Function-Level Execution Feedback for Code Preference Optimization

Idris Nechnech[1], **Sehwan Kim**[1], **Jimin Seo**[1], **Yeongoon Kim**[1],
Minhae Oh[1], **Sangwoo Hong**[2], **Jungwoo Lee**[1†]
[1]Department of Electrical and Computer Engineering, Seoul National University
[2]Department of Computer Science and Engineering, Konkuk University
{inechnech,junglee}@snu.ac.kr

Abstract

Process supervision has improved mathematical reasoning, where intermediate steps are naturally expressed as chains of thought. In code generation, however, process supervision remains underexplored because there is no standard notion of a step. Supervision can target lines, reasoning traces, or program states, making it unclear what to label and optimize. We propose STEP-KTODER, a framework for code preference optimization that defines steps as module-level functions in decomposed multi-function programs and assigns binary correctness labels via automatically generated unit tests. Our method provides a code-specific instantiation of stepwise KTO, combining function-level process supervision with outcome-level feedback on the full program. We evaluate on HumanEval(+), MBPP(+), BigCodeBench, and LiveCodeBench, showing that STEP-KTODER improves over outcome-only KTO and DPO. Further analysis shows that execution-based labels are essential: LLM-as-a-judge annotations systematically over-predict function failures, corrupt positive step labels, and degrade downstream preference optimization. Code is available at: https://github.com/inechnech/STEP-KTODER.

1 Introduction

Code generation has become a major application of Large Language Models (LLMs), with recent code-specialized models achieving strong performance on a wide range of programming tasks (Guo et al., 2024; Hui et al., 2024). These models are typically improved through two stages: supervised fine-tuning (SFT) (Wei et al., 2022) on high-quality code data, followed by an alignment stage that adapts the model toward better outputs (Ouyang et al., 2022). Among alignment methods, Direct Preference Optimization (DPO) (Rafailov et al., 2023) is widely used to learn from paired preference data, whereas Kahneman–Tversky Optimization (KTO) (Ethayarajh et al., 2024) learns from binary desirability labels on individual outputs.

However, applying these techniques to code generation is nontrivial. These methods are typically instantiated at the level of the entire program, treating a full solution as a single unit of feedback. This provides only a coarse training signal: a program can be mostly correct and fail only on a narrow edge case, but outcome-level supervision still marks the entire output as undesirable. Such supervision does not reveal which component is correct and which is responsible for the failure, motivating finer-grained supervision for code generation.

In mathematical reasoning, *process supervision*—providing feedback on intermediate reasoning steps—has proven highly effective (Lightman et al., 2024; Luo et al., 2024; Wang et al., 2024), with process reward models (PRMs) underpinning state-of-the-art reasoning systems. Building on this success, Lin et al. (2025a) combined stepwise binary feedback with KTO to jointly optimize intermediate and final-answer quality for mathematical problem solving. Transferring this idea to code generation, however, remains an open challenge: code does not naturally decompose into a single standardized sequence of intermediate logical steps, making the very definition of a "step" ambiguous.

We address this ambiguity by defining steps as module-level functions in decomposed programs, and propose **STEP-KTODER**, a framework that brings function-level process supervision to code generation. Our key idea is that decomposed programs provide an executable step structure: when a solution is written as multiple functions, each can be tested independently using automatically generated unit tests, yielding binary labels of local correctness. STEP-KTODER uses these labels to extend outcome-level KTO with function-level supervision, guiding training by both global pro-

†Corresponding author.

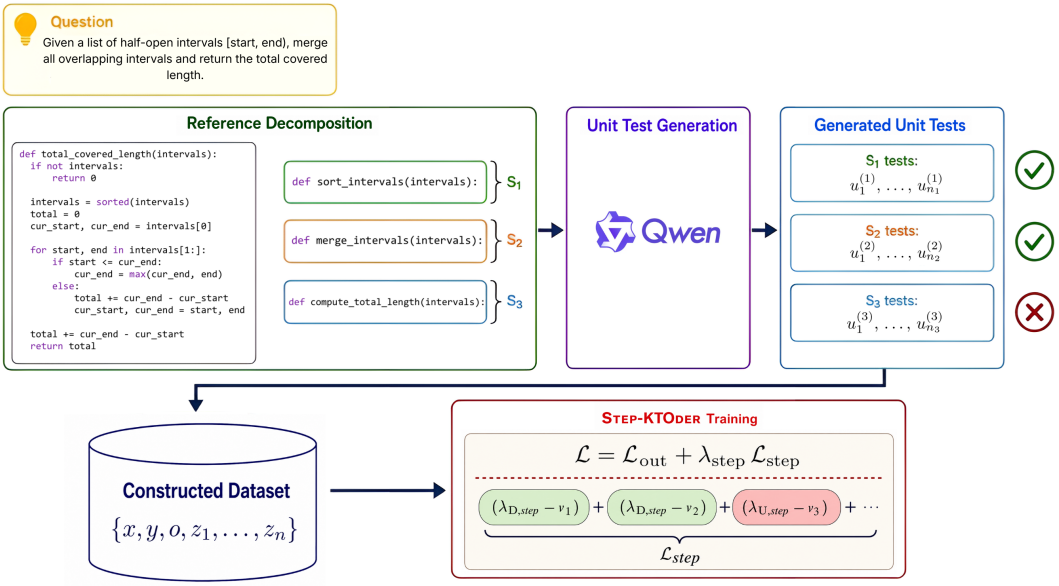


Figure 1: Overview of the STEP-KTODER framework. A reference solution $y^\star$ is decomposed into module-level functions $s_1, \ldots, s_n$, and function-level unit tests are generated for each function. Candidate completions are evaluated against these tests to obtain step labels z_i indicating local correctness, alongside an outcome label o from the dataset-provided test suite. The resulting labeled data is used to train the model with a joint objective that combines outcome-level KTO ($\mathcal{L}_{\text{out}}$) and function-level supervision ($\mathcal{L}_{\text{step}}$), reinforcing locally correct functions while penalizing locally incorrect ones.

gram success and local correctness, as indicated in Figure 1.

Beyond the main empirical gains, we investigate two design choices that are important to STEP-KTODER's effectiveness. First, programs that pass dataset-provided tests but contain a locally incorrect function provide a form of supervision that outcome-only training cannot express. We find that preserving these local/global mismatches strengthens the benefit of step-level supervision, whereas masking such mismatches substantially weakens the signal. Second, we assess whether automatically generated unit tests are necessary for reliable step supervision by replacing execution-based step labels with LLM-as-a-judge annotations. Although these annotations show moderate agreement with execution-based labels, they systematically over-predict step failures, and when used for STEP-KTODER training, they degrade performance. Even for already instruction-tuned Qwen2.5-Coder models, STEP-KTODER achieves substantial improvements: it improves over the base model by up to +26.7% on BigCodeBench Hard (Zhuo et al., 2025) and +27.0% on LiveCodeBench (Jain et al., 2025).

Our contributions are as follows:

- We propose STEP-KTODER, a code-specific instantiation of stepwise KTO that combines function-level process supervision with outcome-level feedback.
- We introduce an automatic data construction pipeline that decomposes solutions into functions, generates function-level unit tests, and derives execution-based step labels for training.
- Experiments across widely used benchmarks show that STEP-KTODER improves post-trained code models. We further demonstrate that replacing execution-based step labels with LLM-as-a-judge annotations weakens the step-level signal and degrades performance.

2 Related Work

2.1 Process Supervision for Reasoning

Process supervision has been highly effective for mathematical reasoning, where step-level feedback improves over outcome-only supervision (Cobbe et al., 2021; Uesato et al., 2022; Lightman et al., 2024). Subsequent work has scaled this supervision: Wang et al. (2024) introduced automatic step-level annotation via Monte Carlo estimation, and Luo et al. (2024) proposed a divide-and-conquer Monte Carlo Tree Search framework for large-scale process supervision data. Lin et al. (2025a) in-

troduced Step-KTO, which combines KTO with PRM-labeled stepwise feedback to jointly optimize intermediate reasoning quality and final response correctness. Our work builds on this idea but reformulates it for code: instead of using PRM-labeled reasoning steps, we define steps as concrete, independently testable functions and supervise them directly through execution-based unit test feedback. This perspective is consistent with recent work on code PRMs showing that execution feedback improves the reliability of process supervision for code (Li et al., 2025b). In contrast to PRM training, we study how execution-based process labels can be incorporated directly into offline preference optimization.

