# ASTOR: Multi-Task Code Reinforcement Learning via Utility-Driven Coordination

**Yujia Chen**[1] **Yang Ye**[2] **Xiao Chu**[2] **Yuchi Ma**[2] **Cuiyun Gao**[1*]
yujiachen@stu.hit.edu.cn gaocuiyun@hit.edu.cn
{yeyang14, chuxiao1, mayuchi1}@huawei.com

[1] Harbin Institute of Technology, Shenzhen [2] Huawei Cloud Computing Technologies Co., Ltd.

## Abstract

Reinforcement learning (RL) with verifiable rewards has proven effective at post-training LLMs for coding, yet deploying separate task-specific specialists incurs costs that scale with the number of tasks, motivating a unified multi-task RL (MTRL) approach. However, existing MTRL methods treat all coding tasks uniformly, relying on fixed data curricula under a shared optimization strategy, ultimately limiting the effectiveness of multi-task training. To address these limitations, we propose **ASTOR**, a multi-t**AS**k code reinforcement learning framework via u**T**ility-driven co**OR**dination. Centered on *task utility*, a signal capturing each task's learning potential and cross-task synergy, ASTOR comprises two coupled modules: *1) Hierarchical Utility-Routed Data Scheduling* module hierarchically allocates training budget and prioritizes informative prompts, steering training toward the most valuable data; and *2) Adaptive Utility-Calibrated Policy Optimization* module dynamically scales per-task KL regularization, matching update constraints to each task's current training state. Experiments on two widely-used LLMs across four representative coding tasks demonstrate that ASTOR consistently improves a single model across all tasks, outperforming the best task-specific specialist in average performance across all tasks by 9.0%–9.5% and surpassing the strongest MTRL baseline by 7.5%–12.8%.

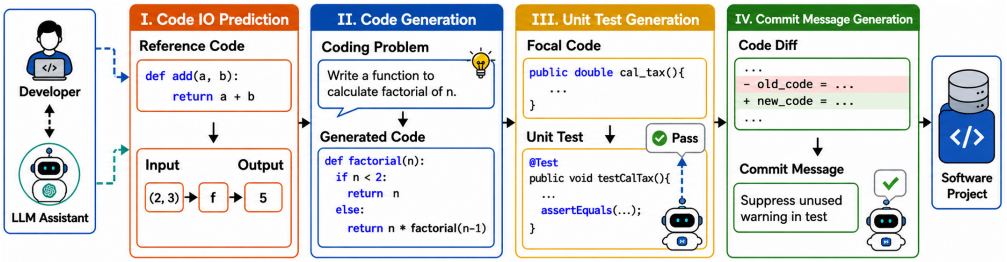
## 1 Introduction

In the era of large language models (LLMs), AI-assisted coding has become a mainstream development paradigm [1–4]. To complete a software project, developers typically need to understand program logic, generate code implementations, validate correctness through unit testing, and record changes via commit messages, as illustrated in Figure 1a. To enhance the capabilities of LLMs on these tasks, reinforcement learning (RL) has emerged as a powerful post-training technique, since coding tasks naturally provide execution-based reward signals [5–7]. Following this direction, recent works train task-specific RL specialists tailored to individual stages of the development workflow [8, 9]. However, deploying separate specialists per task incurs memory and compute costs that scale linearly with task count. Relying on a single specialist offers no remedy either, as no specialist matches the other specialists on their respective tasks (Figure 1b).

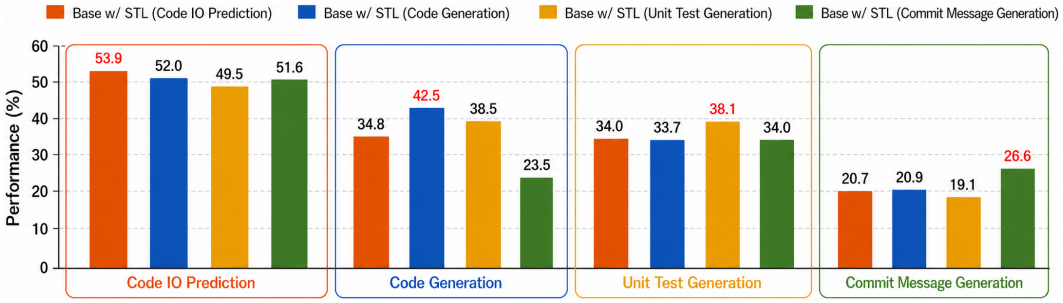
A natural solution is multi-task reinforcement learning (MTRL) [10–14], which trains a single LLM jointly across all coding tasks to achieve diverse capabilities without per-task deployment costs. In LLM post-training, we identify two coupled decisions at each MTRL step: *what to learn*, i.e., scheduling which tasks and prompts to train on, and *how to learn*, i.e., updating the shared policy

