

Code-TVT: Synthesizing Test-Case-Driven Verifiable Tasks for Code Reinforcement Learning

Jin Jiang¹, Junlong Zhang¹, Yuchen Yan³, Ze Zhao⁴
Changzhi Zhou⁴, Suncong Zheng^{4*}, Liangcai Gao^{1,2*}

¹Wangxuan Institute of Computer Technology, Peking University, Beijing, China

²State Key Laboratory of Multimedia Information Processing, Peking University, Beijing, China

³Zhejiang University

⁴Tencent Hunyuan, Beijing, China

jiangjin@stu.pku.edu.cn, gaoliangcai@pku.edu.cn

Abstract

Verifiers play a central role in large language model reasoning by providing reliable supervision for post-training methods such as reinforcement learning. In code generation, however, verification is still predominantly based on fixed test cases, offering a narrow but underexploited form of verifiable supervision. We explore how existing test cases can be reformulated into more diverse and challenging sources of verifiable training signals. We propose CODE-TVT (Test-Case-Driven Verifiable Tasks), a data synthesis framework that transforms program test cases into intrinsically verifiable code reasoning tasks. Using test cases already available in programming problems, we design two complementary tasks: *Edge Test-Case Generation*, which constructs edge-case inputs that expose subtle correctness failures, and *Input Reconstruction via Program Output Inversion*, which reasons backward from outputs to plausible inputs under execution-based verification. Experiments on three time-windowed LiveCodeBench splits (LCB-2408–2411, LCB-V5, and LCB-V6) show that CODE-TVT improves rejection-sampling fine-tuning, strengthens GRPO initialization, and yields additional Pass@8 gains when a small heldout CODE-TVT split is mixed into reinforcement learning.

1 Introduction

Recent progress in large language models shows that reliable verification is a key driver of reasoning-oriented post-training (OpenAI, 2024; Guo et al., 2025; Shao et al., 2024; Yu et al., 2025). Reinforcement learning with verifiable rewards (RLVR) provides strict feedback for complex problem solving (Guo et al., 2025; Yu et al., 2025), yet much of this progress has centered on mathematically formalized benchmarks, leaving code-

intensive settings such as competitive programming less explored. Code generation is a natural RLVR setting because program execution can automatically assess correctness, and prior code-RL work has shown how execution or unit-test feedback can be incorporated into training (Gehring et al., 2025; Liu et al., 2023a; Le et al., 2022).

In code generation, verification is typically realized by executing candidate programs against a fixed set of predefined test cases. Under this paradigm, a solution is considered correct if it passes all tests and incorrect otherwise, resulting in reward signals that are often binary or coarsely aggregated over test outcomes (Le et al., 2022). While this approach enables scalable and automated evaluation, it fundamentally constrains the difficulty and diversity of verifiable supervision. Fixed test suites provide limited coverage of edge cases, and coarse pass–fail rewards lead to sparse learning signals in reinforcement learning, making it difficult to distinguish partially correct solutions (Gehring et al., 2025; Liu et al., 2023a).

As a result, many efforts have focused on synthesizing additional test cases to improve coverage and strengthen verification signals (Ma et al., 2025a); others specifically aim to generate more challenging or adversarial test cases for competitive programming problems, with an emphasis on boundary conditions (Wang et al., 2025b; Cao et al., 2025). Moreover, although finer-grained rewards based on test pass rates can be constructed, prior work in reinforcement learning and verifiable training suggests that denser or improperly specified rewards may encourage opportunistic optimization rather than genuine improvements in overall correctness (Amodei et al., 2016).

Beyond their role as evaluators, test cases constitute an underutilized source of structured information for constructing new training problems. Each test case implicitly encodes assumptions about program behavior, boundary conditions, and

*Corresponding authors.

input–output relationships; however, this information is largely unused when test cases serve only as post hoc verification tools. Recent work further indicates that test cases admit additional exploitable structure, for example, by dynamically allocating or scaling test resources to improve reward quality and model selection under fixed computational budgets (Ma et al., 2025a). This perspective motivates a shift in how test cases are viewed: rather than treating them solely as static evaluators, we regard them as reusable building blocks for generating additional, more challenging, yet still strictly verifiable learning tasks.

Motivated by these observations, we introduce CODE-TVT, a test-case–driven data synthesis framework that reformulates existing program test cases into new code reasoning tasks with execution-verifiable rewards. Starting from standard code generation problems and their associated test suites, CODE-TVT constructs two complementary training tasks that require models to reason about program behavior under stricter or inverted conditions. The first task generates discriminative test inputs that expose behavioral differences between correct and incorrect implementations, thereby stressing boundary conditions. The second reconstructs plausible inputs from given program outputs, encouraging backward reasoning over program semantics. Because these rewards can be exploited through malformed inputs, executor failures, timeout artifacts, or trivial inverse mappings, we use gated verifiers that separate validity, execution status, normalization, and anti-trivial checks before assigning reward.

Our main contributions are summarized as follows:

1. We introduce CODE-TVT, a test-case–driven framework that turns existing program test cases into automatically verifiable code reasoning tasks.
2. We design two complementary RL tasks with verifiable rewards: edge test-case generation and input reconstruction via program output inversion, with gated checks intended to avoid reward-hacking shortcuts.
3. Experiments show that training with CODE-TVT improves generalization across multiple LiveCodeBench (Jain et al., 2025) versions (LCB-2408–2411, LCB-V5, and LCB-V6) and multiple model backbones.

2 CODE-TVT

We present the overall structure of CODE-TVT. After standard code-RL preliminaries (§2.1), we describe two complementary tasks derived from program test cases: edge test-case generation for code verification (§2.2), and input reconstruction from outputs obtained by executing reference programs on proposed valid inputs (§2.3). For each task, we give its formal definition, data synthesis, execution-verified reward, and gated checks intended to avoid reward-hacking shortcuts; Figure 1 summarizes the two DATA and REWARD lanes, with implementation details and telemetry in App. C.

2.1 Preliminaries: Code Reinforcement Learning

A coding problem x comes with a test suite $T_x = \{(a_i, b_i)\}_{i=1}^{|T_x|}$ and a reference program y^* . Executing a program y on input a_i either succeeds with output o_i or fails; let $c_i(y)$ denote whether execution succeeds and the normalized output matches b_i . The standard sparse reward $R_{\text{all-pass}}(x, y) = \prod_{i=1}^{|T_x|} c_i(y)$ therefore rewards a program only when it passes every test in T_x . Both CODE-TVT tasks reuse the same problem, tests, reference program, executor, and normalizer, and only modify the policy action and reward predicate. Full notation is in App. C.

2.2 Task I: Edge Test-Case Generation for Code Verification

2.2.1 Task Synthesis

Given a coding problem specified by a natural-language description $x \in \mathcal{X}$, with input domain $\mathcal{A}_x$ and output domain $\mathcal{B}_x$, we consider a set of candidate programs sampled from a policy model $\pi_\theta(y \mid x)$. Let $\{y^{(k)}\}_{k=1}^K \sim \pi_\theta(\cdot \mid x)$ denote K rollouts for problem x , and let $T_x = \{(a_i, b_i)\}_{i=1}^{|T_x|}$ be the associated test suite. Each sampled program $y^{(k)}$ is executed on all test cases in T_x , yielding the verdict vector

$$\mathbf{v}(y^{(k)}) = (c_i(y^{(k)}))_{i=1}^{|T_x|} \in \{0, 1\}^{|T_x|},$$

where $c_i(\cdot)$ is the per-test verdict from §2.1. Based on $\mathbf{v}(y^{(k)})$ we partition rollouts into: (i) *all-pass* programs, where all entries are 1; (ii) *all-fail* programs, where all entries are 0; and (iii) *partial-pass* programs, where both occur.

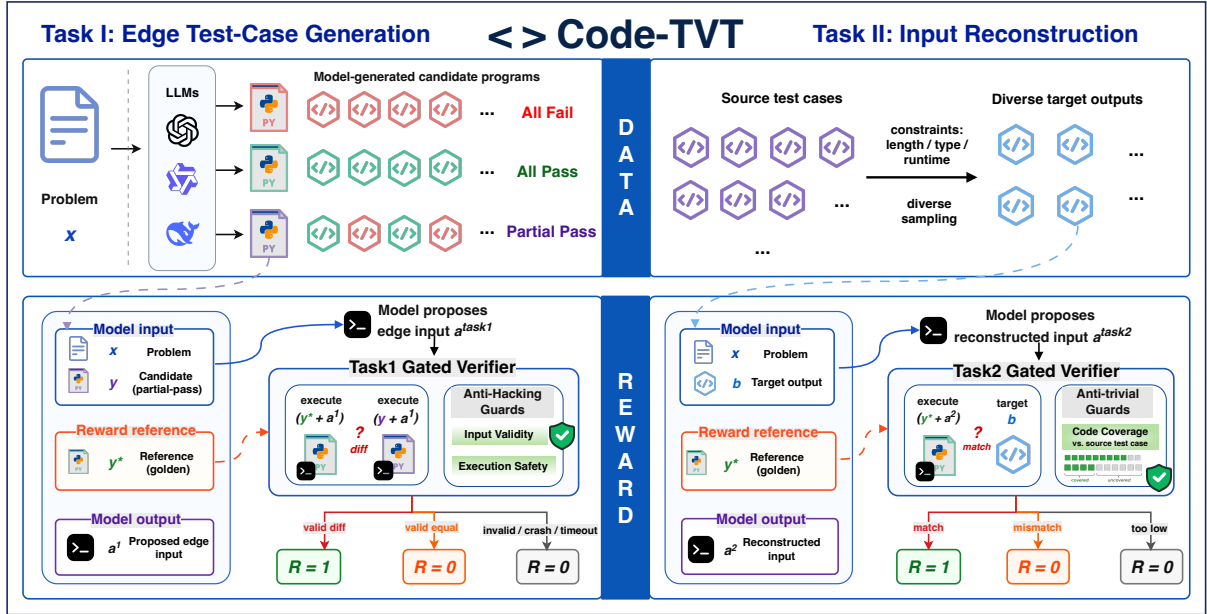


Figure 1: Overview of CODE-TVT, a test-case-driven framework for verifiable code reinforcement learning. The upper DATA lane synthesizes task instances from existing programming problems and test cases, while the lower REWARD lane assigns execution-gated rewards. **Left (Task I: Edge Test-Case Generation)**: partial-pass candidate programs are paired with reference programs, and the model proposes edge inputs that distinguish their behavior. **Right (Task II: Input Reconstruction)**: source-test-derived constraints guide proposals of valid, source-like inputs; reference execution yields diverse target outputs, and the model reconstructs inputs that reproduce them. Task I rewards only valid output differences; Task II rewards only matching, non-trivial reconstructions that pass the gated verifier. The two model-generated inputs are labeled a^1 (Task I) and a^2 (Task II) in the figure; both correspond to the symbol $\hat{a}$ used in the formulas of §2.2 and §2.3.

As illustrated in the left part of Figure 1, we collect rollouts from a diverse pool of large language models and execute each generated program against the test suite T_x . Programs are categorized according to their execution outcomes into all-pass, all-fail, and partial-pass cases. Task I is constructed exclusively from partial-pass programs, which indicate that the candidate solution is incorrect but not trivially so. For each such program, we form a new task instance consisting of the original problem specification x , a reference implementation $y^* \in \mathcal{Y}^*(x)$, and a candidate program y drawn from the partial-pass set. The learning objective is to generate a single *edge test-case input* $a \in \mathcal{A}_x$ that exposes behavioral differences between y and y^* .