2.2 Preference Optimization for Code

Preference optimization has become a standard post-training technique for code models. DPO (Rafailov et al., 2023) learns from paired preferences, while KTO (Ethayarajh et al., 2024) removes the requirement for paired data using binary desirability labels grounded in prospect theory. Recent work has explored localized and fine-grained feedback for code: Zhang et al. (2025b) concentrated the DPO loss on error-prone token spans, while Wu et al. (2025) introduced Target-DPO, a focal preference alignment framework that localizes preference updates to targeted code regions. Dai et al. (2025) trained a PRM that provides dense line-level feedback during code generation via reinforcement learning. Zhang et al. (2025a) introduced CodeDPO, which constructs self-generated preference pairs to align code models for both correctness and efficiency.

These approaches target different granularities—tokens, code blocks, lines, or whole programs—but to our knowledge, none of these approaches defines steps as semantically meaningful, independently testable functions and integrates them into a binary preference optimization objective.

2.3 Solution Decomposition and Automated Testing

Fill-in-the-middle objectives (Bavarian et al., 2022) train models to complete code given surrounding context, encouraging modular structure. Ren et al. (2025) extend this idea to alignment, splitting code into AST-based blocks to construct more diverse preference pairs for DPO. Prompting-based approaches have also emphasized modular decomposition for code generation: Pan and Zhang (2025) break complex programming problems into smaller reasoning modules through hierarchical prompting. Lin et al. (2025b) proposed generating both code and unit tests simultaneously, then using self-validation to enhance generation quality. Automated unit test generation has matured significantly (Chen et al., 2023; Ma et al., 2025); our pipeline leverages this capability in a function-level setting, generating tests for individual functions and using execution outcomes as binary step labels for training.

2.4 Execution-Grounded Inference-Time Methods

Recent execution-grounded methods instead improve code at inference time. S* combines parallel sampling with iterative debugging and selects among candidates using adaptively synthesized inputs and their execution results (Li et al., 2025a), while ORPS explores a tree of reasoning and code trajectories guided by execution outcomes and self-critique (Yu et al., 2025). Unlike these per-query search and selection methods, STEP-KTODER uses function-level execution during offline data construction to produce labels that update the model policy.

3 Method

3.1 Problem Formulation

Given a prompt x and a reference solution $y^\star$, we first rewrite $y^\star$ into a decomposed multi-function program

$$\tilde{y} = (s_1, s_2, \ldots, s_n),$$

using a strong code language model, where each s_i denotes one function in the decomposed program and n is the total number of such functions. We treat each function s_i as a *step*.

Training samples are then generated from this decomposition: the target model fills the function skeleton defined by $\tilde{y}$ to produce a candidate solution y. Each training sample receives two forms of binary supervision:

- An **outcome label** $o \in \{0, 1\}$, indicating whether y passes the dataset-provided test suite for x.
- A sequence of **step labels** $\mathbf{z} = (z_1, \ldots, z_n)$, where each $z_i \in \{0, 1, \varnothing\}$ indicates whether the implementation of s_i in y passes its function-level unit tests. A label of $\varnothing$ indicates that no valid unit test is available for that step.

This formulation distinguishes *local correctness* from *global correctness*, providing finer-grained supervision than outcome-only evaluation. Rather than treating partially correct programs as uniformly desirable or undesirable, it enables the objective to reinforce correct functions while penalizing those responsible for failure.

3.2 KTO Background

We build on Kahneman–Tversky Optimization (KTO) (Ethayarajh et al., 2024), which aligns a policy π_θ from binary feedback using a Kahneman–Tversky-inspired value function over the log-ratio between the policy and a frozen reference π_{ref}:

$$r_\theta(x, y) = \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)}.$$

The outcome-level reference point is

$$z_0^{\text{out}} = \text{KL}\big(\pi_\theta(y' \mid x) \,\big\|\, \pi_{\text{ref}}(y' \mid x)\big)\,,$$

where y' denotes an output sequence used to estimate the divergence between the policy and reference. Given a binary desirability label $o \in \{0, 1\}$, the outcome-level value function is

$$v_{\text{out}} = \begin{cases} \lambda_D\, \sigma\big(\beta_{\text{out}}(r_\theta - z_0^{\text{out}})\big), & o = 1, \\ \lambda_U\, \sigma\big(\beta_{\text{out}}(z_0^{\text{out}} - r_\theta)\big), & o = 0, \end{cases}$$

where $r_\theta \equiv r_\theta(x, y)$ and $\sigma(\cdot)$ denotes the sigmoid function. $\beta_{\text{out}} > 0$ controls the sensitivity of the outcome-level value function, and λ_D and λ_U weight desirable and undesirable samples, respectively. The outcome-level KTO loss is then

$$\mathcal{L}_{\text{out}}(\pi_\theta, \pi_{\text{ref}}) = \mathbb{E}_{(x,y,o)\sim\mathcal{D}}\big[\lambda_o - v_{\text{out}}\big],$$

where $\lambda_o = \lambda_D$ if $o = 1$ and $\lambda_o = \lambda_U$ if $o = 0$.

3.3 Step-KTO Objective

Following Lin et al. (2025a), we instantiate stepwise KTO over the function-level steps defined in Section 3.1 and introduce reliability masking for functions without validated local tests. For each step s_i, we compute a step-local log-ratio over the tokens of that function:

$$r_i = \sum_{t \in s_i} \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}.$$

Each step has a label z_i as defined above and a mask $m_i \in \{0, 1\}$; only steps with $m_i = 1$ and $z_i \in \{0, 1\}$ contribute to the step-level loss.

Similarly, we define a step-level reference point

$$z_0^{\text{step}} = \text{KL}\big(\pi_\theta(y_i' \mid x, s_{<i}) \,\big\|\, \pi_{\text{ref}}(y_i' \mid x, s_{<i})\big)\,,$$

where y_i' denotes the subsequence of generated tokens belonging to the i-th function. The step-level value function is

$$v_i = \begin{cases} \lambda_{D,\text{step}}\, \sigma\big(\beta_{\text{step}}(r_i - z_0^{\text{step}})\big), & z_i = 1, \\ \lambda_{U,\text{step}}\, \sigma\big(\beta_{\text{step}}(z_0^{\text{step}} - r_i)\big), & z_i = 0. \end{cases}$$

Here, β_{step}, $\lambda_{D,\text{step}}$, and $\lambda_{U,\text{step}}$ are the step-level counterparts of the outcome-level KTO parameters.

Let $\mathcal{M}_\tau = \{i : m_i = 1,\ z_i \in \{0, 1\}\}$ denote the supervised steps for sample τ, and let $M = |\mathcal{M}_\tau|$. Let $\lambda_{z_i,\text{step}}$ denote the step-level target coefficient, with $\lambda_{z_i,\text{step}} = \lambda_{D,\text{step}}$ when $z_i = 1$ and $\lambda_{z_i,\text{step}} = \lambda_{U,\text{step}}$ when $z_i = 0$. The masked step-level loss is

$$\mathcal{L}_{\text{step}} = \mathbb{E}_{\tau\sim\mathcal{D}}\left[\frac{1}{n}\sum_{i\in\mathcal{M}_\tau}(\lambda_{z_i,\text{step}} - v_i)\right],$$

where the sum is zero when $M = 0$, so samples without supervised steps fall back to outcome-only KTO.

Combining with the outcome-level loss yields the final STEP-KTODER objective:

$$\mathcal{L}_{\text{STEP-KTODER}} = \mathcal{L}_{\text{out}} + \lambda_{\text{step}}\, \mathcal{L}_{\text{step}}.$$

By jointly optimizing outcome-level and function-level feedback, STEP-KTODER reinforces locally correct functions and penalizes locally incorrect ones, providing a more localized learning signal than outcome-level supervision alone. Figure 2 illustrates this difference on a failing candidate: outcome-only KTO treats the full program as undesirable, whereas STEP-KTODER assigns negative step-level signal only to the faulty function.

3.4 Data Construction Pipeline

Our pipeline consists of four stages, illustrated in Figure 1.

(1) Reference decomposition. We use a strong instruction-tuned code model to rewrite each ground-truth solution into an interface-preserving multi-function program, retaining only rewrites that pass the original test suite.

