlay-7B

Question for CoSPlay-7B

Given a **set** of N axis-aligned rectangular seals, find the number of overlapped seals on the region which has the maximum number of overlapped seals.

Constraints

- $1 \leq N \leq 100000$
- $0 \leq x_{1,i} < x_{2,i} \leq 1000$
- $0 \leq y_{1,i} < y_{2,i} \leq 1000$
- $x_{1,i}, y_{1,i}, x_{2,i}, y_{2,i}$ are given **in** integers

Input

The **input is** given **in** the following **format**.

N
$x_{1,1}$ $y_{1,1}$ $x_{2,1}$ $y_{2,1}$
$x_{1,2}$ $y_{1,2}$ $x_{2,2}$ $y_{2,2}$
:
$x_{1,N}$ $y_{1,N}$ $x_{2,N}$ $y_{2,N}$

$(x_{1,i}, y_{1,i})$ **and** $(x_{2,i}, y_{2,i})$ are the coordinates of the top-left **and** the bottom-right corner of the i-th seal respectively.

Output

Print the maximum number of overlapped seals **in** a line.

Examples:

Input
2
0 0 3 2
2 1 4 3
Output
2

Input
2
0 0 2 2
2 0 4 2
Output
1

Input
3
0 0 2 2
0 0 2 2
0 0 2 2
Output
3

Wrong Code for CoSPlay-7B

```
def read_rectangle():
    x1, y1, x2, y2 = map(int, input().split())
    return (x1, y1, x1, y2), (x2, y1, x2, y2)

def preprocess_rectangles():
    rectangles = []
    for _ in range(int(input())):
        top_left, bottom_right = read_rectangle()
        rectangles.append(top_left)
        rectangles.append(bottom_right)
    rectangles.sort(key=lambda x: (x[0], x[1]))
    return rectangles

def count_overlaps(rectangles):
    events = []
    for (x1, y1, x2, y2), (x3, y3, x4, y4) in zip(rectangles, rectangles[1:]):
        events.append((x1, 'start', y1))
        events.append((x4, 'end', y4))

    # Sort events by x-coordinate, and by y-coordinate for 'start' events
    events.sort(key=lambda x: (x[0], -1 if x[1] == 'start' else 1, x[2]))

    max_overlaps = 0
    current_overlaps = 0

    for x, event_type, y in events:
        if event_type == 'start':
            current_overlaps += 1
            max_overlaps = max(max_overlaps, current_overlaps)
        else:
            current_overlaps -= 1

    return max_overlaps

def main():
    rectangles = preprocess_rectangles()
    print(count_overlaps(rectangles))

if __name__ == "__main__":
    main()
```

Erroneously coupled UT for CoSPlay-7B

Input:
5
1 1 4 4
2 2 5 5
0 0 3 3
3 3 6 6
2 1 4 3

Generated expected output:
3
Buggy-code output:
3
Correct-reference output:
4

New generated UT for CoSPlay-7B

Input:
3
0 0 1 2
1 0 2 2
0 1 1 3

Generated expected output:
3

Buggy-code output:
4

W.2.2 CoSPlay-14B

Question for CoSPlay-14B

Given are two integer sequences of N elements each: $A_1, \ldots, A_N$ **and** $B_1, \ldots, B_N$. Determine **if** it **is** possible to do the following operation at most $N-2$ times (possibly zero) so that **for** every integer i **from** 1 to N, $A_i \leq B_i$ holds:

- Choose two distinct integers x **and** y between 1 **and** N (inclusive), **and** swap the values of A_x **and** A_y.

Constraints

- $2 \leq N \leq 10^5$
- $1 \leq A_i, B_i \leq 10^9$

Input

Input **is** given **from** Standard Input **in** the following **format**:

N
A_1 A_2 ... A_N
B_1 B_2 ... B_N

Output

If the objective **is** achievable, **print** `Yes`; **if** it **is not**, **print** `No`.