2.2.2 Verified Reward System

Given a task instance (x, y, y^*) synthesized as above, the policy outputs a proposed edge input $\hat{a} \in \mathcal{A}_x$ (the Task I generated input in Figure 1). The reward is determined by a gated verifier rather than by a raw Boolean output comparison. Let $V_x(\hat{a}) \in \{0, 1\}$ denote the input validator for problem x , and abbreviate $(s_y, o_y) = \text{Exec}(y, \hat{a})$,

$(s_*, o_*) = \text{Exec}(y^*, \hat{a})$, $\bar{o}_y := \text{norm}_x(o_y)$, $\bar{o}_* := \text{norm}_x(o_*)$. We define the execution-validity gate

$$g_x(y, y^*, \hat{a}) := \mathbf{1}[V_x(\hat{a}) = 1 \wedge s_y = 1 \wedge s_* = 1].$$

We then compute a tri-state comparison:

$$\text{cmp}_x(y, y^*, \hat{a}) = \begin{cases} \text{invalid}, & g_x = 0, \\ \text{equal}, & g_x = 1 \wedge \bar{o}_y = \bar{o}_*, \\ \text{different}, & g_x = 1 \wedge \bar{o}_y \neq \bar{o}_*. \end{cases}$$

The edge-test reward is then

$$R_{\text{edge}}(x, y, y^*, \hat{a}) = \mathbf{1}[\text{cmp}_x(y, y^*, \hat{a}) = \text{different}].$$

Thus, execution failure is not collapsed into “outputs are different.” Malformed inputs, reference failures, candidate crashes, timeouts, and parsing errors are assigned the invalid state and receive zero reward. This reward depends solely on program execution, but only assigns credit to semantic behavioral disagreement under valid execution.

Operationally, the model is prompted to assess the candidate program y and propose a single edge test-case input. The input is executed by both y

and y^* and receives reward only when both executions are valid and their normalized outputs differ; matching outputs or gated failures receive zero reward. This keeps the reward sparse, executable, and aligned with the all-pass verification logic used in code RL.

Reward-Hacking Safeguards. For Task I, the verifier rejects malformed inputs, execution failures, timeouts, and parsing errors before comparing program outputs. A reward is assigned only when the generated input is valid, both the candidate and reference programs execute successfully, and their normalized outputs differ. By construction, invalid inputs and verifier failures are assigned zero reward rather than treated as successful edge cases. Appendix C.1 summarizes the Task I shortcut, the validity gate that blocks it, and the invalid-input audit motivating this design.

2.3 Task II: Input Reconstruction via Program Output Inversion

2.3.1 Task Synthesis

Given a coding problem specified by $x \in \mathcal{X}$, with input domain $\mathcal{A}_x$ and output domain $\mathcal{B}_x$, we assume access to a set of reference programs $\mathcal{Y}^*(x)$ and an associated test suite $T_x = \{(a_i, b_i)\}_{i=1}^{|T_x|}$. Task II uses source tests to infer sampling constraints rather than treating their outputs as fixed labels. We first infer admissible sampling constraints $\mathcal{C}_x$ from the problem and source tests, including input length, type structure, and runtime/resource limits, and then perform diverse sampling subject to $\mathcal{C}_x$ to obtain a pool $\tilde{\mathcal{A}}_x \subseteq \mathcal{A}_x$ of source-like inputs. In our implementation, $\tilde{\mathcal{A}}_x$ is produced by an LLM-driven proposer that conditions on x , T_x , and $\mathcal{C}_x$, followed by validator filtering ($V_x(\tilde{a}) = 1$) and de-duplication; see Appendix C for the reward-hacking safeguards. Executing a reference program on these sampled inputs produces a diverse target-output pool

$$\begin{aligned} \mathcal{B}_x^{\text{div}} = \{ \text{norm}_x(o_\star) \mid & \tilde{a} \in \tilde{\mathcal{A}}_x, V_x(\tilde{a}) = 1, \\ & (s_\star, o_\star) = \text{Exec}(y^*, \tilde{a}), \\ & s_\star = 1 \}. \end{aligned}$$

For Task II, a task instance is defined by a triple (x, y^*, b) , where $y^* \in \mathcal{Y}^*(x)$ is a reference implementation and $b \in \mathcal{B}_x^{\text{div}}$ is a target output. The objective is to predict a plausible reconstructed input $a \in \mathcal{A}_x$ such that executing y^* on a yields the target output b .

As shown in the right DATA lane of Figure 1, Task II infers constraints from source tests, proposes and validates source-like inputs, and executes a reference program to obtain diverse target outputs. For each target output b , we pair it with the problem specification x and a reference program y^* to form a new task instance. The model is then asked to reconstruct one plausible test input that could produce b when executed by y^* . Importantly, multiple valid inputs may correspond to the same output, making this task inherently non-unique.

2.3.2 Verified Reward System

Given a task instance (x, y^*, b) and a model-predicted reconstructed input $\hat{a} \in \mathcal{A}_x$ (the Task II generated input in Figure 1), the reward is computed by executing the reference program y^* on $\hat{a}$ and comparing the result with the target output. To avoid rewarding trivial or malformed reconstructions, the Task II gated verifier also requires input validity and an anti-trivial guard based on coverage relative to the source test cases. We use $G_x(\hat{a}; T_x) \in [0, 1]$ to denote the non-triviality score (instantiated by branch- or basic-block coverage when instrumentation is available and by structural complexity proxies such as input length or nesting depth otherwise), and γ_x to denote a per-problem threshold (set, by default, to a low percentile of $\{G_x(a_i; T_x)\}_{i=1}^{|T_x|}$ on the source tests). Both G_x and γ_x are detailed in Appendix C. Let $(s_\star, o_\star) = \text{Exec}(y^*, \hat{a})$. The gated reward is

$$\begin{aligned} R_{\text{inv}}(x, y^*, b, \hat{a}) = 1 [& V_x(\hat{a}) = 1 \wedge s_\star = 1 \\ & \wedge \text{norm}_x(o_\star) = \text{norm}_x(b) \\ & \wedge G_x(\hat{a}; T_x) \geq \gamma_x]. \end{aligned}$$

The reward is one only if the execution result matches the target output and the reconstructed input is valid and non-trivial under these gates; mismatches and too-low coverage receive zero reward. This verification relies solely on program execution and admits multiple valid inputs for the same task instance.

In practice, the model outputs a single candidate input for each task instance, which is executed by the reference program and compared against the target output. The input receives reward only when the normalized output matches b and the validity and anti-trivial gates pass. This admits multiple correct reconstructions while screening out degenerate inputs that would bypass the intended inverse-execution reasoning.

Reward-Hacking Safeguards. For Task II, output matching alone is insufficient for reward. The reconstructed input must be valid, execute successfully under the reference program, match the target output after normalization, and pass the anti-trivial coverage guard. By construction, malformed or overly simple inputs are assigned zero reward rather than treated as successful reconstructions. Appendix C.2 summarizes the Task II shortcut, the non-degenerate gate that blocks it, and the diagnostic audit motivating this design.

3 Experiments

3.1 Experimental Setup

Data and models. We use TACO (Li et al., 2023) as the primary training corpus, which serves both as the source for synthesizing the two CODE-TVT tasks and as the in-domain baseline. After standard filtering (test-case count ≥ 5 and reference-solution sandbox-execution checks), we retain 7,624 of the original 25,443 problems. Experiments span four base backbones: Qwen3-4B-Base, Qwen2.5-7B-Base, Qwen2.5-Coder-7B-Base, and Llama-3.1-8B-Base. Qwen3-4B-Base is the primary backbone used for all subsequent ablations. All reported training runs start from these base checkpoints. Full TACO description, full-mixture RFT composition (our internal 200K general SFT responses, 7,537 TACO-RFT responses, 61,133 Task I responses, and 72,430 Task II responses), and complete RFT/RL hyperparameters (LLaMA-Factory + r11m frameworks, learning rate, batch size, sequence length, hardware) are reported in App. B.2 and App. B.3.

Model checkpoints. Table rows denote stage checkpoints:

- **Stage I (RFT).** This stage yields four checkpoints per backbone: **Baseline-SFT** (general-data SFT only), + **CODE-TVT**, + **TACO**, and + **TACO & CODE-TVT**, where the latter three add the named RFT data to the general-data mixture.
- **Stage II (TACO-GRPO).** This stage yields two checkpoints per backbone: **RL from Baseline-SFT** and **RL from + CODE-TVT**, initialized from the corresponding Stage I checkpoints.

- **Stage III (mixed RL).** This stage keeps most synthesized CODE-TVT data in RFT and reserves a small hard CODE-TVT heldout split for joint RL with TACO.

Experimental Design. In production-scale post-training, the RL phase is typically constrained to a single high-value task such as code generation, because each RL objective requires substantial compute and task-specific verifier engineering. Consequently, auxiliary synthesized data is more commonly consumed in the preceding mid-training stage, typically through rejection-sampling fine-tuning (RFT). We evaluate CODE-TVT under three direct RFT/RL configurations, organized into two parts. Stage I and Stage II simulate the practical production pipeline: (**Stage I**, §3.2) synthesized CODE-TVT and/or TACO data are consumed in RFT, with no code-RL afterwards; (**Stage II**, §3.3) CODE-TVT is consumed in RFT, and TACO-only GRPO-RL is continued from the resulting RFT checkpoint. (**Stage III**, §3.4) separately studies the effect of using CODE-TVT during GRPO-RL. Stage I and Stage II share the unified main results table; Stage III is reported separately (App. B.1). For the formal relation to **CodeI/O** (Li et al., 2025), see Appendix D.

Evaluation. We evaluate primarily on three time-windowed LiveCodeBench splits (166 problems from LCB-2408–2411, 167 from LCB-V5, 175 from LCB-V6), and use CRUXEval (Gu et al., 2024) and a reasoning-focused BBH (Suzgun et al., 2023) subset as auxiliary probes in selected analyses. All inference uses vLLM v0.11.2 (Kwon et al., 2023) with $n=8$ samples per prompt at temperature 0.7, top- p 0.8, top- k 20, and a repetition penalty of 1.05. LCB and CRUXEval are scored via sandbox execution; full per-benchmark protocols are in App. B.4.

3.2 Stage I: Rejection Sampling Fine-Tuning

Stage I compares RFT-only variants with no subsequent code-RL (the four white rows per backbone block of Table 1: Baseline-SFT, + CODE-TVT, + TACO, and + TACO & CODE-TVT). Across the three Qwen-family backbones, the + CODE-TVT variants consistently improve over the general-instruction Baseline-SFT on Code Avg Pass@1, with gains of +2.2, +2.8, and +3.6 points for Qwen3-4B, Qwen2.5-7B, and Qwen2.5-Coder-7B. The Pass@8 gains are also positive and often larger, reaching +1.6, +2.9, and +5.1 points

Table 1: **Main Results: Pass@1 / Pass@8 on LiveCodeBench across Stage I/II checkpoints.** Columns report three LiveCodeBench (LCB) evaluation splits—LCB-08/11 (LCB-2408–2411), LCB-V5, LCB-V6—and Code Avg. White rows: Stage I RFT; blue rows: Stage II GRPO-RL from the indicated Stage I initialization. Row names are defined in [Model checkpoints](#). Within each backbone block, the best value in each metric column is in **bold**; ties are both bolded.

Model	LCB-08/11		LCB-V5		LCB-V6		Code Avg	
	Pass@1	Pass@8	Pass@1	Pass@8	Pass@1	Pass@8	Pass@1	Pass@8
Qwen3-4B								
Baseline-SFT	17.4	28.9	15.1	26.9	19.8	30.9	17.43	28.90
+ CODE-TVT	19.4	28.3	18.3	31.7	21.1	31.4	19.60	30.47
+ TACO	22.1	31.3	19.9	32.9	21.9	34.9	21.30	33.03
+ TACO & CODE-TVT	20.0	27.7	20.1	35.3	21.6	33.1	20.57	32.03
RL from Baseline-SFT	23.9	32.6	21.9	31.2	26.5	34.7	24.09	32.85
RL from + CODE-TVT	26.3	32.4	25.7	37.0	27.5	35.0	26.47	34.83
Qwen2.5-7B								
Baseline-SFT	14.2	24.7	12.4	24.6	14.4	25.1	13.67	24.80
+ CODE-TVT	15.7	27.1	14.9	26.9	18.9	29.1	16.50	27.70
+ TACO	17.6	24.7	19.6	31.7	19.4	30.9	18.87	29.10
+ TACO & CODE-TVT	19.0	26.5	17.7	32.5	19.1	33.7	18.60	30.90
RL from Baseline-SFT	19.2	27.4	22.7	32.3	24.0	31.6	21.97	30.43
RL from + CODE-TVT	21.2	28.1	24.2	36.1	25.8	33.3	23.73	32.50
Qwen2.5-Coder-7B								
Baseline-SFT	13.0	22.4	15.6	30.5	20.0	29.3	16.20	27.40
+ CODE-TVT	19.0	31.7	19.2	33.3	21.3	32.6	19.83	32.53
+ TACO	16.1	26.8	18.1	35.2	22.4	32.2	18.87	31.40
+ TACO & CODE-TVT	22.5	33.7	18.7	32.9	22.9	32.0	21.37	32.87
RL from Baseline-SFT	21.2	29.1	26.8	37.7	24.9	33.1	24.30	33.30
RL from + CODE-TVT	25.2	32.9	28.1	37.7	27.7	37.0	27.00	35.87

on the same backbones, indicating that CODE-TVT expands the model’s effective coverage of correct code more than its mode peak. Despite its smaller seed corpus, CODE-TVT is competitive with TACO on Code Avg, and TACO + CODE-TVT gives the best Stage I Code Avg Pass@8 on the two larger Qwen backbones. The same RFT improvement transfers to Llama-3.1-8B with the largest gain in our experiments, improving Code Avg by +5.6 Pass@1 points and +9.9 Pass@8 points; full numbers are in App. E.1.