*Corresponding Author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).




(a) Four representative tasks spanning the software development workflow.




(b) Performance of task-specific RL specialists across the four tasks.

Figure 1: Performance of task-specific RL models across four coding tasks. Single-task learning (STL) models perform well on their own tasks but fall short of the corresponding specialists on other tasks.

across heterogeneous tasks. However, existing methods apply uniform strategies to both decisions, ignoring the inherent heterogeneity among coding tasks.

- For **what to learn**, existing methods either sample from a mixed task pool or follow a fixed curriculum, relying mainly on task-intrinsic signals [15, 16]. They ignore inter-task effects; for example, training on code generation may affect unit test generation, as both require program reasoning but emphasize different output behaviors. This leads to suboptimal task schedules.

- For **how to learn**, existing methods usually apply the same update strategy to all tasks [15, 16]. Yet coding tasks have different optimization requirements, with syntactically strict tasks demanding conservative updates for executable correctness and semantically flexible tasks tolerating greater language variation. A uniform strategy therefore over-constrains some tasks while leaving others insufficiently controlled.

Inspired by how skilled learners allocate more effort to less-mastered topics and leverage connections between subjects, we introduce *task utility*, a signal capturing both a task's learning potential and its cross-task synergy, and propose **ASTOR**, a multi-t**AS**k code reinforcement learning framework via u**T**ility-driven co**OR**dination that resolves both challenges through two coordinated modules. *Hierarchical Utility-Routed Data Scheduling* uses task utility to hierarchically allocate training budget across tasks and prioritize informative prompts within each task, addressing *what to learn*. *Adaptive Utility-Calibrated Policy Optimization* uses the same signal to calibrate per-task KL regularization, dynamically adapting policy optimization to each task's training state, addressing *how to learn*.

We evaluate ASTOR on four coding tasks spanning the software development workflow: code I/O prediction (understanding), code generation (implementation), unit test generation (testing), and commit message generation (documentation), using Qwen2.5-Coder-7B [17] and Qwen3-8B [18] as base models. Experiments demonstrate that the single ASTOR model surpasses the best task-specific RL specialist by 9.0%–9.5% in average performance across all tasks. ASTOR further surpasses all MTRL baselines by 7.5%–19.2% across models and baselines, demonstrating the effectiveness of utility-driven coordination across heterogeneous coding tasks.

The main contributions of our paper are summarized as follows:

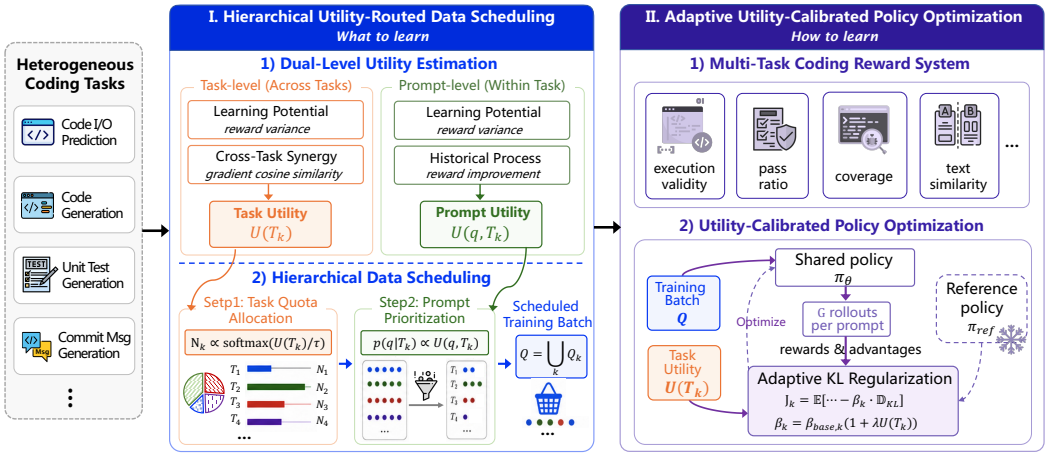



Figure 2: Overview of ASTOR. (I) hierarchically schedules training data at task and prompt levels based on estimated task utilities; and (II) updates the shared policy with task-adaptive KL regularization conditioned on task utilities.

- To the best of our knowledge, we are the first to study multi-task RL for code LLMs, formulating it around two coupled decisions, what to learn and how to learn, and proposing task utility as a unified signal governing both.
- We propose ASTOR, a coordination framework with hierarchical task allocation and prompt prioritization for data scheduling, and task-adaptive KL regularization for policy optimization.
- We empirically validate ASTOR across multiple base LLMs and heterogeneous coding tasks, showing consistent gains over both task-specific RL specialists and MTRL baselines.

## 2 Methodology

As illustrated in Figure 2, ASTOR consists of two coupled modules: *(1) Hierarchical Utility-Routed Data Scheduling* hierarchically schedules training data at task and prompt levels based on estimated task utilities; *(2) Adaptive Utility-Calibrated Policy Optimization* updates the shared policy with task-adaptive KL regularization conditioned on task utilities.

### 2.1 Problem Formulation

We formulate multi-task code post-training as a reinforcement learning problem over $K$ heterogeneous coding tasks $\mathcal{T} = \{T_1, \ldots, T_K\}$. Each task $T_k$ provides a prompt dataset $\mathcal{D}_k$ and a reward function $R_k$ for evaluating model rollouts. All tasks share a single policy $\pi_\theta$ parameterized by the code LLM, together with a frozen reference policy $\pi_{\text{ref}}$ for regularization.

At each training iteration $t$, we face two coupled decisions that jointly determine how the shared policy improves across heterogeneous tasks:

**Decision 1: What to Learn.** The first decision concerns *data scheduling*, i.e., which tasks and which prompts to select for the next policy update. Under a fixed total batch budget $B$, we allocate task-specific quotas $\{N_k^{(t)}\}_{k=1}^K$ and sample prompt batches $\mathcal{Q}_k^{(t)} \subseteq \mathcal{D}_k$ with $|\mathcal{Q}_k^{(t)}| = N_k^{(t)}$ such that

$$\mathcal{Q}^{(t)} = \bigcup_{k=1}^{K} \mathcal{Q}_k^{(t)}, \qquad \sum_{k=1}^{K} N_k^{(t)} = B. \tag{1}$$

This decision determines both *which tasks* appear in the update (via task quotas) and *which prompts* represent each task (via prompt selection). In multi-task RL, an effective schedule should account for both task-intrinsic signals (e.g., whether a task still has learning potential) and cross-task effects (e.g., whether training on one task benefits others). Ignoring either aspect leads to suboptimal allocation. Over-sampling saturated tasks wastes computation, while over-sampling conflicting tasks induces negative transfer.

**Decision 2: How to Learn.** The second decision concerns *policy optimization*, i.e., how to update the shared policy on the scheduled data. We build on GRPO [19] with task-specific KL regularization. For each prompt $q$, we sample $G$ rollouts $\{o_i\}_{i=1}^{G}$ from the current policy and maximize:

$$J_k(\theta;t) = \mathbb{E}_{q\sim\mathcal{Q}_k^{(t)},\{o_i\}}\left[\frac{1}{G}\sum_{i=1}^{G}\min\Big(\rho_i A_i, \mathrm{clip}(\rho_i, 1-\epsilon, 1+\epsilon)A_i\Big) - \beta_k(t)\,\mathbb{D}_{\mathrm{KL}}\big[\pi_\theta(\cdot\mid q)\|\pi_{\text{ref}}(\cdot\mid q)\big]\right], \tag{2}$$

where $\rho_i = \pi_\theta(o_i|q)/\pi_{\theta_{\text{old}}}(o_i|q)$ is the probability ratio between the current policy and the pre-update policy $\pi_{\theta_{\text{old}}}$, $A_i$ is the group-normalized advantage, and $\beta_k(t)$ is a task-specific KL coefficient.