Examples

Input
3
1 3 2
1 2 3

Output
Yes

Input
3
1 2 3
2 2 2

Output
No

```
Input
6
3 1 2 6 3 4
2 2 8 3 4 3

Output
Yes
```

Wrong Code for CoSPlay-14B

```python
# Read input
N = int(input())
A = list(map(int, input().split()))
B = list(map(int, input().split()))

# Initialize counters
greater_than_all_B_count = 0
greater_than_all_A_count = 0

# Count the number of elements in A that are greater than all elements in B
for i in range(N):
    if all(A[i] > b for b in B if b != B[i]):
        greater_than_all_B_count += 1

# Count the number of elements in B that are greater than all elements in A
for i in range(N):
    if all(B[i] > a for a in A if a != A[i]):
        greater_than_all_A_count += 1

# Determine whether the requirement is satisfied based on the observed conditions
if greater_than_all_B_count > 1 or (greater_than_all_B_count == 1 and
greater_than_all_A_count == 0):
    print("No")
else:
    print("Yes")
```

Erroneously coupled UT for CoSPlay-14B

Input:
5
1 2 3 4 999999999
1 2 3 4 5

Generated expected output:
Yes

Buggy-code output:
Yes

Correct-reference output:
No

Coupling analysis:
This generated UT has a wrong oracle. The value `999999999` **in** A **is** larger than every value **in** B, whose maximum **is** only `5`. Therefore, no permutation of A can make $A_i \leq B_i$ hold **for all** positions, so the GT output **is** `No`. However, the buggy

code outputs `Yes`, matching the wrong oracle, **while** the correct reference code outputs `No`. Thus, this wrong UT rewards the wrong code **and** penalizes the correct code, forming a clean Code--UT coupling case.

New generated UT for CoSPlay-14B

Input:
5
1 2 3 4 999999999
1 2 3 4 5

Generated expected output:
No

Buggy-code output:
Yes

Explanation:
The correct answer **is** `No`. The largest value **in** A **is** `999999999`, **while** the largest value **in** B **is** `5`. Since `999999999` **is** greater than every element **in** B, this element cannot be placed at **any** position satisfying $A_i \leq B_i$. Therefore, the objective **is** impossible regardless of how the allowed swaps are used.

Effect:
With this corrected oracle, the buggy code would output `Yes` **and** fail the UT, **while** the correct reference code outputs `No` **and** passes it.

W.2.3 Analysis for case study of UT-code coupling

Figure 61 presents two concrete examples illustrating how spurious Code-UT coupling arises in practice and how Step 2 of our iterative self-play framework detects and breaks it. Figure 60 further visualizes the same process through the lens of the binary execution matrix, offering a complementary macro-level view of how the UT pool's discriminative structure changes before and after regeneration.

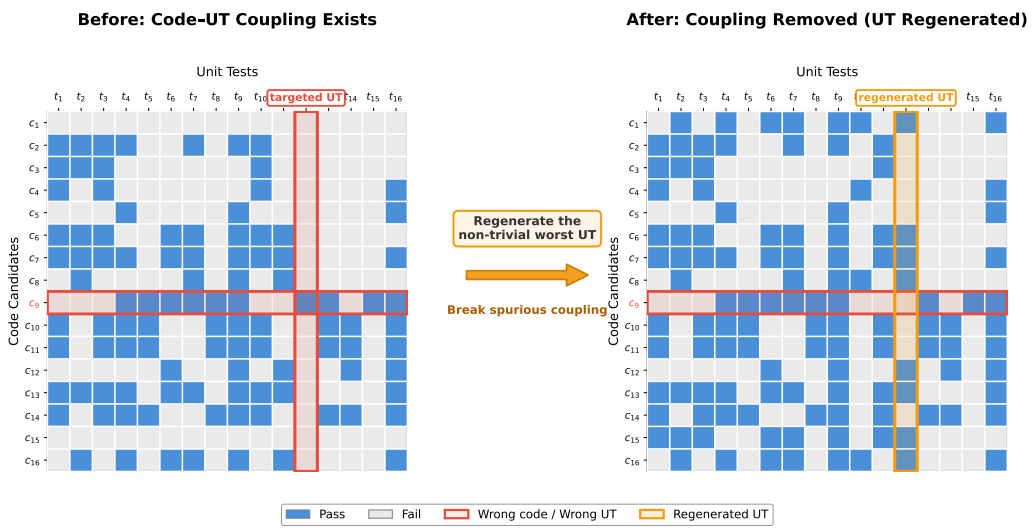
Execution Matrix Analysis (Figure 60). Each row of the matrix corresponds to a code candidate c_i and each column to a unit test t_j; a blue cell indicates a pass ($\mathbf{M}_{ij} = 1$) and a grey cell indicates a failure ($\mathbf{M}_{ij} = 0$). The highlighted red row marks the suspicious wrong or BoN-selected code, while the highlighted red column marks the suspicious UT before regeneration. The highlighted orange column marks the regenerated or resampled UT after Step 2.