3.3 Stage II: Continued GRPO Reinforcement Learning

Stage II runs GRPO-RL on TACO only on top of the Stage I checkpoints (the two blue rows per backbone block in Table 1). Figure 2 contrasts two initializations: continuing from Baseline-SFT versus continuing from the + CODE-TVT RFT checkpoint. Across the three Qwen-family back-

bones, the CODE-TVT-initialized RL run generally starts higher and ends higher, with one tied Pass@8 endpoint on Qwen2.5-Coder-7B LCB-V5. On LCB-V6, Stage II further improves over the corresponding + CODE-TVT RFT checkpoint by **+3.6**, **+4.2**, and **+4.4** points across the three Qwen backbones, suggesting that the verification-aware RFT signal remains useful during code-RL.

The strongest absolute endpoint is Qwen2.5-Coder-7B, where the CODE-TVT-initialized GRPO-RL reaches **37.0** Pass@8 on LCB-V6 and **35.87** Code Avg, the best Stage II result in Table 1. We also ran the same Stage I/II pipeline on Llama-3.1-8B; it improves the two evaluated older LCB splits but is unstable on LCB-V6, so we report it separately as a cross-backbone replication rather than include it in the main Qwen-family comparison (App. E.1).

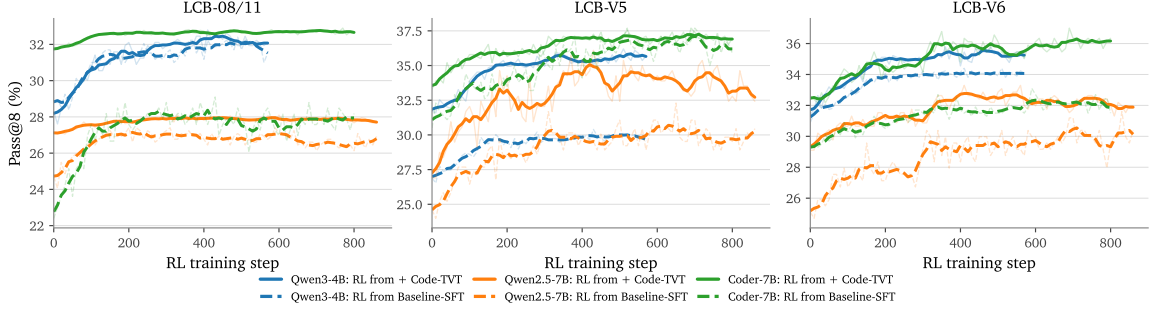


Figure 2: **Stage II GRPO-RL training curves on three backbones (Pass@8)**. Solid: RL continued from the + CODE-TVT RFT checkpoint; dashed: RL continued from Baseline-SFT. Curves are truncated to the shorter effective validation horizon within each backbone pair.

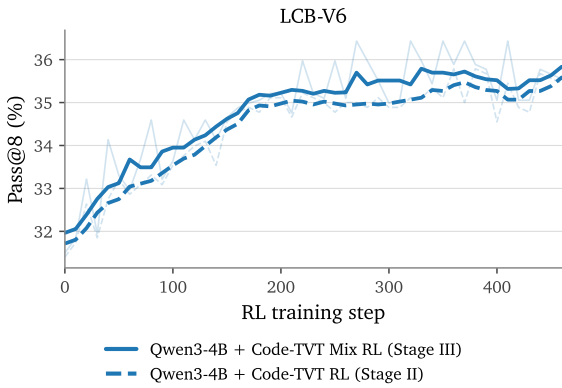


Figure 3: **Stage III mixed-RL on Qwen3-4B LCB-V6 (Pass@8)**. Solid: TACO mixed with the heldout CODE-TVT split; dashed: TACO-only RL from the same + CODE-TVT RFT checkpoint. (App. E.2)

3.4 Stage III: Mixed Reinforcement Learning

Stage III is reported on Qwen3-4B and tests whether a small hard CODE-TVT heldout split can be mixed into the RL phase rather than used only in RFT. Most synthesized CODE-TVT prompts remain in Stage I RFT; during GRPO-RL, the mixed run adds 1,000 hard Task I prompts and 1,000 hard Task II prompts to the TACO RL train file. The comparison is against the Stage II TACO-only RL baseline from the same + CODE-TVT RFT checkpoint, with the LCB-V6 Pass@8 curve shown in Figure 3 and the full setup in App. E.2.

Across the three LCB splits, mixed RL generally attains higher Pass@8 than the TACO-only RL baseline, as shown by the full training trajectories in Figure 5. The same comparison is less consistent for Pass@1. We therefore view Stage III as exploratory evidence of a sample-diversity trade-off: incorporating a small heldout CODE-TVT split during RL may improve multi-sample coverage, while TACO-only RL can remain stronger in

Table 2: **Task Component Ablation on Qwen3-4B**. Pass@1 / Pass@8 on LiveCodeBench.

Model	LCB-08/11		LCB-V5		LCB-V6	
	P@1	P@8	P@1	P@8	P@1	P@8
Baseline-SFT	17.4	28.9	15.1	26.9	19.8	30.9
+ CODE-TVT-Task I	20.0	32.5	17.2	34.1	20.1	32.0
+ CODE-TVT-Task II	20.4	30.1	20.0	35.3	21.0	33.1
+ CODE-TVT (Full)	19.4	28.3	18.3	31.7	21.1	31.4

single-sample correctness.

4 Analysis

4.1 Ablation on Task Components

Table 2 isolates the two task components of CODE-TVT against the same general-data Baseline-SFT reference. Both Task I and Task II outperform Baseline-SFT on every LCB split; Task II gives the strongest single-component Pass@1 gains, while Task I has the highest LCB-08/11 Pass@8 and Task II has the highest V5/V6 Pass@8. The full mixture is not uniformly best, but it attains the best LCB-V6 Pass@1; together with the different Pass@8 winners, this suggests that edge-case generation and output-to-input inversion provide complementary but non-identical supervision. Extended task-component ablations are reported in Appendix F.1.

4.2 Auxiliary Reasoning Diagnostics

As an auxiliary analysis, we inspect whether the behavior associated with CODE-TVT is also visible on code-execution and related reasoning diagnostics outside LiveCodeBench. We report CRUXEval together with three illustrative BBH (Suzgun et al., 2023) subtasks whose formats emphasize temporal sequences, object-state tracking, and structured spatial constraints. These BBH

Table 3: **Auxiliary Reasoning Diagnostics on Qwen3-4B.** Pass@1 on CRUXEval and three illustrative BBH subtasks.

Model	CRUX	TempSeq	ObjCount	GeoShape
Baseline-SFT	47.2	65.3	53.1	43.2
+ TACO	51.7	64.5	59.1	44.9
+ CODE-TVT	55.5	71.0	60.7	51.0
+ TACO & CODE-TVT	55.6	67.8	56.9	45.8

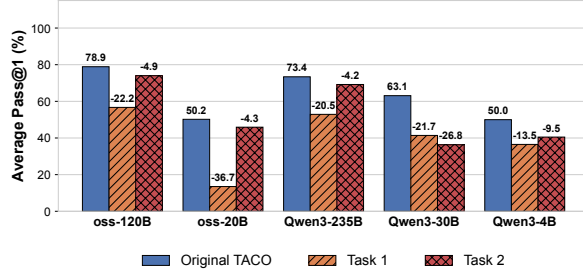


Figure 4: **Average Pass@1 on TACO and CODE-TVT tasks.** The y-axis reports Average Pass@1 (%), with each model group containing bars for Original TACO, Task I, and Task II. Labels above the Original TACO bars are absolute scores; labels above the Task I and Task II bars are changes relative to Original TACO.

columns serve as compact, mechanism-oriented probes rather than an aggregate BBH evaluation or evidence of broad general-reasoning gains. Table 3 reports Pass@1 on Qwen3-4B for the four §3.2 variants. On CRUXEval, the +Code-TVT score is **8.3** pt above Baseline-SFT (47.2 → 55.5), compared with **4.5** pt for +TACO (47.2 → 51.7); the combined +TACO & Code-TVT recipe records **55.6**. For the three reported BBH probes, the +Code-TVT values are **5.7**, **7.6**, and **7.8** pt above Baseline-SFT on TempSeq, ObjCount, and GeoShape, respectively. The complete BBH results, which include both higher and lower subtask scores, are provided in Appendix F.2.

4.3 Difficulty Stratification

The synthesized tasks are substantially harder than the original TACO data. Figure 4 compares five representative models on TACO, Task I, and Task II: while all models are at or above 50% average Pass@1 on TACO, both CODE-TVT tasks are consistently lower, with Task I dropping by 13.5–36.7 points across models and Task II showing a 26.8-point gap on Qwen3-30B-A3B-Instruct-2507. This gap persists under Pass@2 and across difficulty, skill, and input-length breakdowns, indicating that CODE-TVT stresses verification and inverse I/O reasoning beyond ordinary code generation; full stratified analyses further examine

Pass@k scaling, original TACO difficulty, algorithmic skill categories, and input-length sensitivity in Appendix F.3.

4.4 Case Study

Even when models solve the original code-generation problem, they can still fail on the corresponding CODE-TVT task. We show two representative traces: in Task I, the model misses a hollow-constellation counterexample and incorrectly accepts a flawed star-map solution (Figure 10, Appendix G.1); in Task II, it recognizes the scheduling problem and its greedy structure but fails to invert the implementation-specific sorting order, producing an input whose execution yields the wrong output (Figure 11, Appendix G.2). These cases support our core mechanism claim: CODE-TVT is not merely another form of forward code generation, but targets complementary skills—semantic verification and edge-case construction in Task I, and output-to-input inversion in Task II—which explains why it can provide useful training signal beyond standard code synthesis.

5 Conclusion and Future Work

We have shown that existing program test cases can serve a broader role than final evaluation: they can be transformed into executable training tasks that reward semantic verification and inverse program reasoning. The main lesson is that test-case-derived tasks provide supervision complementary to ordinary code-generation data, improving rejection-sampling fine-tuning and continued code-RL while exposing where the placement of auxiliary tasks in the training pipeline matters. The gated verifiers have been central to this framing, because they turn invalid inputs, execution artifacts, and trivial reconstructions into explicit non-rewarded outcomes rather than hidden positives.

Future work should test whether this recipe extends from competitive-programming problems to richer software environments. We plan to broaden CODE-TVT beyond a single language and strengthen test-case generation for harder, more diverse, and more diagnostic rewards. The two task designs may also transfer to repository-level code-agent settings such as SWE-Bench (Jimenez et al., 2024): Task I can generate issue-revealing tests, while Task II can simulate repository code trajectories to improve repository understanding.

Limitations

CODE-TVT is designed for settings where executable programming tasks, reference solutions, and test suites are available. We filter low-quality seeds and use gated execution-based rewards with input validation, sandboxed execution checks, output normalization, anti-triviality guards, and diagnostic audits (Appendix C). These safeguards are intended to make common reward-hacking channels explicit and non-rewardable, and the framework naturally benefits from stronger benchmark curation and test generation.

Our experiments focus on competitive-programming-style tasks derived from TACO and primarily evaluate 4B–7B Qwen-family models, with auxiliary analyses on Llama and larger models. Extending CODE-TVT to repository-level, interactive, stateful, or expensive-execution settings would require richer task construction and environment modeling, which we leave for future work.

Ethical Considerations

This work uses publicly released programming datasets and model checkpoints, and does not involve human-subject experiments, private user data, or personally identifying information. The synthesized tasks are intended for research on executable code reasoning and post-training. Because stronger code-generation models can be misused to produce insecure or unwanted software, evaluations and artifact releases should preserve sandboxed execution, dataset-license checks, and provenance documentation. We report verifier safeguards and diagnostic audits in Appendix C to make invalid inputs, execution artifacts, and trivial reconstructions explicit rather than hidden in the reward signal.