The multi-task objective aggregates task-specific objectives weighted by their batch proportions:

$$J_{\text{MT}}(\theta;t) = \sum_{k=1}^{K}\frac{N_k^{(t)}}{B}\,J_k(\theta;t). \tag{3}$$

These two decisions are coupled; task quotas $N_k^{(t)}$ and prompt batches $\mathcal{Q}_k^{(t)}$ determine *what to learn*, while KL coefficients $\beta_k(t)$ determine *how to learn*. In the following, we describe how ASTOR addresses these coupled decisions. Additional formulation details are provided in Appendix A.1.

### 2.2 Hierarchical Utility-Routed Data Scheduling

This module addresses *what to learn* by using task utility to hierarchically allocate training budget $B$ across tasks and prioritize informative prompts within each task.

#### 2.2.1 Dual-Level Utility Estimation

Intuitively, a high-utility task should exhibit two key properties: *learning potential*, which reflects actionable optimization signals, and *cross-task synergy*, which captures constructive gradient alignment with other tasks. In other words, such a task should both provide useful learning signals for the current model and facilitate positive transfer across tasks. We therefore define task utility as a signal that jointly captures both properties.

**Task Utility.** *1) Learning Potential.* We employ the statistical variance of rollout rewards as an aggregation metric for remaining learning opportunity. High variance indicates a heterogeneous result set where the model yields both successful and failed trajectories. For a sampled query batch $\mathcal{Q}_k \subseteq \mathcal{D}_k$ at step $t$, the potential utility is defined as

$$\mathcal{U}_{\mathsf{pot}}^{(t)}(T_k) = \frac{1}{|\mathcal{Q}_k|}\sum_{q\in\mathcal{Q}_k}\mathsf{Var}\left(\{R_k(q,o_{q,i})\}_{i=1}^{G}\right) \tag{4}$$

*2) Cross-Task Synergy.* High learning potential alone does not guarantee positive cross-task effects. From a first-order optimization perspective, processing $T_k$ induces a parameter update $\theta \leftarrow \theta + \eta\mathbf{g}_k$, whose effect on task $T_j$ is proportional to $\eta\langle\mathbf{g}_j,\mathbf{g}_k\rangle$, which is positive when the two tasks reinforce each other and negative when they conflict. This motivates using gradient cosine similarity as a measure of cross-task synergy [20–22]. Let $\mathbf{g}_k$ denote the policy gradient aggregated from task $T_k$; the synergy score is given by

$$\mathcal{U}_{\mathsf{syn}}^{(t)}(T_k) = \frac{1}{|\mathcal{T}|-1}\sum_{T_j\in\mathcal{T}\setminus\{T_k\}}\mathsf{sim}(\mathbf{g}_k,\mathbf{g}_j) \tag{5}$$

Note that computing full-parameter gradient similarity is infeasible for LLM-scale models; we compress each task gradient by summing over all parameter dimensions except the last within each layer, then concatenating the resulting row vectors across layers into a compact vector per task. The compressed vectors serve as an efficient proxy for measuring task relationships rather than exactly preserving gradient directions: the resulting similarities agree in sign with those computed from full gradients in 92.3% of cases (Spearman correlation 0.91). Details, validation, and scalability analysis are in Appendix A.2.

**Prompt Utility.** Within a task, not all prompts are equally informative. A prompt the model already solves provides no useful gradient signal; a prompt the model never solves contributes little beyond reinforcing failures. The most informative prompts are those where the model is actively improving and still uncertain about the correct output. We therefore define prompt-level utility as a combination of current learning potential and historical progress. Letting $\mu_R^{(t)}(q)$ denote the mean reward over $G$ rollouts at step $t$, we define

$$\mathcal{U}_{\mathsf{prog}}^{(t)}(q, T_k) = \mu_R^{(t)}(q) - \mu_R^{(t-1)}(q) \tag{6}$$

$$\mathcal{U}_{\mathsf{pot}}^{(t)}(q, T_k) = \mathsf{Var}\left(\{R_k(q, o_{q,i})\}_{i=1}^{G}\right) \tag{7}$$

where $t-1$ denotes the last earlier step at which $q$ was sampled.