In the 7B case, the before panel shows a clean Code-UT coupling pattern: the highlighted wrong code passes the highlighted low-pass UT, creating a false positive that can inflate the pass count of an wrong solution. After regeneration, the corresponding orange UT column no longer accepts the same wrong code, thereby removing this spurious agreement.

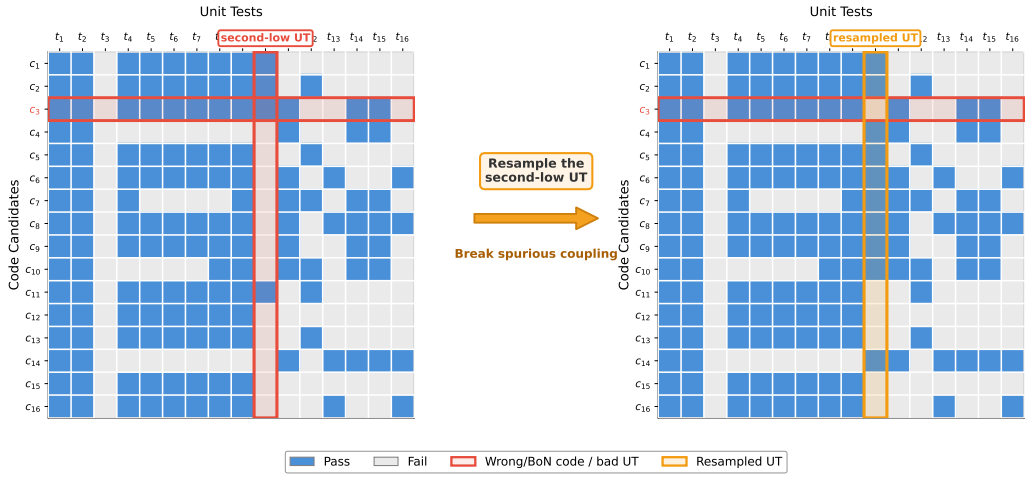
In the 14B case, the highlighted second-low UT also accepts the highlighted wrong/BoN-selected code before resampling, indicating that this UT can still provide misleading support for an wrong candidate. Unlike the 7B case, this column is not a unique-pass column; rather, it is a low-pass suspicious UT whose agreement with the highlighted wrong code contributes to the coupling phenomenon. After resampling, the highlighted wrong code fails the new UT, showing that the accidental agreement between the code and the previous UT has been broken.

Overall, the execution matrices show that Step 2 operates in a targeted manner: it replaces a suspicious low-but-nonzero-pass-rate UT column, while leaving the rest of the UT pool unchanged. This intervention removes the highlighted false positive and restores a more useful discriminative signal for subsequent refinement and selection.