Acknowledgments

This work is supported by the projects of Beijing Nova Interdisciplinary Program (20240484647) and National Natural Science Foundation of China (No. 62376012), which is also a research achievement of State Key Laboratory of Multimedia Information Processing and Key Laboratory of Science, Technology and Standard in Press Industry (Key Laboratory of Intelligent Press Media Technology).

References

- Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. [A3Test: Assertion-augmented automated test case generation](#). *Information and Software Technology*, 176:107565.
- Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. [Concrete problems in AI safety](#). *arXiv preprint arXiv:1606.06565*.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *arXiv preprint arXiv:2108.07732*.
- Yuhan Cao, Zian Chen, Kun Quan, Ziliang Zhang, Yu Wang, Xiaoning Dong, Yeqi Feng, Guanzhong He, Jingcheng Huang, Jianhao Li, Yixuan Tan, Jiafu Tang, Yilin Tang, Junlei Wu, Qianyu Xiao, Can Zheng, Shouchen Zhou, Yuxiang Zhu, Yiming Huang, and Tianxing He. 2025. [Can LLMs generate reliable test case generators? a study on competition-level programming problems](#). *arXiv preprint arXiv:2506.06821*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *arXiv preprint arXiv:2107.03374*.
- Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. [ChatUniTest: A framework for LLM-based test generation](#). In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576. ACM.
- Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. [StepCoder: Improving code generation with reinforcement learning from compiler feedback](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand. Association for Computational Linguistics.
- Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. 2016. [Automated test generation using model checking: an industrial evaluation](#). *International Journal on Software Tools for Technology Transfer*, 18(3):335–353.
- Gordon Fraser and Andrea Arcuri. 2011. [EvoSuite: Automatic test suite generation for object-oriented software](#). In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference*

- on *Foundations of software engineering*, pages 416–419. ACM.
- Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. 2025. [Klear-CodeTest: Scalable test case generation for code reinforcement learning](#). *arXiv preprint arXiv:2508.05710*.
- Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. [RLEF: Grounding code LLMs in execution feedback with reinforcement learning](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 19034–19055. PMLR.
- Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. [CRUXEval: A benchmark for code reasoning, understanding and execution](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. [DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning](#). *Nature*, 645(8081):633–638.
- Zhongmou He, Yee Man Choi, Kexun Zhang, Ivan Bercovich, Jiabao Ji, Junting Zhou, Dejie Xu, Aidan Zhang, Yixiao Zeng, and Lei Li. 2026. [HARDTEST-GEN: A high-quality RL verifier generation pipeline for LLM algorithmic coding](#). In *International Conference on Learning Representations*, volume 2026, pages 35671–35715.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with APPS](#). In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Live-CodeBench: Holistic and contamination free evaluation of large language models for code](#). In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.
- Jin Jiang, Yuchen Yan, Yang Liu, Jianing Wang, Shuai Peng, Xunliang Cai, Yixin Cao, Mengdi Zhang, and Liangcai Gao. 2025. [LogicPro: Improving complex logical reasoning via program-guided learning](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26200–26218, Vienna, Austria. Association for Computational Linguistics.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. [SWE-bench: Can language models resolve real-world GitHub issues?](#) In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient memory management for large language model serving with PagedAttention](#). In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, pages 611–626. ACM.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. [CodeRL: Mastering code generation through pre-trained models and deep reinforcement learning](#). In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.
- Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. [Code/O: Condensing reasoning patterns via code input-output prediction](#). *arXiv preprint arXiv:2502.07316*.
- Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. [TACO: Topics in algorithmic code generation dataset](#). *arXiv preprint arXiv:2312.14852*.
- Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. [RLTF: Reinforcement learning from unit test feedback](#). *Transactions on Machine Learning Research*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. [Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.
- Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. [DeepCoder: A fully open-source 14b coder at O3-mini level](#). <https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51>. Notion Blog.
- Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Si-jia Luo, and Jie Tang. 2025a. [Dynamic scaling of unit tests for code reward modeling](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6917–6935, Vienna, Austria. Association for Computational Linguistics.

- Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2025b. [Rethinking verification for LLM code generation: From generation to testing](#). In *Advances in Neural Information Processing Systems*, volume 38, pages 137549–137582. Curran Associates, Inc.
- OpenAI. 2024. [OpenAI o1 system card](#). *arXiv preprint arXiv:2412.16720*.
- Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. [An empirical evaluation of using large language models for automated unit test generation](#). *IEEE Transactions on Software Engineering*, 50(1):85–105.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. [DeepSeekMath: Pushing the limits of mathematical reasoning in open language models](#). *arXiv preprint arXiv:2402.03300*.
- Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. [Execution-based code generation using deep reinforcement learning](#). *Transactions on Machine Learning Research*.
- Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and Jason Wei. 2023. [Challenging BIG-bench tasks and whether chain-of-thought can solve them](#). In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051, Toronto, Canada. Association for Computational Linguistics.
- Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025a. [Co-evolving LLM coder and unit tester via reinforcement learning](#). *arXiv preprint arXiv:2506.03136*.
- Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025b. [CodeContests+: High-quality test case generation for competitive programming](#). In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.
- Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. 2025. [CodePlan: Unlocking reasoning potential in large language models by scaling code-form planning](#). In *International Conference on Learning Representations*, volume 2025, pages 78641–78663.
- Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. 2024. [Grounding data science code generation with input-output specifications](#). *arXiv preprint arXiv:2402.08073*.
- Man Fai Wong and Chee Wei Tan. 2024. [Aligning crowd-sourced human feedback for code generation with bayesian inference](#). In *2024 IEEE Conference on Artificial Intelligence (CAI)*, pages 158–163. IEEE.
- Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. [On the evaluation of large language models in unit test generation](#). In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1607–1619. IEEE Computer Society.
- Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025. [DAPO: An open-source LLM reinforcement learning system at scale](#). In *Advances in Neural Information Processing Systems*, volume 38, pages 113222–113244. Curran Associates, Inc.
- Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. [No more manual tests? evaluating and improving ChatGPT for unit test generation](#). *arXiv preprint arXiv:2305.04207*.
- Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyang Luo. 2024. [LlamaFactory: Unified efficient fine-tuning of 100+ language models](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.
- Changzhi Zhou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, Ziyang Xu, Bohui Zhai, Hengyi Liu, Speed Zhu, Wiggan Zhou, and Fengzong Lian. 2026a. [AutoCodeBench: Large language models are automatic code benchmark generators](#). In *International Conference on Learning Representations*, volume 2026, pages 79116–79148.
- Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, Tianfu Fu, Beichen Li, Dongruixuan Li, Wenhao Chai, Zhuang Liu, Aleksandra Korolova, Peter Henderson, Natasha Jaques, Pramod Viswanath, and 2 others. 2026b. [AutoCode: LLMs as problem setters for competitive programming](#). In *International Conference on Learning Representations*, volume 2026, pages 16506–16540.

Appendix Overview

Appendix A. Related Work	p. 12
A.1 Code Generation with Reinforcement Learning	p. 12
A.2 Test Case Generation	p. 12
A.3 Program I/O Supervision for Reasoning	p. 12
Appendix B. Implementation Details and Reproducibility	p. 13
B.1 Three-Stage Experimental Design	p. 13
B.2 TACO Corpus and Data Synthesis Pipeline	p. 14
B.3 Training Hyperparameters	p. 14
B.4 Evaluation Protocols and Terminology	p. 14
Appendix C. Reward-Hacking Safeguards	p. 14
C.1 Task I: Edge Test-Case Generation	p. 15
C.2 Task II: Input Reconstruction	p. 15
C.3 Training-Time Telemetry and Scope	p. 15
Appendix D. Comparison with CodeI/O Input Prediction	p. 16
Appendix E. Additional Main Experimental Results	p. 16
E.1 Llama-3.1-8B: Fine-Tuning and RL Results	p. 17
E.2 Mixed Reinforcement Learning Curves	p. 17
Appendix F. Additional Analysis Results	p. 17
F.1 Full Task Component Ablation	p. 17
F.2 Auxiliary BBH Subtask Diagnostic	p. 18
F.3 Difficulty Stratification: Pass@k, Original Difficulty, Skills, and Input Length	p. 18
Appendix G. Qualitative Examples	p. 19
G.1 Task I Examples	p. 20
G.2 Task II Examples	p. 21

A Related Work

This section positions CODE-TVT relative to three adjacent literatures. We first review code generation with reinforcement learning, then test-case generation as executable supervision, and finally program I/O supervision for reasoning; this order separates our reward-environment contribution from prior dataset and benchmark construction.

A.1 Code Generation with Reinforcement Learning

Reinforcement learning has been extensively used to improve code generation by optimizing models against executable signals such as compiler feedback, program execution, and unit test outcomes (Dou et al., 2024; Shojaee et al., 2023; Wong and Tan, 2024). These studies demonstrate that code generation naturally supports objective and verifiable rewards beyond supervised learning. Recent advances in large-scale language model reinforcement learning emphasize relative and group-based optimization. Group Relative Policy Optimization (GRPO) shows strong scalability and test-

time gains in reasoning tasks when trained with execution-verifiable rewards (Shao et al., 2024; Guo et al., 2025). Parallel work explores joint optimization of code generation and auxiliary components, such as co-evolving code and unit test generators without ground-truth supervision (Wang et al., 2025a). In parallel, substantial effort has focused on constructing executable datasets and benchmarks for verifiable training and evaluation (Li et al., 2023; Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Liu et al., 2023b; Jain et al., 2025), as well as automatically synthesizing programming problems and benchmarks using language models, particularly for competitive programming (Zhou et al., 2026a,b).

A.2 Test Case Generation

Test cases are central to both evaluation and reinforcement learning for code generation. Classical automated test generation relies on evolutionary search, model checking, and assertion-guided synthesis (Fraser and Arcuri, 2011; Enoiu et al., 2016; Alagarsamy et al., 2024), providing systematic coverage but limited adaptability to diverse algorithmic domains. More recent work investigates LLM-based test and unit test generation, often using generate–execute–repair pipelines to improve test quality (Yuan et al., 2023; Chen et al., 2024; Schäfer et al., 2024; Yang et al., 2024). Studies on competitive programming benchmarks show that existing test sets are frequently insufficiently adversarial, leading to unreliable evaluation and biased reinforcement learning rewards (He et al., 2026; Wang et al., 2025b). Recent analyses further reveal that sparse or homogeneous test suites create persistent blind spots for execution-based verifiers (Ma et al., 2025b), motivating human–LLM collaborative test generation frameworks and scalable generate–verify pipelines tailored for code reinforcement learning (Ma et al., 2025b; Fu et al., 2025). These works motivate stronger executable supervision, while the following subsection discusses the separate line of work that uses program I/O behavior as reasoning data.

A.3 Program I/O Supervision for Reasoning

Several recent studies use executable programs and their input–output behavior as supervision for reasoning. CRUXEval formalizes input and output prediction over short Python functions as a benchmark for code execution reasoning (Gu et al., 2024). CodePlan scales code-form planning for

general reasoning (Wen et al., 2025), while LogicPro synthesizes text-form logical reasoning data from algorithmic problems, test cases, reference programs, and intermediate variable traces (Jiang et al., 2025). GIFT4Code uses I/O specifications and execution-derived feedback to ground data-science code generation (Wen et al., 2024).

CodeI/O and CODE-TVT. The closest work to our Task II is CodeI/O, which transforms raw code into input/output prediction tasks, trains models with natural-language chain-of-thought rationales, and verifies predictions by output matching or re-execution; CodeI/O++ further revises rationales through multi-turn correction (Li et al., 2025). CODE-TVT shares the broad motivation of learning from program I/O behavior, but the objective is different: CodeI/O scales supervised I/O-prediction and self-correction data for mid-training or continued SFT, whereas CODE-TVT starts from verified code-generation problems and turns their test cases into reward-bearing post-training tasks for rejection sampling and reinforcement learning. Thus, the distinction is **not merely SFT versus RL as optimization algorithms**: in CODE-TVT, sampled policy actions are judged by executable rewards aligned with sampling and selection at test time. This also changes the verifier’s role. CodeI/O uses verification as an offline correctness filter, while CODE-TVT uses the verifier as the reward function, so malformed inputs, execution artifacts, and trivial inverse solutions become reward-hacking risks that must be gated out. For Task II, CODE-TVT also rewards any valid, non-trivial reconstructed input whose execution under the reference program reproduces the target output, reflecting a one-to-many inverse relation rather than recovery of a single labeled input or rationale. Appendix D gives the formal comparison with CodeI/O’s input-prediction formulation.