### 2.2.2 Hierarchical Data Scheduling

Equipped with the utility metrics, we formulate batch construction as a hierarchical allocation procedure, first determining the training budget quota per task, and subsequently prioritizing prompts to fulfill each quota.

**Task Quota Allocation.** To allocate the budget $B$ across tasks, we need a single scalar utility per task. The two raw components, however, are not yet comparable: each training step observes only one mini-batch per task, making per-batch statistics noisy, and reward variance and gradient cosine similarity operate on different numerical scales, so one would dominate the combined utility without rescaling. We apply EMA smoothing [23] and min-max normalization across tasks to each component before fusion; details are in Appendix A.3. The task-level utility is then

$$\mathcal{U}^{(t)}(T_k) = \mathsf{EMA}\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k)\Big) + \mathsf{EMA}\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k)\Big) \tag{8}$$

We then partition the global budget $B$ via a temperature-scaled softmax to yield the training quota $N_k^{(t)}$ for task $T_k$,

$$N_k^{(t)} = B \cdot \frac{\exp\left(\mathcal{U}^{(t)}(T_k)/\tau\right)}{\sum_{T_j \in \mathcal{T}} \exp\left(\mathcal{U}^{(t)}(T_j)/\tau\right)} \tag{9}$$

where $\tau > 0$ controls how sharply the budget concentrates on high-utility tasks.

**Prompt Prioritization.** Within each task's quota $N_k^{(t)}$, the same preprocessing is applied at the prompt level,

$$\mathcal{U}^{(t)}(q, T_k) = \mathsf{EMA}\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(q, T_k)\Big) + \mathsf{EMA}\Big(\mathcal{U}_{\mathsf{prog}}^{(t-1)}(q, T_k)\Big) \tag{10}$$

Prompts are then selected via a sigmoid-transformed distribution over $\mathcal{D}_k$,

$$\pi^{(t)}(q \mid T_k) = \frac{\sigma\left(\mathcal{U}^{(t)}(q, T_k)\right)}{\sum_{q' \in \mathcal{D}_k} \sigma\left(\mathcal{U}^{(t)}(q', T_k)\right)} \tag{11}$$

and $\mathcal{Q}_k^{(t)}$ is drawn via weighted sampling without replacement. Prompts that have not yet been selected are assigned a weight of 1, the upper bound of $\sigma(\cdot)$, which encourages coverage of $\mathcal{D}_k$ (Appendix A.3). These task-specific batches are combined to produce the final heterogeneous training batch $\mathcal{Q}^{(t)} = \bigcup_k \mathcal{Q}_k^{(t)}$.

## 2.3 Adaptive Utility-Calibrated Policy Optimization

This module addresses *how to learn* by dynamically calibrating the KL regularization coefficient $\beta_k(t)$ for each task based on its current task utility, enabling task-adaptive policy optimization on the heterogeneous training batch produced by the first module.

### 2.3.1 Multi-Task Coding Reward System

The reward system defines the interface between the policy and the training environment. ASTOR only requires each task to provide an automatically computable scalar reward. In our experiments, to cover a representative range of software development competencies, we instantiate the reward system across four tasks that span distinct programming paradigms, each grounded in an objective, verifiable reward signal.

- **Code I/O Prediction.** Given a function signature, predict valid input-output pairs. The reward evaluates execution correctness: $\mathcal{R}_{\mathsf{IO}}(o) = \mathbb{I}[\mathsf{Exec}(o) \text{ is valid}] \in \{0, 1\}$.
- **Code Generation.** Given a problem description, synthesize an implementation. The metric is the test case pass rate: $\mathcal{R}_{\mathsf{CG}}(o) = \mathsf{PassRatio}(o)$.
- **Unit Test Generation.** Given a focal function, generate unit tests. The metric calculates line/branch coverage: $\mathcal{R}_{\mathsf{UT}}(o) = \mathsf{Coverage}(o)$.
- **Commit Message Generation.** Given a code diff, generate a commit message. The metric measures textual similarity against ground truth: $\mathcal{R}_{\mathsf{CM}}(o) = \mathsf{TextSim}(o, o_{\mathsf{gt}})$.