Example 1: Rectangle Overlap Counting. The problem asks to count the maximum number of overlapping axis-aligned rectangles over any region. The shown candidate code is semantically



(a) CoSPlay-7B



(b) CoSPlay-14B

Figure 60: Execution matrices demonstrating the resolution of Code–UT coupling in CoSPlay. Each row represents a generated code candidate, and each column represents a generated unit test (UT). A blue cell indicates that the code passes the UT, while a grey cell indicates failure. In each subfigure, the left panel shows the execution matrix before Step 2, where the highlighted red row marks a suspicious wrong or BoN-selected code and the highlighted red column marks a suspicious low-but-nonzero-pass-rate UT. Their highlighted intersection corresponds to a false positive: the flawed code passes the suspicious UT, creating spurious Code–UT coupling. The right panel shows the matrix after replacing this suspicious UT with a regenerated or resampled UT, highlighted in orange. After replacement, the same highlighted flawed code no longer passes the new UT, indicating that the accidental agreement between the code and the previous UT has been removed. Subfigure (a) shows the CoSPlay-7B case, where regenerating the non-trivial worst UT breaks the coupling between a wrong code and a wrong UT. Subfigure (b) shows the CoSPlay-14B case, where resampling the bad second-low UT removes misleading support for the highlighted wrong/BoN-selected code. Overall, Step 2 performs a targeted intervention by replacing a suspicious UT column while keeping the remaining UT pool unchanged, thereby eliminating the highlighted false positive and improving the discriminative signal for subsequent repair and selection.

flawed because it reduces the two-dimensional overlap problem to a one-dimensional event-style count and fails to correctly model overlaps along both axes. Nevertheless, it passes the initial *False Pass* UT because the UT is spuriously aligned with the same error. Specifically, the expected answer of the initial UT is 3, while the true maximum overlap is 4. The wrong code also outputs 3, which