(2) Function-level unit test generation. For each decomposed function, a strong code model generates several targeted unit tests for its interface and edge cases. We retain only tests that pass

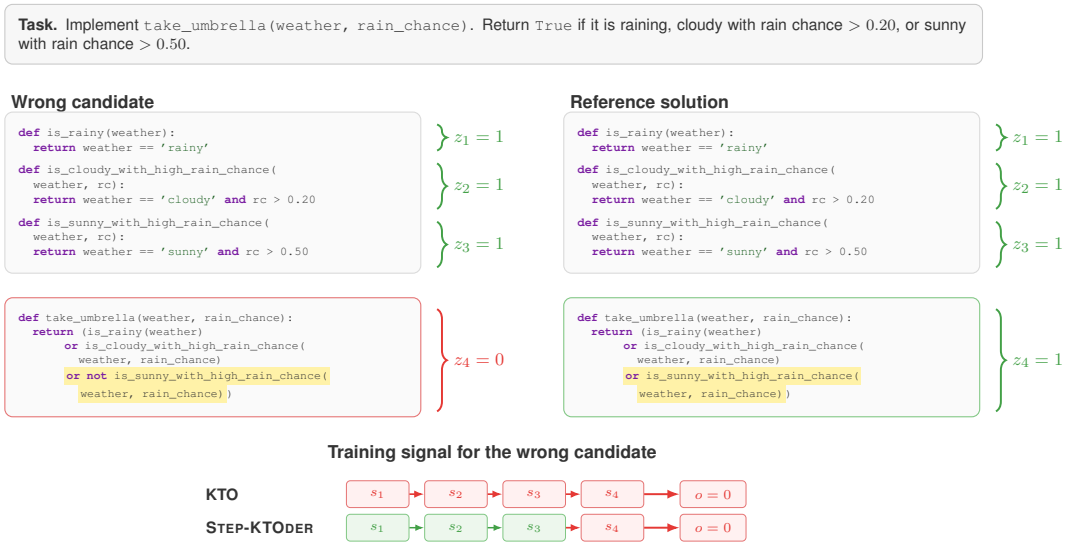


Figure 2: Training signal comparison for a failing candidate. The wrong candidate differs from the decomposed reference solution only in the highlighted line, yielding locally correct functions $z_1, z_2, z_3 = 1$, a faulty final function $z_4 = 0$, and outcome label $o = 0$. Outcome-only KTO penalizes all function spans, whereas STEP-KTODER preserves the locally correct functions and penalizes only the faulty one.

on the reference implementation, invoke the target function, and contain nontrivial assertions whose pass condition depends on the function's output. Functions without valid tests receive a null step label ($z_i = \varnothing$) and are masked from the step loss. The prompts are provided in Appendix O.

(3) On-policy candidate generation. From each decomposition, we construct a *skeleton* by replacing function bodies with `pass` while preserving signatures, docstrings, imports, and module-level context. Then, for each skeleton we sample $k = 8$ candidate completions from the model being post-trained, using temperature $T = 0.4$ and top-p sampling with $p = 0.95$; this keeps the preference data close to the model's own generation distribution.

(4) Labeling and dataset construction. Each candidate is evaluated against the dataset-provided test suite to obtain an outcome label o. Step labels z_i are obtained by executing each candidate function against its generated unit tests. By default, we use *local* step labels: each function is tested after being inserted into the decomposed reference program, with all other functions kept fixed to their reference implementations. This isolates function-level correctness from downstream composition effects. Across datasets, 83.2% of testable decomposed functions retain at least one validated test. We refer to this quantity as *validated-test availability*; the corresponding per-problem distribution is shown in Figure 3. For each task, we retain up to one passing and one failing candidate, preferring candidates with richer step-level supervision. Rows with fewer than two supervised steps fall back to outcome-only KTO, avoiding single-step supervision that largely duplicates the outcome-level signal.

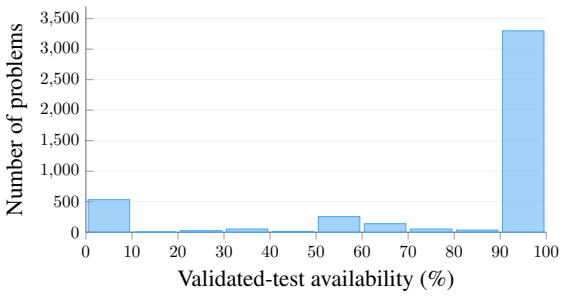


Figure 3: Distribution of per-problem validated-test availability. Most problems achieve near-full test availability.

3.5 Conflict-Preserving Step Labels

A key design question is how to handle examples where outcome-level and step-level supervision disagree. Two conflict patterns arise: **passing programs with negative step labels**, where the full program passes dataset tests but at least one supervised function fails its function-level tests; and **failing programs with all-positive step labels**, where the full program fails dataset tests even though all

supervised functions pass their unit tests. These examples reflect a mismatch between local and end-to-end correctness. Rather than removing them, we compare masked and conflict-preserving variants to test whether such cases provide useful training signal beyond outcome-only supervision and ordinary partial-correctness patterns.

4 Experimental Setup

4.1 Training Configuration

All target models are instruction-tuned code models that have already undergone substantial post-training. We fine-tune each model for 1 epoch using LoRA (Hu et al., 2022) with global batch size 16. KTO and STEP-KTODER use a learning rate of 1×10^{-6}, while DPO uses 5×10^{-7}. We set $\lambda_{\text{step}} = 1.0$ by default. Full training details are provided in Appendix B.

4.2 Datasets

Training data is drawn from the training splits of TACO (Li et al., 2023) and APPS (Hendrycks et al., 2021). We construct training samples using the pipeline described in Section 3: reference solutions are decomposed with Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), and each target model generates on-policy candidate completions from the resulting function skeletons. For each problem, we retain up to one passing and one failing candidate using a ranking procedure that prioritizes positives whose supervised functions all pass their unit tests and negatives with both passing and failing function-level labels. Such negatives localize which components remain correct and which fail (Appendix A).

Table 1 summarizes the resulting training sets. For same-family training with Qwen2.5-Coder-1.5B-Instruct and Qwen2.5-Coder-3B-Instruct, we include decomposed reference solutions as additional positive anchors. For cross-family training with DeepSeek-Coder-6.7B-Instruct, we use only on-policy generated candidates, since adding off-policy samples degrades performance (Appendix M). Rows with fewer than two supervised steps fall back to outcome-only KTO. As a result, 82% of Qwen2.5-Coder-1.5B-Instruct rows, 83% of Qwen2.5-Coder-3B-Instruct rows, and 78% of DeepSeek-Coder-6.7B-Instruct rows carry active step-level supervision. We justify this threshold in Section 6.1.

	Qwen-1.5B	Qwen-3B	DeepSeek
Dataset composition			
Total rows	11,397	11,440	6,573
GT rows	5,018	5,018	0
Generated rows	6,379	6,422	6,573
Step supervision			
Step-supervised	9,376	9,455	5,157
Fallback KTO	2,021	1,985	1,416
Avg. steps / active row	2.66	2.66	2.68
Conflict analysis			
Conflicts (o=1, z_i=0)	1,194	1,397	1,428
Structure quality			
AST alignment	84.2%	84.7%	80.3%

Table 1: Training dataset statistics. Step-supervised rows contain at least two supervised function-level labels and activate the step-level loss; fallback KTO rows use outcome-only KTO. Avg. steps / active row is computed over step-supervised rows. AST alignment measures exact agreement with the decomposition skeleton.

4.3 Evaluation Benchmarks and Baselines

We evaluate on seven code generation benchmarks: HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al., 2023), MBPP (Austin et al., 2021) and MBPP+ (Liu et al., 2023), BigCodeBench Full and Hard (Zhuo et al., 2025), and LiveCodeBench (Jain et al., 2025) release `v4_v5`. All evaluations use greedy decoding with vLLM (Kwon et al., 2023).

We compare STEP-KTODER against four baselines: the original instruction-tuned model, DPO (Rafailov et al., 2023), KTO (Ethayarajh et al., 2024), and Target-DPO (Wu et al., 2025). Baseline construction details are provided in Appendix D.