B Implementation Details and Reproducibility

This section documents the experimental details needed to interpret every reported result. It first fixes the three-stage design, then the data construction recipe, training setup, and evaluation terminology, so that the tables and curves can be mapped back to concrete protocols.

B.1 Three-Stage Experimental Design

We organize all CODE-TVT experiments into three stages that span the practical trade-off between the rejection-sampling fine-tuning (RFT, sometimes called mid-training) phase and the code-RL phase. The three stages reflect three concrete configurations that frequently arise in industrial post-training pipelines:

Stage I: All synthesized data in fine-tuning (§3.2). All synthesized CODE-TVT data and the in-domain TACO data are consumed during the RFT phase, and no code-RL is performed afterwards. The RFT mixture combines (i) our internal 200K general SFT responses, (ii) 7,537 TACO-RFT responses, and (iii) 61,133 Task I and 72,430 Task II rejection-sampled responses. Variants ablate which subset of (ii) and (iii) is included: Baseline-SFT (general only), + CODE-TVT, + TACO, and + TACO & CODE-TVT. This stage validates whether CODE-TVT is useful as a *mid-training* data source on its own, in the regime where the downstream RL phase is reserved for unrelated tasks (or absent altogether).

Stage II: CODE-TVT in fine-tuning, TACO-only in code-RL (§3.3). Stage II runs GRPO-RL on TACO only on top of Stage I RFT checkpoints. The two RL configurations differ only in the RFT initialization: one starts from Baseline-SFT, while the other starts from the + CODE-TVT RFT checkpoint. This stage corresponds to the most common industrial recipe: the code-RL phase is reserved for the single primary task (code generation), and auxiliary synthesized data is absorbed during the preceding RFT phase. Because Stage II shares the same backbone $\times$ benchmark grid as Stage I and uses the same evaluation protocol, we report Stage I and Stage II together in the unified main table (Table 1).

Stage III: mixed code-RL (§3.4). Stage III follows the main-text setup: most synthesized CODE-TVT data remains in RFT, while a small hard CODE-TVT heldout split is mixed with TACO during code-RL. Concretely, the mixed-RL train files contain 7,620 TACO prompts, 1,000 Task I heldout prompts, and 1,000 Task II heldout prompts (9,620 prompts total). The two heldout manifests are selected with seed 2026 from prompts with exactly eight rollout attempts and partial-pass rates from 1/8 to 7/8; their 2,000 unique question hashes are excluded from the RFT

pool.

B.2 TACO Corpus and Data Synthesis Pipeline

TACO corpus. TACO (Li et al., 2023) is a large-scale, fine-grained code generation dataset collected from diverse competitive-programming sources (e.g., CodeChef, Codeforces, GeeksforGeeks), annotated with rich metadata such as problem topics, algorithm types, required programming skills, and difficulty levels. Crucially, each problem contains multiple verified reference implementations, which are essential for constructing the execution-based reward signals in our data synthesis pipeline. The original TACO dataset contains 25,443 problems. Following Luo et al. (2025), we adopt TACO-Verified¹ as the initial training set and further apply additional filtering: we remove problems with fewer than five test cases and problems for which the first three reference (gold) solutions fail under our sandbox execution. After filtering, we obtain 7,624 problems; the corresponding TACO-RFT component contains 7,537 response rows after conversion.

Task I and Task II synthesis. The two CODE-TVT task constructions follow the procedures in §2.2 and §2.3. After sampling and reward-based response selection, we retain 61,133 Task I and 72,430 Task II responses for the full RFT mixture.

B.3 Training Hyperparameters

Rejection Sampling Fine-Tuning. For RFT we use three data components: (i) our internal 200K general SFT responses used to form the Baseline-SFT checkpoint; (ii) 7,537 TACO-RFT responses derived from the TACO-Verified corpus as the in-domain baseline; and (iii) 61,133 Task I and 72,430 Task II rejection-sampled responses from CODE-TVT. Baseline-SFT generates eight responses per instance. We retain one reward-1 response and omit instances without one; invalid, degenerate, and reward-0 rollouts are excluded. All RFT experiments are conducted using the LLaMA-Factory (Zheng et al., 2024) framework with a cosine learning rate schedule, initial learning rate 1×10^{-5} , warmup ratio 0.1, total batch size 128, 3 epochs, and maximum sequence length 32,000, on a cluster of 16 NVIDIA H20 GPUs.

¹<https://huggingface.co/datasets/likeixin/TACO-verified>

Reinforcement Learning. All RL experiments are implemented using rllm with GRPO. Each optimization step uses 128 prompts and eight rollouts per prompt (1,024 trajectories), with a PPO mini-batch size of 64 and one PPO epoch. We use a constant actor learning rate of 1×10^{-6} , asymmetric PPO clipping with $\epsilon_{\text{low}} = 0.20$ and $\epsilon_{\text{high}} = 0.28$, and no KL regularization ($\beta = 0$). Rollouts are sampled at temperature 1.0 and top- $p = 1.0$, with a maximum response length of 6,144 tokens. We use random seed 2026 and save and validate checkpoints every 10 optimization steps.

B.4 Evaluation Protocols and Terminology

All inference-time evaluations are conducted using vLLM (v0.11.2) (Kwon et al., 2023). We sample 8 responses per prompt with temperature 0.7, top- p 0.8, top- k 20, and a repetition penalty of 1.05. LiveCodeBench and CRUXEval are evaluated via sandbox execution; BBH subtasks use answer-normalized exact-match scoring.

Nominal sampling budget and effective denominator. The $n=8$ setting is the requested number of responses per prompt. Our internal evaluation platform runs inference and evaluation as separate stages, and downstream collection may occasionally fail to retrieve an inference output or its evaluation record. Split-level metrics therefore use the records successfully retrieved and matched for scoring in each run. This infrastructure-level missingness can make the effective denominator slightly smaller than the nominal split size and can vary across runs. Thus, the main-text split sizes describe benchmark membership, while platform aggregates reflect the records available to the corresponding run; a reported Pass@8 value need not be an exact multiple of $100/N$ for nominal split size N . The missingness is not a correctness-, reward-, or model-based filter.

C Reward-Hacking Safeguards

This appendix answers two questions: what reward-hacking shortcut each task admits, and how the verifier blocks it. We use $\text{Exec}(y, a) = (s, o)$, where $s = 1$ means normal execution and $s = 0$ covers runtime errors, timeouts, memory failures, and parse failures; $\text{norm}_x(\cdot)$ is the problem-specific output normalizer.

Table 4: **Task I invalid-input audit for edge test-case generation.** Naive output-divergence rewards frequently admit invalid edge inputs, motivating the final validity gate.

Outcome	Count	Rate
Naive-reward-positive Task I samples	17,108	100.00%
Validator-covered samples	12,798	74.81% of all
Valid input	5,858	45.77% of covered
Invalid input	6,940	54.23% of covered
No validator spec / unknown	4,310	25.19% of all

C.1 Task I: Edge Test-Case Generation

Hacking problem. A naive edge-test reward could treat malformed stdin, reference failures, candidate crashes, or timeouts as “different outputs,” giving positive reward without a valid semantic counterexample.

Solution. We first require the generated input $\hat{a}$ to satisfy the problem validator V_x , and require both the candidate program y and reference program y^* to execute normally. Only then do we compare normalized outputs:

$$\text{cmp}_x(y, y^*, \hat{a}) \in \{\text{equal}, \text{different}, \text{invalid}\}.$$

The state is invalid unless $V_x(\hat{a}) = 1$ and both executions return $s = 1$. The reward is

$$R_{\text{edge}}(x, y, y^*, \hat{a}) = \mathbf{1}[\text{cmp}_x(y, y^*, \hat{a}) = \text{different}].$$

Thus invalid execution is never counted as semantic disagreement.

Audit. This gate matters empirically: among 12,798 validator-covered naive reward-positive Task I samples, 54.23% fail input validation and would be rejected by the final verifier (Table 4).

C.2 Task II: Input Reconstruction

Hacking problem. Output matching alone can reward overly small or shallow inputs that reproduce the target output without solving the intended inverse program-reasoning task.

Solution. The reconstructed input must pass four gates: valid input, normal reference execution, normalized output match, and a non-degenerate score relative to source tests. Let $(s_*, o_*) = \text{Exec}(y^*, \hat{a})$ and define the predicate

$$\begin{aligned} \text{success}_x(\hat{a}, b) \Leftrightarrow & V_x(\hat{a}) = 1 \wedge s_* = 1 \\ & \wedge \text{norm}_x(o_*) = \text{norm}_x(b) \\ & \wedge G_x(\hat{a}; T_x) \geq \gamma_x, \end{aligned}$$

Table 5: **Task II degenerate-input audit for input reconstruction.** Output-match-only rewards often admit overly small or shallow reconstructions, motivating the final non-degenerate gate.

Outcome	Count	Rate
Task II rollout samples	17,216	100.00%
Reward-positive samples	5,056	29.37%
Baseline-covered reward-positive	3,424	67.72% of positive
Degenerate input	1,875	54.76% of covered
Non-degenerate input	1,549	45.24% of covered
Early / middle / late degenerate rate	–	62.0 / 56.9 / 51.4%

Table 6: **Verifier telemetry for reward-hacking diagnosis.** The logged fields expose each gate used by the executable reward verifier before reward assignment.

Field	Meaning
input_valid	$V_x(\hat{a}) \in \{0, 1\}$: parsing and constraints
ref_status	$s_* \in \{0, 1\}$: reference halts normally
cand_status	$s_y \in \{0, 1\}$: candidate halts normally
compare_status	equal, different, or invalid
failure_type	Runtime error, time/memory limit exceeded, parse error
resource_usage	Runtime and memory diagnostics
coverage_score	Task II coverage/non-triviality score G_x
coverage_source	Instrumented coverage or structural fallback

so that the Task II reward of §2.3 is exactly $R_{\text{inv}}(x, y^*, b, \hat{a}) = \mathbf{1}[\text{success}_x(\hat{a}, b)]$. Here G_x is either instrumented coverage or a structural fallback such as input length, number of elements, graph size, grid area, or nesting depth; γ_x is set from the source-test distribution for the same problem.

Audit. This gate also matters empirically: among 3,424 baseline-covered output-match-positive Task II samples, 54.76% are degenerate and would be filtered by the final non-degenerate gate (Table 5).

C.3 Training-Time Telemetry and Scope

We log verifier outcomes during rejection sampling and RL so the two shortcuts above are measurable rather than hidden. Table 6 lists the fields used to distinguish invalidity, execution failure, semantic disagreement, and Task II non-degeneracy before reward assignment.

For RFT selection (§B.3), invalid covers parsing, input-validity, or execution failures; Task II degeneracies are also excluded. Reward-0 (Task I equality or Task II mismatch) is diagnostic only, never an SFT target. The audits above motivate the final gates; they are diagnostic checks rather than a formal proof against every possible reward-hacking strategy.

D Comparison with CodeI/O Input Prediction

This section clarifies the relationship between CODE-TVT Task II and CodeI/O’s input prediction task. Both tasks reason over an inverse program relation, but they optimize different objects. Let p denote a program and let (a, b) be an input–output pair such that $p(a) = b$. CodeI/O constructs supervised examples that ask a model to predict either the output from the input or the input from the output, usually with a natural-language chain-of-thought rationale z . For the input-prediction direction, the learning signal can be viewed as a supervised objective of the form

$$\max_{\theta} \log \pi_{\theta}(a, z \mid p, b, \mathcal{E}),$$

where $\mathcal{E}$ denotes the surrounding CodeI/O context such as demonstrations or test cases. Execution is used to verify or revise the predicted answer and rationale, but the training target remains a supervised I/O reasoning trace.