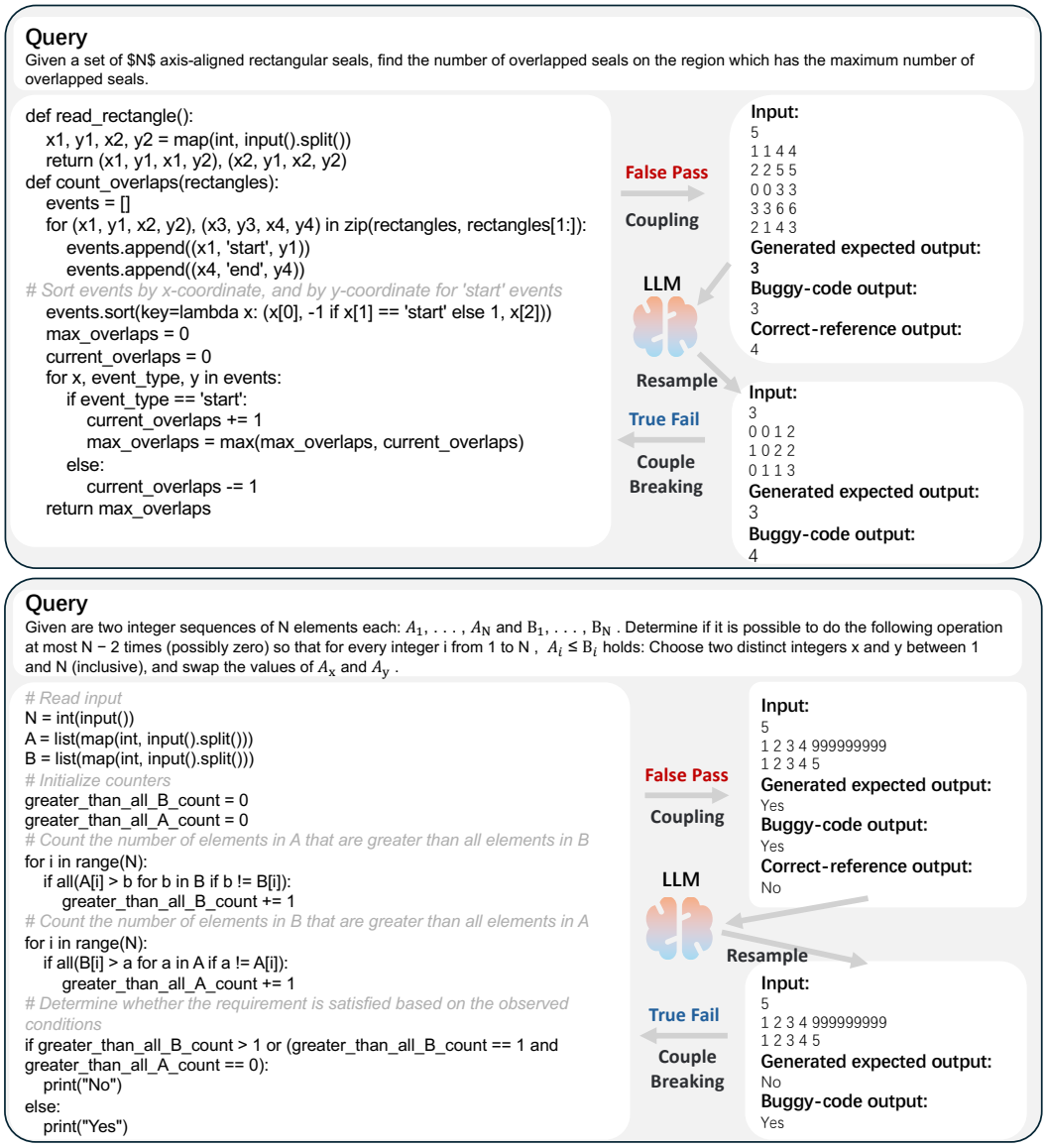


Figure 61: Case study of Code-UT coupling.

leads to a false positive. CoSPlay then resamples the suspicious UT using the spuriously passed code as a negative reference. The regenerated UT is still not oracle-correct, since its expected output remains imperfect. However, it successfully breaks the previous code and UT coupling because the same buggy implementation no longer agrees with the regenerated UT and is therefore filtered out. This example shows that the resampling step is not meant to guarantee an immediately perfect UT. Instead, it serves as a decoupling mechanism that disrupts accidental agreement between wrong code and wrong tests and recovers useful discriminative signal for selection.

Example 2: Swap-Based Array Dominance. The second example considers whether the sequence A can be permuted by swapping elements within A at most $N-2$ times so that $A_i \leq B_i$ holds for every position i. The generated *False Pass* UT is a valid input, but its oracle is incorrect. Specifically, for the input `A = [1,2,3,4,999999999]` and `B = [1,2,3,4,5]`, the generated expected output is `Yes`. However, the true answer is `No`, because the element `999999999` in A is larger than every element in B. Since the allowed operation only permutes elements of A and cannot change their values, this element must occupy some position i and will necessarily violate $A_i \leq B_i$. Thus, the instance is impossible even with unlimited swaps, and therefore also impossible under the stricter $N-2$ swap limit.

The buggy code does not directly check whether the multiset of A can be matched to B under the coordinate-wise inequality. Instead, it uses an indirect heuristic based on counting elements that are larger than almost all elements in the other sequence. On this UT, the heuristic incorrectly prints `Yes`, matching the wrong generated oracle. In contrast, the semantically correct reference code sorts both sequences and correctly detects that the largest element of A still exceeds the largest element of B, outputting `No`. As a result, this wrong UT creates a clean Code-UT coupling: the wrong code passes the wrong test, while the correct code fails it. This is precisely the type of false positive that can inflate the pass count of an incorrect solution and mislead final selection.