5 Results

5.1 Main Results

The experimental results are reported in Table 2. Since all target models are already instruction-tuned code models, easier benchmarks such as HumanEval and MBPP leave limited headroom, and the clearest gains appear on harder benchmarks. On Qwen2.5-Coder-1.5B-Instruct, STEP-KTODER improves over KTO by +11.5% on BigCodeBench Hard and +7.7% on LiveCodeBench. On Qwen2.5-Coder-3B-Instruct, the gains are similarly concentrated on the hardest benchmarks, with +11.3% on BigCodeBench Hard and +9.5% on LiveCodeBench, while matching or slightly improving KTO elsewhere. DeepSeek-Coder-6.7B-Instruct shows the same qualitative pattern: BigCodeBench Hard improves monotonically from

Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
Qwen2.5-Coder-1.5B-Instruct	0.689	0.604	**0.664**	0.556	0.240	0.054	0.052
w/ DPO	0.695	0.610	**0.664**	**0.563**	0.245	0.054	0.056
w/ Target-DPO	0.683	**0.628**	**0.664**	0.558	0.245	**0.068**	0.019
w/ KTO	**0.701**	0.610	0.648	0.542	0.245	0.061	0.052
w/ STEP-KTODER	**0.701**	0.616	0.656	0.548	**0.248**	**0.068**	**0.056**
Relative improvement over KTO	*0.0%*	*+1.0%*	*+1.2%*	*+1.1%*	*+1.2%*	*+11.5%*	*+7.7%*
Qwen2.5-Coder-3B-Instruct	0.860	0.799	0.741	0.619	0.356	0.101	0.100
w/ DPO	0.854	0.811	0.741	**0.630**	0.360	0.115	0.112
w/ Target-DPO	0.854	0.805	**0.746**	**0.630**	0.351	0.108	0.112
w/ KTO	**0.878**	**0.835**	0.741	0.624	**0.367**	0.115	0.116
w/ STEP-KTODER	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**
Relative improvement over KTO	*0.0%*	*0.0%*	*+0.7%*	*+1.0%*	*0.0%*	*+11.3%*	*+9.5%*
DeepSeek-Coder-6.7B-Instruct	**0.799**	0.713	0.746	0.640	0.342	0.108	0.127
w/ DPO	**0.799**	0.726	0.749	0.643	**0.347**	0.115	0.127
w/ Target-DPO	**0.799**	**0.738**	0.725	0.624	0.346	0.108	0.112
w/ KTO	0.793	0.726	0.749	0.640	**0.347**	0.122	0.127
w/ STEP-KTODER	**0.799**	0.732	**0.754**	**0.648**	**0.347**	**0.128**	**0.131**
Relative improvement over KTO	*+0.8%*	*+0.8%*	*+0.7%*	*+1.2%*	*0.0%*	*+4.9%*	*+3.2%*

Table 2: Pass rate across model families and scales. Relative improvements are computed over outcome-only KTO within each model; gains are largest on harder benchmarks. The best results are highlighted in bold.

DPO to KTO to STEP-KTODER, with STEP-KTODER achieving a +4.9% gain over KTO, and LiveCodeBench improving by +3.2%. On the easier benchmarks, where headroom is limited, STEP-KTODER remains competitive, obtaining the best or tied-best result in most cases, with Target-DPO leading slightly on HumanEval+ for DeepSeek-Coder-6.7B-Instruct. The gains persist across seeds: STEP-KTODER exceeds KTO on both BigCodeBench Hard and LiveCodeBench across three seeds (Appendix E).

Both Target-DPO and STEP-KTODER target localized code failures, but they optimize different supervision signals: Target-DPO derives preference pairs from debugging traces and localizes loss to changed token regions, whereas STEP-KTODER labels functions directly via unit tests. STEP-KTODER is consistently stronger on the harder benchmarks. For Qwen2.5-Coder-1.5B-Instruct, Target-DPO is competitive on easier benchmarks and ties on BigCodeBench Hard, but STEP-KTODER is stronger on LiveCodeBench. For Qwen2.5-Coder-3B-Instruct and DeepSeek-Coder-6.7B-Instruct, STEP-KTODER improves over Target-DPO on both BigCodeBench Hard (+18.5% for both) and LiveCodeBench (+13.4% and +17.0%, respectively). On easier benchmarks, STEP-KTODER ties or improves over Target-DPO in most cases. Notably, STEP-KTODER uses at most 11,440 training samples, compared with Target-DPO's 59,000 preference pairs.

5.2 Ablations

Table 3 reports three core ablations on Qwen2.5-Coder-3B-Instruct. *First*, conflict preservation is essential: masking conflicts reduces STEP-KTODER to KTO-level performance on BigCodeBench Hard and decreases on LiveCodeBench as well. This shows that apparent disagreements between outcome-level and function-level labels can provide useful training signal rather than noise to be removed. *Second*, $\lambda_{\text{step}} = 1.0$ gives the best trade-off ($\lambda_{\text{step}} = 0$ corresponds to outcome-only KTO), reported above. Lower step weights underuse the function-level signal, while larger weights begin to regress easier benchmarks without improving the hardest ones. *Third*, requiring at least two supervised steps outperforms the looser threshold of one, consistent with the redundancy of single-step supervision discussed in Section 6.1.

6 Analysis

6.1 Why Function-Level Supervision Helps

Both DPO and KTO operate at the program level, weighting the policy–reference log-ratio by a single binary outcome label. They therefore cannot distinguish which parts of a generated program contributed to success or failure: when correct and incorrect functions coexist, outcome-level methods reinforce or penalize all components together. STEP-KTODER targets this missing information by assigning labels to individual functions, allow-

Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
Conflict-handling policy							
KTO (no step loss)	**0.878**	**0.835**	0.741	0.624	**0.367**	0.115	0.116
STEP-KTODER + mask conflicts	**0.878**	**0.835**	0.741	0.624	**0.367**	0.115	0.119
STEP-KTODER + keep conflicts (default)	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**
Step loss weight λ_{step}							
0.5	0.872	0.829	**0.746**	0.624	0.364	0.122	**0.127**
1.0 (default)	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**
2.0	0.872	0.829	**0.746**	0.622	0.363	0.122	**0.127**
Min. supervised steps threshold							
1	**0.878**	**0.835**	0.741	0.627	0.363	0.115	0.116
2 (default)	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**

Table 3: Core ablations on Qwen2.5-Coder-3B-Instruct. We vary conflict handling, the step-loss weight λ_{step}, and the minimum number of supervised steps required to activate the step loss.

ing the model to reinforce locally correct functions and penalize locally incorrect ones within the same program. This explains why conflict preservation matters. When $z_i = o$ for all supervised steps, the step loss is directionally aligned with the outcome loss and mainly changes where the gradient is applied. In contrast, when $z_i \neq o$, the step loss provides a direction that outcome-level supervision cannot produce, such as penalizing a specific function even when the full program passes. Such conflicts can arise when the decomposed function interface exposes edge cases not exercised by the original test suite. Masking these conflicts removes this directional disagreement, making the remaining step signal largely redundant with outcome-level supervision. The same reasoning explains the minimum-supervised-steps threshold. For single-function rows, the step span covers the full answer, so $r_1 = r_{\text{out}}$. When $z_1 = o$, the step loss adds no information beyond a rescaled outcome loss. Requiring at least two supervised steps filters out these redundant rows.

6.2 Function-Level Labels Localize Failures

To test whether negative step labels identify functions that contribute to end-to-end failure, we perform a repair intervention on failing candidates whose supervised functions include both positive and negative labels. We replace the locally negative functions with their decomposed reference implementations and evaluate the repaired candidates using the original dataset-provided test suite. As controls, we replace either the same number of locally positive functions or the same number of randomly selected observed functions.

As shown in Table 4, replacing locally negative functions repairs 74.9% of failing candidates, substantially outperforming both random and locally positive replacement. This suggests that execution-based step labels localize repair-relevant faults, rather than merely correlating with outcome correctness.

Replacement target	Repair rate
Local-negative ($z_i = 0$)	**74.9%**
Random observed	32.9%
Local-positive ($z_i = 1$)	1.2%

Table 4: Repair intervention on failing candidates.

6.3 Execution-Based Labels Are Essential

A natural question is whether the execution-based unit tests at the core of STEP-KTODER are necessary, or whether an LLM-as-a-judge can provide comparable step-level supervision. We test this by replacing execution-based step labels with judgments from GPT-5.4 mini (OpenAI, 2026), and we retrain Qwen2.5-Coder-3B-Instruct with identical settings. The LLM shows moderate overall agreement with execution-based labels (73.2%), but this aggregate score hides a strong asymmetry. It agrees with execution on 94.1% of failing functions, but on only 52.2% of passing functions, systematically over-predicting failure. This bias corrupts the positive step labels needed for STEP-KTODER: training with LLM-as-a-judge labels instead of execution-based labels degrades performance, especially on BigCodeBench Hard and LiveCodeBench. Thus, execution-based labels are essential not only because they provide step-level supervision: they also provide reliable step-level supervision. Full agreement statistics

and benchmark results across all seven benchmarks are provided in Appendix N.

7 Conclusion

We propose STEP-KTODER, a framework for code preference optimization that defines process-supervision steps as module-level functions in decomposed programs. By combining outcome-level KTO with execution-based function labels, STEP-KTODER provides localized feedback that reinforces correct functions while penalizing incorrect ones. Experiments on post-trained code models show consistent gains over outcome-only KTO and DPO, with ablations confirming that preserving local/global label conflicts and using execution-based labels are central to these gains. These results suggest that function-level execution feedback offers a practical path toward process supervision for code generation.