In CODE-TVT Task II, an instance is (x, y^*, b) , where x is the original programming problem, y^* is a verified reference implementation, and b is obtained by executing y^* on a valid, source-like input proposed under source-test-derived constraints. The policy emits an input $\hat{a}$ and receives the execution-gated reward

$$R_{\text{inv}}(x, y^*, b, \hat{a}) = \mathbf{1}[\text{success}_x(\hat{a}, b)],$$

where $\text{success}_x(\hat{a}, b)$ is the four-conjunct predicate (validator + execution status + normalized output match + non-triviality threshold) defined in §2.3 and Appendix C; here $(s_*, o_*) = \text{Exec}(y^*, \hat{a})$ and G_x is the coverage/non-triviality guard used by the Task II gated verifier. The task therefore defines a reward-bearing environment rather than a fixed supervised target. Any input passing the four gates—validity, execution, output match, and non-triviality—is accepted, making the objective one-to-many.

The practical distinction follows from this formulation. First, the difference is broader than replacing SFT with RL. CodeI/O is designed to mine broad reasoning patterns from large-scale code and to scale supervised prediction and self-correction data during mid-training or continued SFT. CODE-TVT instead converts existing test cases into **executable post-training environments**: candidate actions are sampled from the

policy, judged by execution, and used for rejection sampling or GRPO updates. This connects the training signal to the same sampling-and-selection mechanism that underlies **test-time scaling** for code, rather than treating I/O reasoning as a fixed supervised target. Second, CodeI/O studies the benefit of scaling I/O prediction data mined from code, but does not isolate the marginal value of such expansion once the original code-generation problems have already been used for training. Our experiments are explicitly set up in this incremental regime: TACO problems form the in-domain baseline, and CODE-TVT tests whether synthesizing additional test-case-driven tasks from those verified problems yields further gains. Third, CodeI/O’s verifier is primarily an offline correctness filter for I/O predictions and rationales, whereas CODE-TVT’s verifier is the reward function seen by the policy during optimization. This makes **reward hacking** a central concern for CODE-TVT: invalid stdin, execution failures, timeout artifacts, or trivial inverse solutions could otherwise become high-reward actions. Fourth, CODE-TVT ties each synthesized task to a problem-specific input validator, output normalizer, reference program, and anti-trivial coverage threshold derived from the source test distribution. These components are not just data-cleaning details; they define which policy actions are rewarded.

Empirically, we compare against CodeI/O at the task-formulation level rather than treating it as a direct data baseline, because CodeI/O targets supervised I/O prediction whereas CODE-TVT targets execution-gated policy optimization. The task-component ablation in Table 8 further shows that Task II is the strongest single component on the newer LCB-V5/V6 Pass@1 columns, consistent with its role as inverse program-reasoning supervision. Thus, the contribution of CODE-TVT is not simply that it exposes models to program I/O pairs, but that it converts test cases into executable, verifiable reward functions for policy learning.

E Additional Main Experimental Results

This section complements the compact main result table with two results that use a different scope: the Llama-3.1-8B replication, which is not part of the Qwen-family main table, and the Stage III mixed-RL training curves, which are reported separately from the backbone-wide Stage I/II table.

Table 7: **Llama-3.1-8B replication of the Stage I fine-tuning and Stage II reinforcement-learning pipeline.** Results are reported on the same LiveCodeBench splits as the Qwen-family main table (LCB-08/11, LCB-V5, LCB-V6), with Code Avg computed across reported splits. § under-budgeted 80-step RL run. Best per column in **bold**.

Model	LCB-08/11		LCB-V5		LCB-V6		Code Avg	
	P@1	P@8	P@1	P@8	P@1	P@8	P@1	P@8
Baseline-SFT	5.7	10.2	3.6	11.4	8.7	17.8	6.00	13.13
+ CODE-TVT	10.6	22.9	10.0	22.2	14.1	24.0	11.57	23.03
+ TACO	11.9	19.5	11.0	20.4	14.9	24.1	12.60	21.33
+ TACO & CODE-TVT	14.5	27.7	14.7	27.1	18.6	29.6	15.93	28.13
RL from Baseline-SFT	8.4	12.7	13.8	25.0	14.0	20.0	12.07	19.23
RL from + CODE-TVT §	14.6	26.3	13.9	25.5	12.1	22.9	13.53	24.90

E.1 Llama-3.1-8B: Fine-Tuning and RL Results

We applied the same Stage I/II pipeline to Llama-3.1-8B-Base. As with the Qwen-family experiments, Stage I first produces self-trained SFT/RFT checkpoints from the base model, and Stage II GRPO continues from those checkpoints. Stage I RFT behaves consistently with the three Qwen-family backbones: adding CODE-TVT on top of Baseline-SFT raises Code Avg Pass@1 by +5.6 pt and Pass@8 by +9.9 pt, the largest cross-backbone gain in our experiments. Stage II GRPO-RL is less stable on this backbone: the CODE-TVT-initialized RL setting improves the two older LCB splits relative to the + CODE-TVT RFT checkpoint, while the harder LCB-V6 split is weaker after early RL updates. This behavior is consistent with Llama-3.1-8B starting from a weaker competitive-programming baseline than the Qwen-family backbones, which leaves a narrower margin for stable GRPO optimization on the hardest split. Table 7 reports the RFT and Stage II RL numbers for Llama-3.1-8B.

E.2 Mixed Reinforcement Learning Curves

This subsection reports the full three-split Stage III Pass@8 curves in Figure 5. Stage III keeps most CODE-TVT prompts in RFT, while 1,000 hard Task I prompts and 1,000 hard Task II prompts are added to the 7,620-prompt TACO RL train file. Both runs start from the same + CODE-TVT RFT checkpoint; the only difference is whether the RL train file is TACO-only or TACO mixed with the heldout CODE-TVT split.

F Additional Analysis Results

This section collects diagnostics that explain why CODE-TVT helps and how its effects vary across

Table 8: **Full Task Component Ablation on Qwen3-4B.** Pass@1 / Pass@8 on LiveCodeBench (LCB-08/11, V5, V6). Best per column within each block in **bold**.

Model	LCB-08/11		LCB-V5		LCB-V6	
	P@1	P@8	P@1	P@8	P@1	P@8
(a) no-TACO (general-data baseline)						
Baseline-SFT	17.4	28.9	15.1	26.9	19.8	30.9
+ Task I	20.0	32.5	17.2	34.1	20.1	32.0
+ Task II	20.4	30.1	20.0	35.3	21.0	33.1
+ Full	19.4	28.3	18.3	31.7	21.1	31.4
(b) TACO already in SFT						
+ TACO	22.1	31.3	19.9	32.9	21.9	34.9
+ TACO + Task I	20.7	30.7	20.4	32.9	23.0	33.1
+ TACO + Task II	20.0	28.3	21.0	35.9	23.3	34.3
+ TACO + Full	20.0	27.7	20.1	35.3	21.6	33.1

task composition, reasoning probes, and difficulty slices.

F.1 Full Task Component Ablation

This subsection complements the compact main-text Table 2. We report two complementary recipes on Qwen3-4B: (i) a **no-TACO** block that uses the general-data-only Baseline-SFT as reference and adds each CODE-TVT component without TACO (matching the main-text table); and (ii) a **TACO-already-in-SFT** block that uses the general-data + TACO recipe as reference and adds Task I / Task II / full CODE-TVT on top, isolating the marginal benefit of each component when TACO is already present.

Block (a): no-TACO Task Ablation. This block expands the main-text task ablation (Table 2) across all reported LiveCodeBench Pass@1 and Pass@8 columns. Both Task I and Task II individually outperform Baseline-SFT on every LiveCodeBench split, while the full CODE-TVT mixture attains the overall best LCB-V6 Pass@1 (21.1). At the Pass@8 sample-diversity ceiling, however, the two components diverge: Task I peaks on LCB-08/11 (32.5) while Task II peaks on LCB-V5 / LCB-V6 (35.3 / 33.1), consistent with the view that test-case generation and program-output inversion provide complementary but non-identical supervision over LCB.

Block (b): TACO-already-in-SFT Variants. This block answers a different question: given that TACO has already been mixed into the SFT data, does adding CODE-TVT-Task I, CODE-TVT-Task II, or the full CODE-TVT mixture pro-

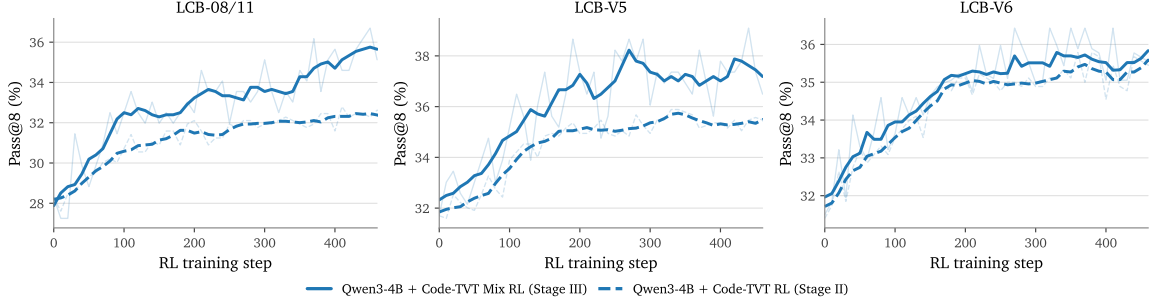


Figure 5: **Full Stage III mixed-RL training curves on Qwen3-4B (Pass@8).** This expands the main-text LCB-V6 view in Figure 3 to all three LiveCodeBench splits. Solid: mixed RL; dashed: the Stage II TACO-only RL baseline from the same + CODE-TVT RFT checkpoint.

vide additional benefit? Adding either Task I or Task II on top of TACO consistently improves Pass@1 on the more recent LCB-V5 / V6 splits (**+0.5 to +1.4** pt over +TACO), with Task II reaching the block-best Pass@1 on both LCB-V5 (**21.0**) and LCB-V6 (**23.3**) and the block-best LCB-V5 Pass@8 (**35.9**). The full CODE-TVT mixture improves LCB-V5 over +TACO but does not dominate the single-component variants on LCB-V6. In contrast, +TACO retains the strongest LCB-08/11 Pass@1 / Pass@8 and LCB-V6 Pass@8, suggesting that TACO continues to drive the older LCB distribution while the individual CODE-TVT components are most useful on the newer V5 / V6 Pass@1 columns.

Together, the two blocks indicate that the gains of CODE-TVT are robust to the presence or absence of TACO in the SFT base, but they are not uniformly additive across all LCB columns. With no TACO, both task components individually exceed the supervised baseline and the full mixture reaches the best LCB-V6 Pass@1; with TACO already in the SFT base, the single-component CODE-TVT variants provide the clearest marginal gains on the newer V5 / V6 Pass@1 columns.

F.2 Auxiliary BBH Subtask Diagnostic

Following the main-text analysis in §4.2, we use BBH as an auxiliary subtask-level diagnostic rather than a primary benchmark. The main table presents three illustrative subtasks whose formats relate to temporal execution, object-state tracking, and structured spatial reasoning. This compact, mechanism-oriented view is exploratory rather than an aggregate or confirmatory BBH evaluation, and it should not be interpreted as evidence that CODE-TVT improves BBH globally. Table 9 reports the complete Qwen3-4B BBH sub-

task results for the four main SFT variants.

The full results are heterogeneous. Relative to Baseline-SFT, +Code-TVT is higher on 17 of the 27 subtasks and lower on 10; the simple unweighted macro average changes from 66.61 to 67.66 (**+1.05** pt). The corresponding averages for +TACO and +TACO & Code-TVT are 67.42 and 66.65. Some of the larger decreases occur on Hyperbaton, MovieRec, DisambigQA, and Sports. We therefore treat the three main-table columns only as illustrative probes of task-specific behavior, not as evidence of uniform broad-reasoning improvement.

F.3 Difficulty Stratification: Pass@k, Original Difficulty, Skills, and Input Length

Overview. This appendix expands the main-text comparison in §4.3 with four stratified views. The central question is whether the lower performance on CODE-TVT simply reflects harder source problems, or whether the transformed tasks introduce additional reasoning demands. We therefore compare the original TACO task with Task I and Task II across sampling budget, original difficulty labels, algorithmic skill tags, and input length. The Qwen3 models in these figures use the July 2025 (2507) non-thinking Instruct releases: Qwen3-4B-Instruct-2507, Qwen3-30B-A3B-Instruct-2507, and Qwen3-235B-A22B-Instruct-2507.²

Pass@k. As shown in Figure 6, all evaluated models score substantially lower on the CODE-TVT tasks than on the original TACO bench-

²HuggingFace model cards: <https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507>, <https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507>, and <https://huggingface.co/Qwen/Qwen3-235B-A22B-Instruct-2507>.