Summary. Taken together, Figure 61 and Figure 60 illustrate the Code-UT coupling problem at two complementary levels of granularity. The former exposes the concrete input-output misalignment that creates a false positive, while the latter reveals its global footprint in the execution matrix as a low-pass-rate UT column spuriously aligned with a wrong code. The rectangle-overlap example shows how targeted UT regeneration can disrupt an accidental agreement between a flawed implementation and an imperfect test. The swap-based array example further isolates an even cleaner failure mode: the generated UT has a valid input but an incorrect oracle, causing the wrong code to pass while a correct reference solution fails.

These cases demonstrate that self-generated UTs can become misleading not only because their inputs are insufficiently diverse, but also because their expected outputs may accidentally agree with a specific erroneous implementation. Step 2 addresses this issue by identifying suspicious low-but-nonzero-pass-rate UTs and regenerating them with the incorrectly passing code as a negative reference. This targeted regeneration discourages repeated agreement with the same flawed logic, breaks spurious Code-UT coupling, and helps recover useful discriminative signal for refinement and selection without relying on ground-truth supervision.

X PROMPTS USED IN OUR METHOD

Prompt for solution hints generation

```
<|im_start|>system
You are an expert Python programmer. You will be given a competitive programming
question. You will return several useful, non-obvious, and correct observations
about the problem, like hints to solve the problem. List them as numbered points.
 You will NOT return any code. Be creative and go beyond obvious insights.
<|im_end|>
<|im_start|>user
This is the problem:
```
{problem}
```
Output format (VERY IMPORTANT):
• Write one observation per line.
• Start each line with a number and a dot, like:
    1. first observation
    2. second observation
    3. third observation
• Do NOT add any extra text, headings, or explanations before or after the list.
Just output the numbered observations, nothing else.
<|im_end|>
<|im_start|>assistant
```

Prompt for specific solution idea generation

```
<|im_start|>system
You are an expert Python programmer.
<|im_end|>
```

```
<|im_start|>user
You will be given an competitive programming question (problem specification) and
 several correct observations about the problem.
This is the problem:
```
{problem}
```
Here are the first-order observations:
```
{first_order_observations}
```
You will brainstorm several new, useful, and correct observations about the
problem, derived from the given observations. You will NOT return any code. Be as
 creative as possible, going beyond what you think is intuitively correct.

Output format (VERY IMPORTANT):
• Write one derived observation per line.
• Start each line with a number and a dot, like:
    1. first derived observation
    2. second derived observation
    3. third derived observation
• Do NOT add any extra text, headings, or explanations before or after the list.
Just output the numbered observations, nothing else.
<|im_end|>
<|im_start|>assistant:
```

Prompt for code generation from solution idea

```
<|im_start|>system
You are a helpful assistant help user solve problems.
<|im_end|>
<|im_start|>user
You need to write Python script based on the observations.  You should use input
() to input and print() to output in your script. Your code should output the
results based on the input read in, rather than generating the given test example.

This is the problem:
```
{problem}
```
Here are intelligent observations:
```
{second_order_observations}
```
<|im_end|>
<|im_start|>assistant
```

Prompt for UT attack idea generation

```
<|im_start|>system
You are an expert Software Engineering Tester designing unit test to uncover the
potential bugs.
<|im_end|>
<|im_start|>user
You will be given a competitive programming question and an observation about the
 problem, which will be used to generate code to solve the problem, you should
```

```
brainstorm several attack ideas at the same idea level, which will unveil
potential design flaws, security risks, and edge case predictions.
This is the problem:
```
{problem}
```
Here is the intelligent observation:
```
{observation}
```
Output format (VERY IMPORTANT):
- Write one attack idea per line.
- Start each line with a number and a dot, like:
  1. first attack idea
  2. second attack idea
  3. third attack idea
- Do NOT add any extra text, headings, or explanations before or after the list.
Just output the numbered observations, nothing else.
<|im_end|>
<|im_start|>assistant
```