8 Limitations

Model and data scope. STEP-KTODER is most natural when solutions admit a meaningful decomposition into independently testable functions, and is most reliable when candidate solutions are generated on-policy by the target model, which adds engineering cost for each new model. Our experiments focus on instruction-tuned code models that have already undergone substantial post-training; applying function-level supervision earlier, during SFT or on larger base-model training runs, remains a promising direction.

Difficulty distribution. Our filtering pipeline requires problems to have a clear function-style structure or to admit a meaningful decomposition. As a result, the training data is biased toward problems whose solutions can be decomposed into testable functions, while harder competition problems are less represented. Extending the pipeline to more complex I/O formats and harder problem regimes is a natural next step.

Unit-test quality. Step labels depend on automatically generated unit tests, which we validate by execution against decomposed reference solutions but do not formally verify. A mutation-sensitivity audit (Appendix H) shows that the retained unit tests reject 92.6% of semantically perturbed reference implementations, and our repair intervention (Section 6.2) shows that negative step labels identify functions responsible for end-to-end failure. Our LLM-as-a-judge comparison further shows that execution-based labels are substantially more reliable than LLM judgments. Still, stronger automated test-generation or verification methods could further improve label quality.

Ethical Considerations

The models used in this paper, Qwen2.5-Coder (Hui et al., 2024) and DeepSeek-Coder (Guo et al., 2024), are licensed for academic research purposes. The training datasets, TACO (Li et al., 2023) and APPS (Hendrycks et al., 2021), and all evaluation benchmarks (Chen et al., 2021; Liu et al., 2023; Austin et al., 2021; Zhuo et al., 2025; Jain et al., 2025) are publicly available and distributed for research use.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (MSIT) (RS-2024-00451435, 20%; RS-2024-00413957, 20%), the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the MSIT (RS-2025-02305453, 15%; RS-2025-02273157, 15%; RS-2025-25442149, 15%; RS-2021-II211343, 15%), the Institute of New Media and Communications (INMAC), the BK21 FOUR program funded by the Ministry of Education, the Artificial Intelligence Graduate School Program (Seoul National University), and the Research Program for Future ICT Pioneers at Seoul National University in 2026.

References

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. *Preprint*, arXiv:2108.07732.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *Preprint*, arXiv:2207.14255.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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. *Preprint*, arXiv:2107.03374.

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

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2025. Process supervision-guided policy optimization for code generation. *Preprint*, arXiv:2410.17621.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Model alignment as prospect theoretic optimization. In *Forty-first International Conference on Machine Learning*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

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 *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

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

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. 2025a. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15964–15978, Suzhou, China. Association for Computational Linguistics.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025b. CodePRM: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182, Vienna, Austria. Association for Computational Linguistics.

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. *Preprint*, arXiv:2312.14852.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Yen-Ting Lin, Di Jin, Tengyu Xu, Tianhao Wu, Sainbayar Sukhbaatar, Chen Zhu, Yun He, Yun-Nung Chen, Jason E Weston, Yuandong Tian, Arash Rahnama, Sinong Wang, Hao Ma, and Han Fang. 2025a. Step-KTO: Optimizing mathematical reasoning through stepwise binary feedback. In *Proceedings of The 3rd Workshop on Mathematical Natural Language Processing (MathNLP 2025)*, pages 15–33, Suzhou, China. Association for Computational Linguistics.

Zi Lin, Sheng Shen, Jingbo Shang, Jason E Weston, and Yixin Nie. 2025b. Learning to solve and verify: A self-play framework for mutually improving code and

test generation. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. 2024. Improve mathematical reasoning in language models by automated process supervision. *Preprint*, arXiv:2406.06592.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. 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.

OpenAI. 2026. Introducing GPT-5.4 mini and nano. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/. Accessed: 2026-04-28.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Gray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*.

Ruwei Pan and Hongyu Zhang. 2025. Modularization is better: Effective code generation with modular prompting. *Preprint*, arXiv:2503.12483.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Houxing Ren, Zimu Lu, Weikang Shi, Haotian Hou, Yunqiao Yang, Ke Wang, Aojun Zhou, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025. Alignment with fill-in-the-middle for enhancing code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8304–8320, Suzhou, China. Association for Computational Linguistics.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process- and outcome-based feedback. *Preprint*, arXiv:2211.14275.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, Bangkok, Thailand. Association for Computational Linguistics.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

Jie Wu, Haoling Li, Xin Zhang, Xiao Liu, Yangyu Huang, Jianwen Luo, Yizhen Zhang, Zuchao Li, Ruihang Chu, Yujiu Yang, and Scarlett Li. 2025. Teaching your models to understand code via focal preference alignment. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 14003–14023, Suzhou, China. Association for Computational Linguistics.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Xingru Jiang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. 2025. Reasoning through execution: Unifying process and outcome rewards for code generation. In *Forty-second International Conference on Machine Learning*.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025a. CodeDPO: Aligning code models with self generated and verified source code. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15854–15871, Vienna, Austria. Association for Computational Linguistics.

Kechi Zhang, Ge Li, Jia Li, Yihong Dong, Jia Li, and Zhi Jin. 2025b. Focused-DPO: Enhancing code generation through focused preference optimization on error-prone points. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 9578–9591, Vienna, Austria. Association for Computational Linguistics.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

A Preference Pair Selection

For each problem we generate 8 candidate completions from the target model and select up to one positive and one negative candidate for the training set, based on the outcome label o and the step labels.

Positive candidates. Among candidates with $o = 1$, we select the one with the cleanest step-level signal: we prioritize candidates whose supervised functions all pass their unit tests, followed by candidates with passing labels and some unknown steps, then candidates containing both passing and failing step labels, and finally candidates without active step supervision. Within each category, ties are broken by preferring candidates with more locally passing functions, fewer locally failing functions, valid parsing, and non-empty outputs.

Negative candidates. Among candidates with $o = 0$, we select the one with the richest local supervision. We first prioritize candidates that contain both passing and failing supervised functions, since they identify which components remain correct and which are responsible for the failure. If none are available, we fall back to candidates with at least one locally failing function. As a secondary fallback, we use *generated-context labels*, obtained by executing each generated function within the full generated module rather than inserting it into the decomposed reference program. We prioritize candidates with both passing and failing generated-context labels, then candidates with at least one generated-context failure, and finally candidates without informative step signal.

Reference-context and generated-context agreement. We compare our default local labels, computed in the reference context, with generated-context labels on selected training rows. They agree on 89.4% of $43{,}458$ comparable supervised steps, consistently across models (89.2–89.6%), suggesting that local labels closely align with generated-context behavior while isolating function-level correctness.

Pairing. Tasks with only a positive candidate contribute an unpaired positive row, and tasks with only a negative candidate contribute an unpaired negative row; both are usable by KTO but not DPO. Tasks with both contribute one positive and one negative row.

B Training Details

We fine-tune all target models with LoRA (Hu et al., 2022) using rank $r = 32$, scaling factor $\alpha = 64$, dropout 0.05, maximum sequence length 2048, and global batch size 16. We use AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$ and a cosine learning-rate schedule with 50 warmup steps and minimum ratio 0.1. We use learning rate 1×10^{-6} for KTO and STEP-KTODER, and 5×10^{-7} for DPO. For DPO and KTO, we set $\beta = 0.1$. For STEP-KTODER, we use $\beta_{\text{out}} = \beta_{\text{step}} = 0.1$, set $\lambda_{\text{step}} = 1.0$, and use unit weights for all KTO value-function coefficients: $\lambda_D = \lambda_U = \lambda_{D,\text{step}} = \lambda_{U,\text{step}} = 1.0$. Sensitivity to these coefficients is analyzed in Appendix K.

All experiments were run on 2 NVIDIA RTX A5000 GPUs with 24 GB of memory.

C Data Construction Cost

For Qwen2.5-Coder-3B-Instruct, data construction required approximately 78 GPU-hours: 24 GPU-hours for reference decomposition, 43 for function-level unit-test generation, and 11 for sampling $k = 8$ candidate completions. Decompositions, skeletons, and tests are generated once and reused across target models; only candidate sampling and CPU-based execution and labeling are model-specific.

D DPO and Baseline Construction Details

DPO (Rafailov et al., 2023) trains from paired preferences. Given a prompt x, a preferred solution y^+, and a dispreferred solution y^-, its loss is

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}\left[\log \sigma\left(\beta\left(r_\theta(x, y^+) - r_\theta(x, y^-)\right)\right)\right],$$

where

$$r_\theta(x, y) = \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)}.$$

Here, π_θ is the optimized policy, π_{ref} is the frozen reference policy, $\sigma(\cdot)$ is the sigmoid function, and β controls the KL regularization strength.