Table 9: **Complete Qwen3-4B Big-Bench Hard (BBH) subtask diagnostic (Pass@1, %)**. All rows share the same general-data reference and compare Baseline, CODE-TVT, TACO, and TACO+CODE-TVT variants across individual BBH subtasks. This auxiliary table is used to inspect heterogeneous subtask-level behavior; the main text reports CRUX together with three illustrative BBH columns in Table 3.

Subtask	Base	+CODE-TVT	+TACO	+TACO&CODE-TVT	Subtask	Base	+CODE-TVT	+TACO	+TACO&CODE-TVT
Dyck	32.8	25.8	32.9	22.5	ObjCount	53.1	60.7	59.1	56.9
Track3	93.0	92.0	92.8	89.8	FormalFall	69.2	75.3	72.6	72.7
LogDed7	60.4	60.6	62.3	60.4	Hyperbaton	71.5	61.1	67.6	60.4
Snarks	61.7	64.0	62.9	64.4	GeoShape	43.2	51.0	44.9	45.8
LogDed3	91.4	91.1	91.8	91.1	RuinNames	45.3	46.9	47.9	47.8
TempSeq	65.3	71.0	64.5	67.8	Track5	83.0	85.6	87.4	85.7
LogDed5	70.0	69.3	72.1	71.8	SaliTrans	51.2	52.5	52.6	51.7
MultiArith	97.2	95.2	97.2	95.3	Penguins	93.7	94.6	94.3	93.7
CausalJ	59.1	59.2	58.2	58.2	Bool	90.4	89.1	89.8	87.5
ColorObj	82.7	83.7	85.0	84.6	WebLies	77.8	82.5	78.8	77.1
DisambigQA	45.1	40.8	41.4	43.4	MovieRec	58.0	52.1	51.5	50.8
Date	66.5	66.7	68.0	66.9	Track7	64.3	79.2	71.7	78.5
WordSort	29.3	30.0	30.7	30.6	Sports	59.6	58.0	58.1	56.5
Navigate	83.6	88.8	84.3	87.7					

mark. The average Pass@1 gap exceeds 20 absolute points for Task I and 10 absolute points for Task II. Increasing the sampling budget narrows some model-specific gaps, but it does not remove the separation from TACO; for Task II, the gap remains especially pronounced under larger k . This pattern suggests that the separation reflects task-specific capabilities—semantic verification and counterexample construction for Task I, and inverse input reconstruction for Task II—rather than sampling variance alone.

Original difficulty. We next examine performance across TACO’s difficulty levels (Figure 7). Task I is consistently below the original task in every difficulty bucket; even on “Easy” problems, gpt-oss-20B changes from 60.0% on TACO to 16.96% on Task I. Thus, an easy source problem can still become difficult once the model must judge a candidate solution and construct a distinguishing test case. Task II shows a related but distinct effect: performance does not strictly follow the original TACO difficulty curve, and “Easy” source problems can remain hard after the inverse transformation (e.g., Qwen3-30B-A3B-Instruct-2507 changes from 78.17% to 33.44%). This indicates that the original labels capture implementation complexity, whereas Task II is additionally governed by the size and structure of the inverse search space.

Original skill classification. Figure 8 further shows that the transformed-task difficulty depends on the algorithmic structure of the source problem. For Task II, skills such as *Bit Manipulation* and *Complete Search* are especially difficult, plau-

sibly because the output often compresses or aggregates information from the input. By contrast, *Sorting* and *Greedy Algorithms* are comparatively more robust; Qwen3-235B-A22B-Instruct-2507 even improves from 69.49% to 75.35% on *Sorting*. These cases suggest that inverse reconstruction is easier when the output preserves visible input structure, and harder when the forward computation discards information needed to recover a valid input.

Input Length. Finally, Figure 9 shows that accuracy generally decreases as input length grows, which is expected for both code generation and the two CODE-TVT transformations. For Task II, this length sensitivity is particularly informative: longer inputs expand the space of possible preimages for a target output, so the transformed task can become harder even when the original problem label is unchanged. Together, the four views support the same conclusion as the main text: CODE-TVT creates difficulty along axes that are related to, but not reducible to, original TACO generation difficulty.

G Qualitative Examples

This section grounds the quantitative trends in task-native examples for Task I and Task II. The examples illustrate the verifier-facing skills targeted by CODE-TVT—edge-case construction and semantic checking in Task I, and output-to-input inversion in Task II—and serve as explanatory support rather than quantitative evidence.

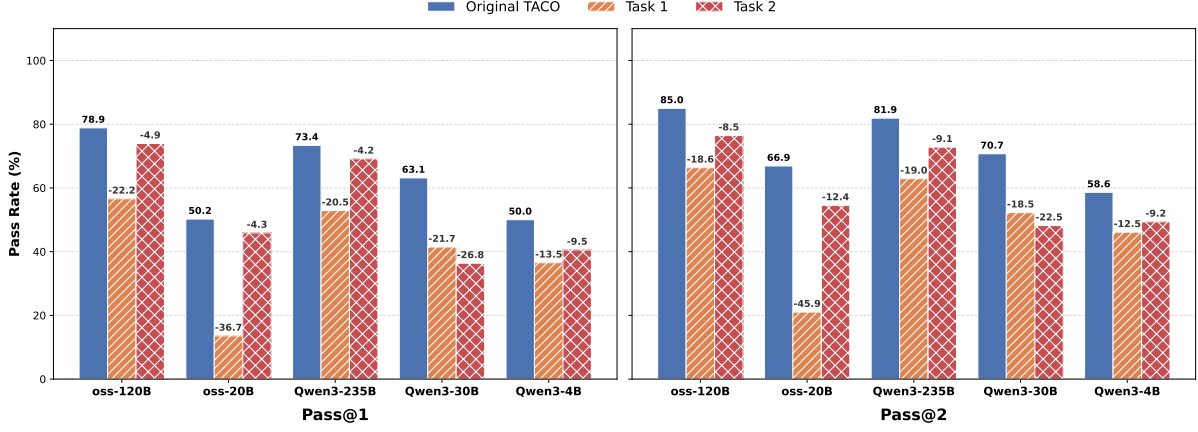


Figure 6: Pass@1 and Pass@2 scores on the original TACO benchmark and the CODE-TVT tasks (Task I and Task II) for gpt-oss-120B, gpt-oss-20B, Qwen3-235B-A22B-Instruct-2507, Qwen3-30B-A3B-Instruct-2507, and Qwen3-4B-Instruct-2507.

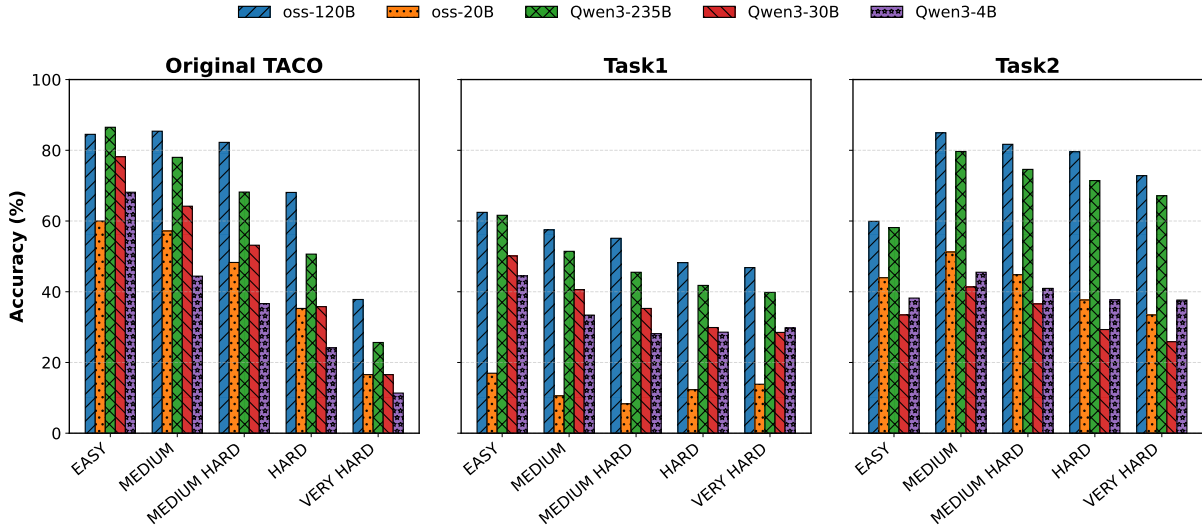


Figure 7: Accuracy of gpt-oss-120B, gpt-oss-20B, Qwen3-235B-A22B-Instruct-2507, Qwen3-30B-A3B-Instruct-2507, and Qwen3-4B-Instruct-2507 on the Original TACO, Task I, and Task II datasets across difficulty levels from Easy to Very Hard.

G.1 Task I Examples

This Task I case expands the main-text observation that solving the original code-generation problem does not guarantee success on the corresponding verification task. The model can generate a correct forward solution, but still miss a subtle logical flaw in another candidate solution when asked to construct a distinguishing edge input.

Success in Original Generation. The problem asks to find the k -th “Cross” constellation in a star map, defined by 5 stars (center and four arms at distance x) at specific relative coordinates. In the original generation task, the model successfully produces a correct solution (Figure 10, left).

The generated code adopts a straightforward simulation approach: it iterates through all possible centers and radii, directly checking the existence of the 5 required stars. This approach is robust because it directly translates the problem definition into code without introducing unnecessary assumptions about the intermediate cells.

Task I Verification Gap. In Task I, the model is presented with a more complex candidate solution (Figure 10, right) that attempts to optimize the radius calculation using precomputed arrays (up, down, etc.). These arrays count the number of *consecutive* stars in each direction. The candidate solution then computes the maximum radius as the

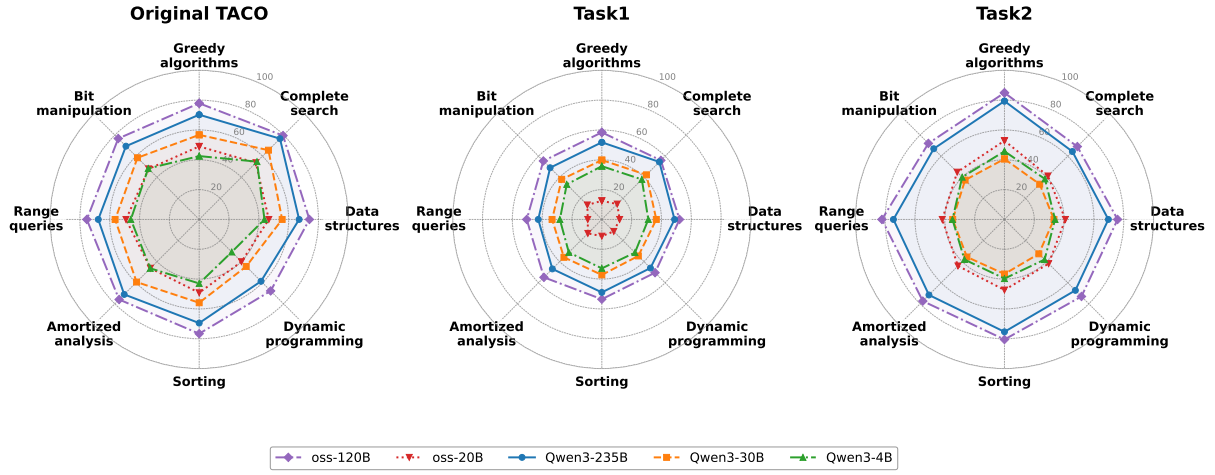


Figure 8: Accuracy of gpt-oss-120B, gpt-oss-20B, Qwen3-235B-A22B-Instruct-2507, Qwen3-30B-A3B-Instruct-2507, and Qwen3-4B-Instruct-2507 on the Original TACO, Task I, and Task II datasets across various algorithmic skills.