Prompt for UT input generation from UT attack idea

```
<|im_start|>system
You are an expert Software Engineering Tester designing unit test to uncover the
potential bugs.
<|im_end|>
<|im_start|>user
# Role
You are an expert Quality Assurance Engineer and Competitive Programmer. Your
task is to generate a strict "Unit Test Input" based on a provided "Attack Idea".

# Problem Description
```
{problem}
```

# Attack Idea
```
{attack_idea}
```


# Instructions (Chain of Thought)
To ensure the generated input is valid and effective, follow these steps strictly:


1. Analyze Constraints & Hidden Rules:
   - Identify all explicit constraints (e.g., $1 \le N \le 10^5$, sum of weights
 < $10^9$).
   - Crucial: Identify implicit constraints in the text (e.g., "distinct
integers", "connected graph", "permutation", "tree structure").
   - List these constraints clearly in your explanation.

2. Map Attack to Constraints:
   - How can you implement the `attack_idea` while strictly obeying ALL
constraints listed above?
   - If the attack requires a large number, ensure it fits within the variable
limits.
```

3. **Construct & Verify:**
   - Construct the test case **input** step-by-step.
   - **Self-Correction:** Check the generated **input** against the constraints one last time. Does the number of elements match N? Are the formats (spaces/newlines) correct?

4. **Final Output:**
   - Produce the explanation, **and finally** the raw **input** block.

After constructing the **input**, explicitly show how it can be decomposed back into the components described **in** the problem. If you cannot show this, discard the **input**.

Output Format
You must structure your response exactly **as** follows:

Explanation:
[Step 1: List constraints found **in** the problem...]
[Step 2: Reasoning on how to **apply** the attack idea within these constraints...]
[Step 3: Verification of the constructed **input**...]

Test Input:
[The raw **input** data ONLY. No comments. Strictly follow formatting.]
Let's think step by step.
<|im_end|>
<|im_start|>assistant

Prompt for UT output generation from specific input

<|im_start|>system
You are an expert Software Engineering Tester designing unit test to uncover the potential bugs.
<|im_end|>
<|im_start|>user
Role
You are an expert **in** Competitive Programming. Your specific task **is** to generate the correct output **for** a programming problem based on a unit test.

Problem
```
{problem}
```

Unit Test Input
```
{ut_input}
```

IMPORTANT INSTRUCTIONS
1. **Roleplay:** Act **as** a code execution engine.
2. **Chain of Thought:** You MUST write an "Explanation" section first. Trace the code logic step-by-step **with** the provided **input**.
3. **Format:** After the explanation, output the result inside a strict block.
4. Use only the given unit test **input**; **if** it seems mismatched to the problem **format**, do **not** invent missing data.
5. Match the EXAMPLE's output format exactly (spacing/line breaks/order); no brackets/commas unless shown; empty output -> blank line.

```
# Response Format
Explanation:
[Your step-by-step logic tracing here]

Test output:
[Raw Output Data ONLY]
Let's think step by step.
<|im_end|>
<|im_start|>assistant
```

Prompt for generating non-coupling UT

```
# Role
You are an expert in Competitive Programming. Your task is to regenerate the
correct output for a given unit test input.

# Problem
```
{problem}
```

# Unit Test Input
```
{ut_input}
```

# Previous Attempt (for reference, may contain format issues or logical errors,
please improve)
```
{previous_code}
```

```
{previous_ut}
```

IMPORTANT INSTRUCTIONS
1. Roleplay: Act as a code execution engine.
2. Chain of Thought: You MUST write an "Explanation" section first. Trace the
code logic step-by-step with the provided input.
3. Format: After the explanation, output the result inside a strict block.
4. Recompute strictly from the given input; if the input format seems off, do not
 invent missing data. Do NOT copy the previous attempt.
5. Match the EXAMPLE's output format exactly (spacing/line breaks/order); no
brackets/commas unless shown; empty output → blank line.

Response Format
Explanation:
[Your step-by-step logic tracing here]

Test output:
[Raw Output Data ONLY]

Let's think step by step.
```