In our experiments, DPO pairs are constructed using the same candidate pool as KTO and STEP-KTODER: a passing candidate is used as y^+ and a failing candidate as y^-. Unlike DPO, KTO and STEP-KTODER operate on individually labeled candidates, allowing them to use unpaired positives or negatives. KTO samples are created using the same candidate rows and outcome labels as STEP-KTODER, but we discard all function-level

Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
KTO	$\mathbf{0.872}_{\pm 0.006}$	$0.827_{\pm 0.007}$	$\mathbf{0.743}_{\pm 0.003}$	$\mathbf{0.628}_{\pm 0.003}$	$0.364_{\pm 0.004}$	$0.113_{\pm 0.004}$	$0.109_{\pm 0.006}$
STEP-KTODER	$\mathbf{0.872}_{\pm 0.006}$	$\mathbf{0.829}_{\pm 0.006}$	$\mathbf{0.743}_{\pm 0.004}$	$\mathbf{0.628}_{\pm 0.003}$	$\mathbf{0.365}_{\pm 0.003}$	$\mathbf{0.128}_{\pm 0.000}$	$\mathbf{0.123}_{\pm 0.006}$

Table 5: Mean pass rate ± standard deviation across three training seeds. Bold denotes the best mean within each benchmark.

Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
Qwen2.5-7B-Instruct	0.811	0.768	0.839	0.704	0.366	0.135	0.134
w/ DPO	0.817	0.768	0.844	0.706	0.362	0.149	**0.142**
w/ Target-DPO	0.799	0.744	0.796	0.677	**0.376**	0.108	0.138
w/ KTO	0.823	0.780	**0.847**	0.709	0.361	0.155	0.138
w/ STEP-KTODER	**0.829**	**0.787**	0.844	**0.717**	0.362	**0.162**	**0.142**

Table 6: Pass rates for the general-purpose Qwen2.5-7B-Instruct model.

step labels, reducing training to outcome-only preference optimization.

We train Target-DPO (Wu et al., 2025) on its dataset of 59,000 preference pairs. For comparability, we match our DPO LoRA setup: $r = 32$, $\alpha = 64$, dropout 0.05, global batch size 16, and learning rate 5×10^{-7}.

E Multi-Seed Robustness

To assess seed sensitivity, we train Qwen2.5-Coder-3B-Instruct with both KTO and STEP-KTODER across three random seeds, holding all other settings fixed. Table 5 reports the resulting mean pass rate and standard deviation.

STEP-KTODER exceeds seed-matched KTO in all three runs on both BigCodeBench Hard and LiveCodeBench. The corresponding mean gains are 0.015 and 0.014, respectively; on the other five benchmarks, the difference between the method means is at most 0.002.

F General-Purpose Model

To test whether this pattern persists for a general-purpose instruction-tuned model, we additionally train Qwen2.5-7B-Instruct using on-policy candidates together with the validated ground-truth anchors used in our Qwen experiments. Qwen2.5-7B-Instruct starts from a stronger base than our other target models on MBPP, BigCodeBench, and LiveCodeBench. STEP-KTODER exceeds KTO on six of seven benchmarks, with the largest gains on BigCodeBench Hard and LiveCodeBench; see Table 6 for full results.

G Dataset Statistics

This section reports difficulty distributions across the three stages of our data construction pipeline—the raw TACO and APPS training splits, the problems retained after ground-truth decomposition validation, and the final per-source training rows used by each target model—together with quality statistics for the function-level unit tests generated in Stage 2.

Raw datasets. Table 7 reports the difficulty composition of the raw TACO and APPS training splits as released. Both datasets contain difficulty annotations covering a wide range from introductory exercises to competition-level problems. Our pipeline retains only problems with an explicit function-style entry point—identified by a non-empty `fn_name` field in the dataset's annotations—and discards problems with other interfaces, which typically contain monolithic algorithmic solutions resistant to function-level decomposition. As a result, higher-difficulty buckets contribute few or no problems to our training set: in TACO, only `easy`, `medium`, and `medium_hard` problems contribute, while `hard`, `very_hard`, and `unknown_difficulty` problems are filtered out entirely. In APPS, only `introductory` and `interview` problems remain; `competition` problems are excluded.

Ground-truth decomposition validation. Table 8 reports per-difficulty pass rates after Stage 1, where ground-truth solutions are decomposed with Qwen2.5-Coder-32B-Instruct and validated by execution against the original test suite. In

Source	Difficulty	# Problems	%
TACO	easy	8,904	35.0
	medium	3,244	12.7
	medium_hard	2,745	10.8
	hard	3,162	12.4
	very_hard	2,374	9.3
	unknown_difficulty	5,014	19.7
	Total	25,443	100.0
APPS	introductory	2,353	84.0
	interview	450	16.0
	Total	2,803	100.0

Table 7: Difficulty distribution of the raw TACO and APPS training splits before any filtering.

Source	Difficulty	Total	Validated	Pass rate
TACO	easy	4,112	2,141	52.1%
	medium	1,200	225	18.8%
	medium_hard	520	193	37.1%
	Total	5,832	2,559	43.9%
APPS	introductory	2,325	2,069	89.0%
	interview	442	390	88.2%
	Total	2,767	2,459	88.9%

Table 8: Ground-truth decomposition validation rates by difficulty.

the table, *Total* denotes the number of problems entering Stage 1 with a clear function-style entry point, and *Validated* denotes the number whose decomposed solution still passes the original tests. Only validated problems proceed to candidate generation. Validation is much more permissive on APPS (88.9%) than on TACO (43.9%). The TACO drop is concentrated in the harder buckets: `easy` validates at 52.1%, `medium_hard` at 37.1%, and `medium` at only 18.8%, reflecting the difficulty of producing semantically equivalent decompositions for problems with complex global state.

Final training sets. Table 9 reports the difficulty distribution of the rows actually used to train each target model. For Qwen2.5-Coder-3B-Instruct (same-family), we retain all 5,018 validated ground-truth decompositions as additional positive anchors and add 6,422 generated rows produced by the target model itself. For DeepSeek-Coder-6.7B-Instruct (cross-family), ground-truth rows are discarded (Appendix M) and only on-policy generated candidates are retained.

The final training sets are skewed toward easier problems, reflecting both the natural distribution of decomposition-friendly problems and the per-difficulty validation rates from Table 8. We discuss the implications in the *Difficulty distribution* paragraph of Section 8.

H Unit Test Generation Quality

Generation and validation protocol. Function-level unit tests are produced by Qwen2.5-Coder-32B-Instruct (Section 3.4, Stage 2) and validated by execution against the decomposed reference implementation. The generator outputs structured JSON test cases with named inputs, which are converted into a `unittest` class that calls the target function on each input and compares the result to the reference output. Tests are validated by execution: only test classes that parse, execute, and pass against the ground-truth code are retained. Functions for which no valid test remains receive a null step label ($z_i = \varnothing$) and fall back to outcome-only KTO supervision during training. Table 10 summarizes unit-test generation quality, and Figure 3 shows the full validated-test-availability distribution.

As shown in Table 10, first-pass parsing succeeds in over 98% of calls, and covered functions have around 5.5 valid tests on average, providing multiple independent assertions per supervised step. Overall, 83.2% of testable decomposed functions retain at least one validated test, while mean per-problem validated-test availability is 84.4%; the latter is visualized in Figure 3.

Mutation-sensitivity audit. As an additional sanity check, we evaluate whether the generated function-level tests detect controlled perturbations rather than merely executing successfully on the reference implementation. We audit all 11,705 functions from validated decomposed programs that retained at least one valid generated unit test across TACO and APPS, and apply syntax-preserving mutations to the reference function, including comparison flips, Boolean-operator flips, arithmetic-operator changes, constant perturbations, and default-return replacements. Across 27,958 valid mutants, the generated tests reject 25,880 mutants, yielding a mutation kill rate of 92.6%. This suggests that the retained tests reliably detect local behavioral changes rather than only validating executability on the reference solution.