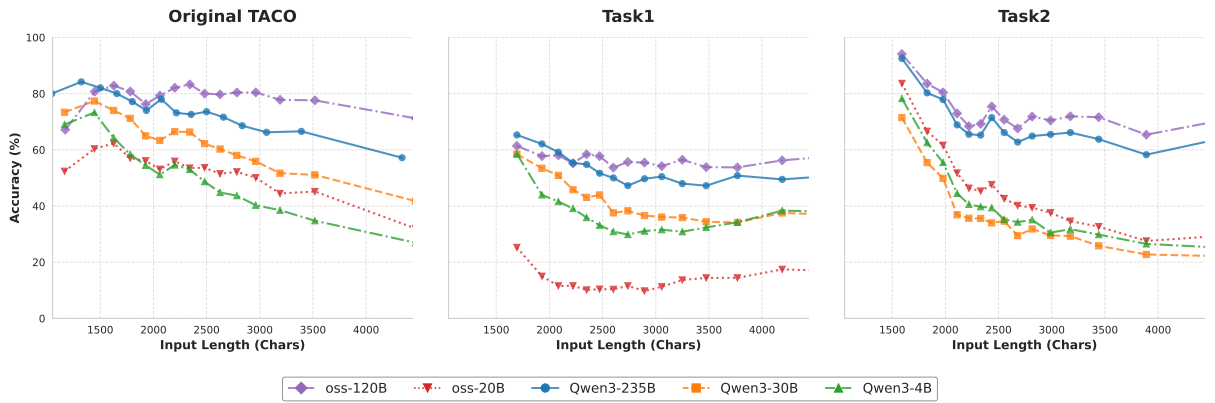


Figure 9: Accuracy of gpt-oss-120B, gpt-oss-20B, Qwen3-235B-A22B-Instruct-2507, Qwen3-30B-A3B-Instruct-2507, and Qwen3-4B-Instruct-2507 on the Original TACO, Task I, and Task II datasets as input lengths increase.

minimum of these consecutive counts. This logic contains an important flaw: the problem definition does not require the stars between the center and the tips to be present (i.e., the cross can be “hollow”). However, the candidate solution implicitly assumes that the arms must be solid lines of stars. Despite this error, the model marks the candidate solution as “Correct.” It fails to generate an edge test case (e.g., a hollow cross with empty spaces between the center and tips) that would expose this flaw. This case highlights a concrete verification gap: the model is misled by code that appears algorithmically sophisticated (using prefix sums/DP) and locally consistent, and does not check whether the implementation strictly follows the boundary conditions in the problem statement.

G.2 Task II Examples

This Task II case expands the main-text example of output-to-input inversion. The representative scheduling problem in Figure 11 shows two common error modes: misunderstanding branch or statement conditions, and introducing inconsistent fields during long-context input generation.

The task requires reconstructing an input that produces the target output indices “4 1 6 9”. The reference program contains a critical logic statement: `listi = sorted(listi, key=lambda x: x[1])`. This means the final output is not sorted in the order the items are input, but by their time cost t .

Ignoring the Sorting Logic. The model’s error stems from misreading how the variable t dictates the execution order. Since the reference program explicitly prioritizes problems with smaller time

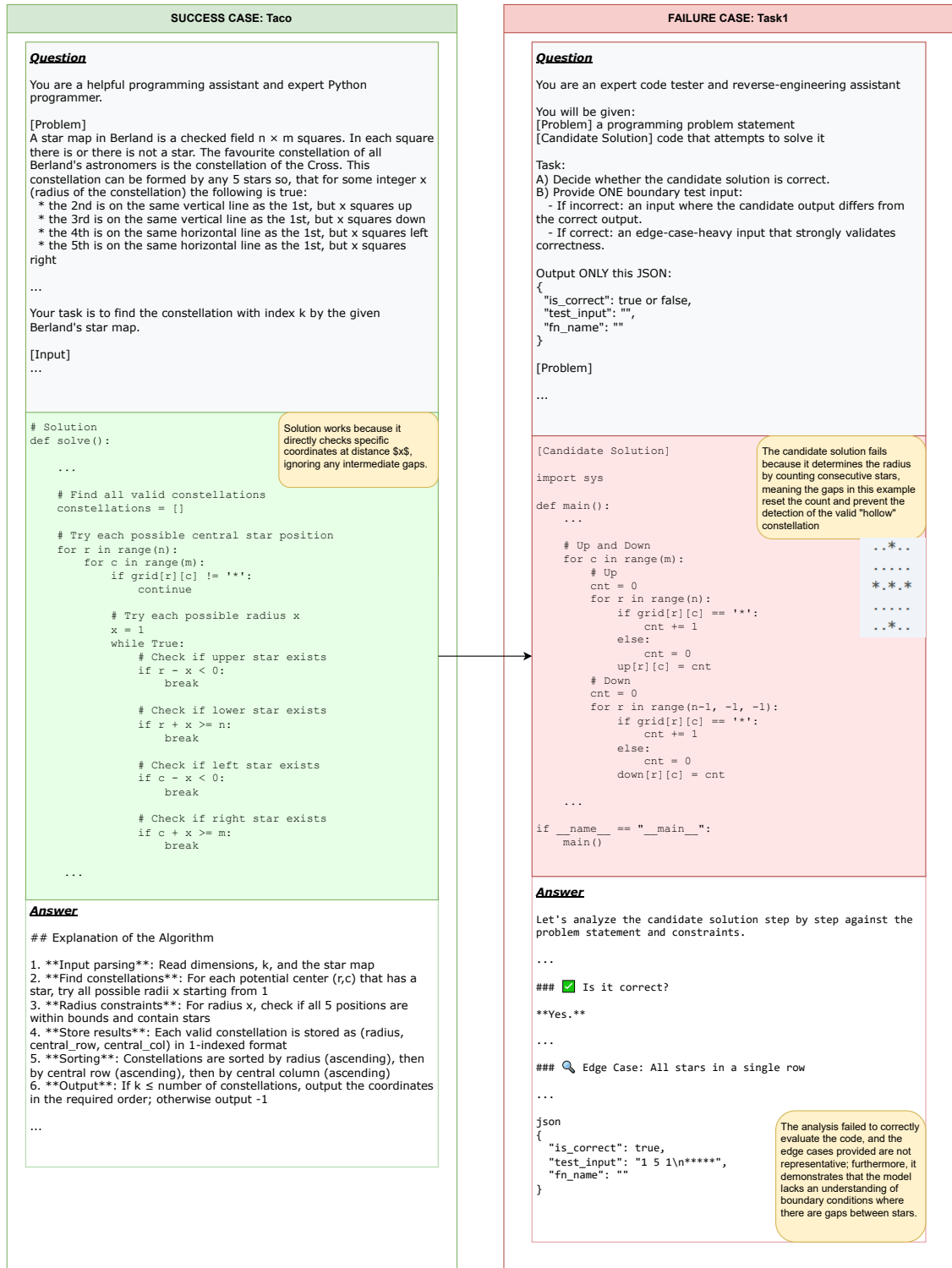


Figure 10: Case study of the model on Task I (Edge Test-Case Generation). The candidate program y counts only *consecutive* stars in each direction, so it fails on “hollow” constellations where stars are separated by gaps. Instead of generating an edge input $\hat{a}$ that exposes this error, the model marks the candidate as “Correct,” i.e., $\text{cmp}_x(y, y^*, \hat{a}) \neq \text{different}$ and $R_{\text{edge}} = 0$.

costs via `sorted(listi, key=lambda x: x[1])`, the target output “4 1 6 9” implies that Problem 4 must have a smaller time cost than Problem 1 ($t_4 < t_1$) to appear first in the sorted sequence.

However, the model creates a conflict by assigning Problem 1 a minimal time cost ($t = 1$) while assigning Problem 4 a higher cost ($t = 5$) and Problem 5 an intermediate cost ($t = 2$). By assigning

SUCCESS CASE: Taco	FAILURE CASE: Task2
Question You are a helpful programming assistant and expert Python programmer. [Problem] You are taking an exam with (n) problems and a total duration of (T). Each problem (i) takes (t_i) time to solve and has a threshold (a_i). Your task is to select a subset of problems to solve. Let (k) be the total number of problems you choose to solve. A problem (i) contributes 1 point to your score if and only if (k ≤ a_i). Your goal is to select a set of problems such that the total time does not exceed (T) and the final score is maximized. [Input&Output] ... Answer This is a scheduling problem with constraints on both time and scoring rules. ... 4. Greedy Strategy: We should try different values of k from n down to 0, and for each k: - We can only get a point from problems with a_i ≥ k - We need to select exactly k problems such that each selected problem has a_i ≥ k - We want to minimize the total time required for these k problems ... # Solution <pre>def solve(): n, T = map(int, input().split()) problems = [] for i in range(n): a, t = map(int, input().split()) problems.append((a, t, i + 1)) # (a_i, t_i, original_index) # Try each possible number of problems to solve (from n down to 0) max_score = 0 best_k = 0 best_solutions = [] # Try from high to low k to maximize score for k in range(n, 0, -1): # Get all problems where a_i ≥ k candidates = [] for a, t, idx in problems: if a ≥ k: candidates.append((t, idx)) # Sort by time (ascending) candidates.sort() # Check if we can select k problems from candidates if len(candidates) ≥ k: total_time = sum(t for t, idx in candidates[:k]) if total_time ≤ T: # We found a valid solution with score k max_score = k best_k = k best_solutions = [idx for t, idx in candidates[:k]] break # If no solution found, k = 0 if best_k == 0: print(0) print(0) print(0) else: print(max_score) print(best_k) print(*best_solutions) solve()</pre> The model correctly implements a greedy algorithm to maximize the final score by iterating through possible values of k.	Question You are an expert code tester. I will give you: 1) A programming problem statement. 2) A candidate solution implemented as code. 3) A TARGET OUTPUT produced by running the candidate solution on some unknown input. Propose ONE plausible input such that running the candidate solution on this input produces the TARGET OUTPUT exactly [Problem] ... [Target output] 4 4 1 6 9 [Candidate Solution] <pre>import sys import operator import itertools (n, T) = list(map(int, input().split())) list1 = [] for i in range(n): (a, b) = list(map(int, input().split())) list1.append((a, b, i)) list1 = sorted(list1, key=lambda x: x[1]) time = [0] * (n + 1) count = [0] * (n + 1) time_sum = 0 count_sum = 0 k = 1 for (a, t, ind) in list1: if k ≤ a: time[a] += t time_sum += t if time_sum > T: break count[a] += 1 count_sum += 1 if count_sum == k: count_sum -= count[k] time_sum -= time[k] k += 1 max_score = max(count_sum, k - 1) print(max_score) print(max_score) print(*list(itertools.islice((idx + 1 for (a, t, idx) in list1 if a ≥ max_score), max_score)))</pre> Answer We are given: - A TARGET OUTPUT: "4\n4\n4 1 6 9" ... ### Step 6: Construct Input Let: - n = 9, T = 10 - For problems: 1,4,6,9 → a ≥ 4 - Others (2,3,5,7,8) → a < 4 ... Now k = 5, count_sum = 2 → so max_score = max(2, 5 - 1) = 4 ... <pre>{ "test_input": "9 10 4 1 2 10 3 15 1 5 5 2 4 3 3 20 2 25 6 4", "fn_name": "" }</pre> The model's answer was wrong primarily because it overlooked that the candidate code sorts the output indices by time (t); to achieve the target order 4 1 6 9, Problem 4 must have a smaller t than Problem 1, yet the model assigned t₁ < t₄, inevitably forcing the incorrect sequence 1 4... This logical flaw was further compounded by a transcription error where the input parameters for indices 4 and 5 were swapped in the final JSON string, causing the code to erroneously select Index 5 instead of Index 4.

Figure 11: Case study of the model on Task II (Input Reconstruction). Given the target output “4 1 6 9” and reference program y^* , the model produces a reconstructed input $\hat{a}$ that, after y^* executes, yields “1 5 6 9” instead of “4 1 6 9”. The normalized output therefore mismatches the target, so the gated reward $R_{\text{inv}} = 0$.

the lowest t value to Problem 1, the model forces the algorithm to process and output Problem 1 first. The generated input therefore follows the line order of the input rather than the ascending order of

t , producing an output starting with “1 . . .” instead of the target “4 . . .”.

Generation Inconsistency. The model also introduces an inconsistency during final input gen-

eration. While its reasoning plan intended to assign valid parameters to Problem 4, the final string swaps the lines for Problem 4 and Problem 5. This creates an input where Problem 4 receives invalid parameters ($a = 1$) and Problem 5 receives valid ones ($a = 5$), leading to the actual output “1 5 6 9” instead of the target “4 1 6 9”.