Prompt for code fixing

```
<|im_start|>system
You are an expert programmer designing code for competitive programming question.
<|im_end|>
<|im_start|>user
# Role
You are an expert Python Debugger and Algorithm Engineer. Your task is to fix the
 provided code which failed specific unit tests.

# Problem Description
```
{problem}
```
# The Buggy Code
```python
{failed_code}
```
Failure Report
The code failed the following test case(s). Note that the "Actual Execution
Output" might contain a runtime error (Traceback) or simply an incorrect
calculation result.
```
{failed_uts_text}
```
Debugging Instructions (Chain of Thought)
To fix the code correctly, you MUST follow these steps:
1. Analyze the Failure Type:
   • Case A: Runtime Error (Traceback): If the output contains words like Error,
 Exception, or Traceback, identify strictly WHICH line caused the crash and WHY (
e.g., index out of bounds, division by zero, recursion limit).
   • Case B: Logical Error (Wrong Answer): If the code ran but produced
different output than expected, compare the Expected vs Actual. Trace the logic
to find where the calculation diverged.
2. Identify the Root Cause:
   • Don't just look at the input values. Ask yourself: "What logical flaw in
the algorithm allows this specific input to break the code?"
   • Is it an edge case (e.g., N=1, empty list)?
   • Is it a variable initialization issue?
3. Refine the Code:
   •
 Fix the logic to handle the general case, not just the specific failed input.
   • DO NOT simply add if input == ...: return ... (Hardcoding is strictly
forbidden).
   • Ensure all necessary libraries (e.g., sys, math) are imported.

Output Format
Analysis:
[Step 1 & 2: Explain the error type (Runtime/Logic) and the specific bug location
]
Refined Code:
```
[The fully fixed, runnable code]
```
Let's think step by step to find the bug.
<|im_end|>
<|im_start|>assistant
```

Prompt for random UT input generation

```
<|im_start|>system
You are an expert Software Engineering Tester designing unit test to uncover the
potential bugs.
<|im_end|>
<|im_start|>user
Your task is to generate EXACTLY {num_cases} NEW RANDOM and VALID test inputs for
 a competitive programming problem.
You should not repeat any example inputs in the problem.

Instructions (STRICT):
1. Each line represents ONE independent test input.
2. Each test input must strictly follow the input format and constraints of the
problem.
3. Each line MUST start with the exact prefix: CASE|
4. After CASE|, output ONLY the raw input values in correct order. Do NOT include
 parameter names, variable labels, code, or any explanatory text.

Problem:
```
{problem}
```
You MUST output in the following EXACT format:
CASE|```<input for test case>```
<|im_end|>
<|im_start|>assistant
```

Prompt for direct code generation

```
<|im_start|>system
You are a helpful assistant help user solve problems.
<|im_end|>
<|im_start|>user
You need to think first then write Python script. Use input() to input and print
() to output.
This is the problem:
```
{problem}
```
<|im_end|>
<|im_start|>assistant
```

Prompt for direct UT generation

```
<|im_start|>system
You are a helpful assistant help user generate test examples for coding tasks. <|
im_end|>
<|im_start|>user
Given a coding task, instead of providing the final script, your task is to
generate a new test example (both input, output and explanation).
This is the problem:
```
{problem}
```
You need to provide a new test example. A good test example should be completely
accurate and conform to the problem's format requirements, while also possessing
```

```
enough discriminative power to distinguish correct code from incorrect code.
Before providing a test example, you must think carefully and reason step by step
 to derive an input and output you are very confident are correct. For example,
start by designing an input you can reliably handle, then compute the output step
 by step. If you're unsure about the output, revise or re-design the input to
ensure accuracy. Finally, after completing these previous thinking and derivation
 steps, you MUST put your final test example in the following format:
Test Input:
[input here]
Test Output:
[output here]
Explanation:
[explanation here]
<|im_end|>
<|im_start|>assistant
```