I Representative Function-Level Label Cases

Incomplete local test. For a Pair of Shoes task (TACO 11007), the candidate checks whether the

Source	Difficulty	Qwen2.5-Coder family			DeepSeek-Coder
		Ground truth anchors	1.5B gen.	3B gen.	6.7B gen.
TACO	easy	2,141	2,671	2,695	2,748
	medium	225	271	278	301
	medium_hard	193	245	237	260
APPS	introductory	2,069	2,643	2,657	2,661
	interview	390	549	555	603
Total rows		**5,018**	**6,379**	**6,422**	**6,573**

Table 9: Difficulty distribution of the final training sets for each target model. Qwen2.5-Coder-1.5B-Instruct and Qwen2.5-Coder-3B-Instruct use the same validated decomposed reference solutions as positive anchors, but differ in their on-policy generated candidates. DeepSeek-Coder-6.7B-Instruct uses only on-policy generated candidates.

	TACO	APPS
Decomposition scale		
Validated problems	2,559	2,459
Total functions	7,199	6,926
Testable functions	7,144	6,920
Unit-test generation		
First-pass parse success	98.8%	99.6%
Testable functions w/ ≥ 1 valid test	82.0%	84.5%
Mean valid tests / covered function	5.46	5.59
Mean per-problem test availability	83.4%	85.4%
Excluded functions		
Skipped functions (I/O glue)	55	6

Table 10: Function-level unit test generation quality. Validated-test availability is over testable functions.

concatenated left and right-shoe sizes are unique, rather than comparing the two size multisets. Because the retained test for `pair_of_shoes` covers only the empty input, both local labels are positive although the full program is incorrect. This illustrates why STEP-KTODER retains the outcome-level term.

Useful local/global conflict. In a Task Scheduler problem (APPS 171), the candidate passes the dataset-provided test suite, but its decomposed step `calculate_min_intervals` omits `len(tasks)` from the maximum, understating the required number of intervals on inputs where the task count exceeds the frequency-based bound. Its negative local label exposes a hidden defect that outcome-only KTO cannot represent.

J Repair Intervention Details

Beyond the aggregate repair rates reported in Section 6.2, we also examine how the intervention behaves across failure types. The analysis is run on originally failing Qwen2.5-Coder-3B-Instruct selected training candidates whose supervised functions include both positive and negative step labels. For each candidate, we replace local-negative functions with their decomposed reference implementations and rerun the dataset-provided tests. As controls, we replace either the same number of local-positive functions or the same number of randomly selected observed functions.

The repair effect is strongest for wrong-answer failures: replacing local-negative functions repairs 78.6% of such candidates on the combined APPS and TACO subset. This suggests that local-negative labels are especially effective at identifying semantic errors, rather than merely capturing parsing or execution artifacts.

K Sensitivity to KTO Value-Function Weights

The default configuration uses unit weights for all KTO value-function coefficients ($\lambda_D = \lambda_U = \lambda_{D,\text{step}} = \lambda_{U,\text{step}} = 1.0$). The KTO authors recommend rebalancing λ_U when the positive/negative sample ratio is skewed; our Qwen2.5-Coder-3B-Instruct training set has a step-level positive/negative ratio of approximately 2.5:1, motivating a check on whether upweighting undesirable samples improves performance. When one coefficient is varied, all other value-function coefficients remain fixed at 1.0. Table 11 reports two rebalancing runs against the default.

Both deviations from uniform weights regress BigCodeBench Hard. The outcome-level $\lambda_U = 2.5$ run additionally collapses LiveCodeBench performance, suggesting that outcome-level rebalancing is unstable in this setting. One possible explanation is that the observed positive/negative ratio reflects the natural pass distribution of a strong instruction-tuned 3B model rather than simple dataset imbalance; reweighting failures too aggressively may

Configuration	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
$\lambda_U = \lambda_{U,\text{step}} = 1.0$ (default)	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**
$\lambda_{U,\text{step}} = 2.5$	**0.878**	**0.835**	**0.746**	0.624	0.365	0.122	**0.127**
$\lambda_U = 2.5$	**0.878**	**0.835**	**0.746**	0.624	0.365	0.122	0.049

Table 11: Effect of rebalancing outcome- and step-level λ_U on Qwen2.5-Coder-3B-Instruct.

Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
KTO only	**0.878**	**0.835**	0.741	0.624	**0.367**	0.115	0.116
Step only	0.866	0.823	0.741	**0.630**	0.360	0.100	0.112
STEP-KTODER (default)	**0.878**	**0.835**	**0.746**	**0.630**	**0.367**	**0.128**	**0.127**

Table 12: Step-only objective ablation on Qwen2.5-Coder-3B-Instruct.

over-allocate gradient budget toward avoiding failures at the cost of reinforcing successful behavior, hurting generalization to harder benchmarks.

L Additional Objective Ablations

Our default objective combines outcome-level and function-level supervision:

$$\mathcal{L}_{\text{STEP-KTODER}} = \mathcal{L}_{\text{out}} + \lambda_{\text{step}} \mathcal{L}_{\text{step}}.$$

To better understand the role of the outcome-level term, we evaluate a step-only variant on Qwen2.5-Coder-3B-Instruct. This variant removes the outcome-level KTO loss and optimizes only the stepwise loss, allowing us to assess how much signal is provided by local execution feedback alone.

As shown in Table 12, the step-only variant retains useful signal from function-level execution feedback, matching KTO on MBPP and matching the full STEP-KTODER objective on MBPP+. However, removing the outcome-level KTO term weakens performance on HumanEval, BigCodeBench, and LiveCodeBench. The STEP-KTODER objective performs best overall, with the clearest advantage on BigCodeBench Hard. These results support our formulation of STEP-KTODER as a joint objective: the step loss provides localized credit assignment, while the outcome-level term preserves the end-to-end correctness signal needed for robust program-level performance.

M Off-Policy Training Data in Cross-Family Training

To test whether on-policy candidate generation is necessary, we train DeepSeek-Coder-6.7B-Instruct on candidates generated by Qwen2.5-Coder-3B-Instruct on the same set of decomposed problems used in our main DeepSeek run. Both training sets share the same decomposed function skeletons and unit tests, generated by Qwen2.5-Coder-32B-Instruct; the only difference is the model that produced the candidate completions. The Qwen-generated candidates are fully off-policy with respect to the DeepSeek reference model.

As shown in Table 13, off-policy training degrades both methods on the hardest benchmark (BigCodeBench Hard: KTO 0.122 to 0.115, STEP-KTODER 0.128 to 0.122). The effect on LiveCodeBench is asymmetric: KTO regresses substantially (0.127 to 0.112) while STEP-KTODER regresses marginally (0.131 to 0.127), suggesting that function-level supervision is more robust to policy mismatch than outcome-only supervision. Easier benchmarks (HumanEval, HumanEval+) show small improvements under off-policy training, likely reflecting the broader problem distribution covered by the Qwen model's candidate pool.

We attribute the BigCodeBench Hard regression to noisier log-ratio rewards: when the candidate distribution diverges from the reference model's, the ratio $\log \pi_\theta / \pi_{\text{ref}}$ is poorly calibrated, weakening the KTO objective. The function-level signal partially compensates because it operates on shorter token spans, where calibration noise has less cumulative effect. We therefore use the on-policy configuration as our default setting.

N LLM-as-a-Judge vs. Execution-Based Step Labels

To evaluate execution-based step labeling against a common alternative, we compare STEP-KTODER's unit-test-based function labels with annotations from GPT-5.4 mini, applied to the

Training data	Method	HumanEval		MBPP		BigCodeBench		LiveCodeBench
		Base	Plus	Base	Plus	Full	Hard	
On-policy (default)	KTO	0.793	0.726	0.749	0.640	0.347	0.122	0.127
On-policy (default)	**STEP-KTODER**	0.799	0.732	**0.754**	0.648	**0.347**	**0.128**	**0.131**
Off-policy (Qwen 3B-generated)	KTO	**0.805**	**0.735**	0.749	0.640	0.345	0.115	0.112
Off-policy (Qwen 3B-generated)	STEP-KTODER	**0.805**	**0.735**	0.751	**0.648**	**0.347**	0.122	0.127

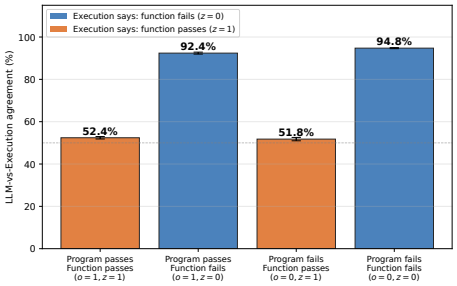
Table 13: Effect of training with off-policy data on DeepSeek-Coder-6.7B-Instruct.

same candidate functions in the Qwen2.5-Coder-3B-Instruct training pipeline. The comparison has two components: (i) label-level agreement statistics on the full candidate pool, and (ii) a downstream training experiment in which execution-based labels are replaced by LLM-as-a-judge labels in the actual STEP-KTODER training set.

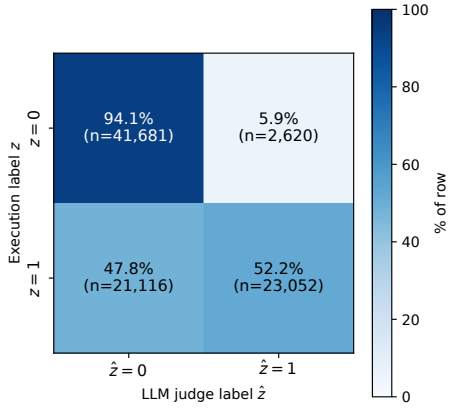
Setup. For agreement analysis, we sample N=88,469 supervised functions from the Qwen2.5-Coder-3B-Instruct candidate pool, stratified by the (o, z) outcome–step pair, and ask GPT-5.4 mini to predict whether each function would pass its associated unit tests. The judge sees the function source code and the unit tests, and returns a binary label with a one-sentence rationale; the system prompt is reproduced in Appendix O. For the downstream experiment, we apply the same procedure only to the supervised functions in the final training rows, replace execution-based labels with LLM-as-a-judge labels where available, and retrain with the same settings as the main result.

Agreement is asymmetric. Figure 4a reports agreement by (o, z) stratum, and Figure 4b shows the row-normalized confusion matrix. Overall agreement is 73.2%, but this aggregate score hides a strong asymmetry. The LLM agrees with execution on 94.1% of functions labeled as failing by execution (z=0), but on only 52.2% of functions labeled as passing by execution (z=1). At the stratum level, agreement is high on both failing-function strata (o=1, z=0: 92.4%; o=0, z=0: 94.8%), but only around 52% on both passing-function strata.

This pattern indicates that the LLM judge has a strong false-negative bias: it often flags true failures, but also frequently hallucinates issues in correct implementations. This preserves many negative conflict labels while corrupting the much larger pool of clean positive step labels.



(a) Agreement by (o, z) stratum.



(b) Row-normalized confusion matrix.

Figure 4: LLM-as-a-judge agreement with execution-based labels is high for failing functions (z=0) but much lower for passing functions (z=1), revealing a bias toward predicting failure.

Downstream impact. Table 14 compares STEP-KTODER trained with execution-based labels versus LLM-as-a-judge labels on Qwen2.5-Coder-3B-Instruct. Replacing execution labels with GPT-5.4 mini judgments substantially degrades the hardest benchmarks, with relative drops of 25.8% on BigCodeBench Hard and 11.8% on LiveCodeBench. Notably, BigCodeBench Hard falls below the outcome-only KTO baseline, confirming that moderate label-level agreement can still hide systematic bias that harms downstream training.

Discussion. Moderate aggregate agreement does not translate into useful supervision because the LLM judge's errors are highly asymmetric: its

Step labels	HumanEval		MBPP		BigCodeBench		LiveCodeBench
	Base	Plus	Base	Plus	Full	Hard	
KTO baseline	**0.878**	**0.835**	0.741	0.624	**0.367**	0.115	0.116
STEP-KTODER + execution (default)	**0.878**	**0.835**	**0.746**	0.630	**0.367**	**0.128**	**0.127**
STEP-KTODER + LLM-as-a-judge (GPT-5.4 mini)	0.860	0.817	**0.746**	**0.635**	0.354	0.095	0.112
Relative change vs. execution	*-2.1%*	*-2.2%*	*+0.0%*	*+0.8%*	*-3.5%*	*-25.8%*	*-11.8%*

Table 14: Effect of replacing execution-based step labels with GPT-5.4 mini judgments on Qwen2.5-Coder-3B-Instruct.

false-negative bias corrupts passing step labels, which STEP-KTODER needs to stabilize function-level training. Within the operating regime of our framework, execution-based labels are not merely a convenient choice but a necessary one.

O Prompts

Here, we outline the key prompts used in our STEP-KTODER data construction pipeline. Figure 5 shows the prompts used to decompose reference solutions into multi-function programs. Figure 6 gives the prompt used for generating function-level unit tests, and Figure 7 shows the prompt used to sample on-policy candidate completions from function skeletons. Finally, Figure 8 gives the prompt used by the LLM-as-a-judge in our labeling-method comparison (Appendix N).

Prompt for Function Decomposition

SYSTEM PROMPT

```
You are an expert Python refactoring assistant.
Your task is to decompose a Python solution into semantically meaningful top-level helper functions
while preserving exact behavior and the exact required interface.
When the solution contains distinct logical stages, decompose it into multiple helper functions
rather than leaving everything inside one large function.
Do not invent trivial helpers or change the algorithm unnecessarily. Follow the requested XML
schema exactly. Return ONLY valid XML. No explanations.
```

USER PROMPT

```
Refactor the following Python solution into structured XML with multiple <function> blocks.

Goal:
- Decompose into helper functions where useful.
- Preserve behavior exactly.
- Keep the required entrypoint unchanged (same name and signature).

Required entrypoint (must appear EXACTLY in your code):
{required_entrypoint_line}

Rules:
1. Do NOT introduce class Solution (this is a plain function problem).
2. Do NOT read from stdin or print output.
3. Do NOT include any top-level execution (no main(), no __starting_point(), no if __name__ ==
"__main__": ...).
4. Put helper functions ABOVE the required entrypoint.
5. Each function definition must be in its own <function> block.
6. The <code> section should contain exactly ONE def statement (or one class).
7. Copy all original imports at the very top of the FIRST <code> block.
8. Do NOT add unit tests yet: keep <tests> as placeholder comments.
9.  Do NOT define nested functions inside another function or method.  Every helper must be
top-level and placed in its own <function> block.
10.  When all <code> blocks are concatenated in order, the result must be a valid standalone
Python solution with exactly the same behavior as the original.
11. When the solution contains distinct logical stages, decompose it into multiple semantically
meaningful helper functions.
12. Avoid returning a single large function unless decomposition is genuinely unnecessary.
13. Do not invent trivial helpers just to increase the number of functions.

XML format:
<function name="..."> <docstring>...</docstring> <tests> {tests_placeholder} </tests> <code> ...
code here ... </code> </function>

Problem: {problem_description}

Original solution: {ground_truth_solution}

Refactored XML format (ONLY XML):
```

Figure 5: System and user prompts used to decompose reference solutions into behavior-preserving multi-function programs.

Prompt for Unit Test Generation

```
Output must be a JSON object with this exact schema:
{ "cases": [ {"name": "short_name", "args": [...], "kwargs": {...}}, ... ] }

Rules:
- args must be a JSON array; kwargs must be a JSON object (use {} if none).
- Use only JSON-serializable values: null, true/false, numbers, strings, lists, objects.
- Keep inputs physically SMALL (e.g., arrays under 5 items, integers between -50 and 50) to avoid
execution timeouts.
- The test suite MUST include at least one or two ADVERSARIAL EDGE CASES (e.g., empty lists [],
empty strings "", zero 0, or negative numbers -1).
- Do NOT make every case an edge case. Provide a balanced mix of typical and boundary inputs.
- Do NOT include expected outputs.
- Provide 4 to 6 diverse cases.
```

Figure 6: User prompt schema and constraints for generating function-level unit tests. The model returns only test inputs; expected outputs are derived by executing the reference implementation.

Prompt for Candidate Generation

```
Solve the following programming problem.

You MUST implement the provided skeleton by replacing each pass with a correct implementation.

Rules:
- Keep all function/class names and signatures unchanged.
- Do not remove any definitions.
- Do not add new functions, methods, or classes.
- Only fill the bodies of the provided definitions.
- Return a single complete Python module.

### problem
{question}

### skeleton
{skeleton_code}
```

Figure 7: User prompt for candidate generation. The target model fills the function skeleton (with `pass` as each function body) constructed in Stage 3 of our pipeline.

Prompt for LLM-as-a-Judge

```
You are an expert Python programmer acting as a code reviewer.  You will be given a Python
function and a set of unit tests for that function. Your job is to decide whether the function
implementation would pass all the provided unit tests when executed.

Respond in strict JSON of the form:
{"label": 0 or 1, "reason": "<one short sentence>"}

- label = 1 means you believe the function would pass all provided tests.
- label = 0 means you believe at least one test would fail.

Do not include any other text outside of the JSON object.
```

Figure 8: System prompt for the GPT-5.4 mini judge in the LLM-as-a-judge experiment (Appendix N). The judge sees the function's source code and its generated unit tests, and predicts whether the function passes them.