To ensure structural validity, the final reward objective fuses the task-specific metric with a format compliance constraint:

$$\mathcal{R}(o) = w_{T_k} \cdot \mathcal{R}_{T_k}(o) + w_{\mathsf{fmt}} \cdot \mathcal{R}_{\mathsf{fmt}}(o) \tag{12}$$

where $T_k \in \{\mathsf{IO}, \mathsf{CG}, \mathsf{UT}, \mathsf{CM}\}$, and $\mathcal{R}_{\mathsf{fmt}} \in \{0, 1\}$ acts as a strict boolean filter for output parsing. Implementation details of these reward functions, including the concrete weights and format penalty used in our experiments, are provided in Appendix A.4, and prompt templates for all tasks are in Appendix A.5.

### 2.3.2 Utility-Calibrated Policy Optimization

Coding tasks differ in how tightly their outputs are constrained: some must produce complete executable programs, whose syntactic validity is easily broken by large policy shifts, whereas others produce short answers or natural language, for which broader exploration is less risky. As a result, the uniform KL coefficient used by existing methods over-constrains some tasks while under-constraining others, and the appropriate constraint also changes as a task's utility evolves. We therefore decompose each task's KL coefficient into a task-specific base and a dynamic multiplier conditioned on task utility,

$$\beta_k^{(t)} = \beta_{\mathsf{base},k} \cdot \left(1 + \lambda \cdot \mathcal{U}^{(t)}(T_k)\right) \tag{13}$$

where $\beta_{\mathsf{base},k}$ is a task-specific baseline coefficient set according to the task's output constraints and $\lambda$ scales how strongly utility modulates it. The sampling quota and the KL coefficient play complementary roles. A high-utility task receives a larger quota so that training makes greater use of it, and a larger $\beta_k^{(t)}$ so that the shared policy does not over-specialize toward it. Low utility may instead reflect saturation, persistent failure, or task conflict; in each case, the smaller quota already limits the task's contribution, so a smaller $\beta_k^{(t)}$ avoids further suppressing the useful signal its remaining samples may still carry.

Overall, ASTOR operates as an end-to-end coordination framework, alternating between hierarchical task allocation and prompt prioritization (data scheduling) and task-adaptive policy optimization. The complete algorithm pseudocode is presented in Appendix A.6.

## 3 Experiments

### 3.1 Experimental Setup

**Training Datasets.** We construct training datasets for four heterogeneous coding tasks: 2,088 samples for *Code I/O Prediction* from LeetCode [24]; 2,082 samples for *Code Generation* from LiveCodeBench [25] and Codeforces [26]; 1,148 samples for *Unit Test Generation* from real-world open-source Java repositories with executable validation environments; and 2,446 samples for *Commit Message Generation* from the MCMD dataset [27]. Dataset construction details are provided in Appendix B.1.

Table 1: Performance comparison of ASTOR and baselines across four coding tasks. The abbreviation of "STL" denotes Single-Task Learning. Bolded values mark the best per column; underscored values mark the second best.

| Method | CRUXEval *Code I/O Prediction* | | Aider-Polyglot *Code Generation* | | Defects4J *Unit Test Generation* | | | MCMDEval+ *Commit Message Gen.* | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Input Acc. | Output Acc. | Pass@1 | Pass@2 | Line Cov. | Branch Cov. | Compile Rate | BLEU | ROUGE | METEOR |
| ***Qwen2.5-Coder-7B*** | *47.8* | *41.0* | *24.4* | *26.7* | *24.4* | *20.5* | *49.3* | *23.8* | *15.5* | *12.9* |
| STL (Code I/O) | 54.4 | 53.5 | 31.1 | 38.5 | 28.4 | 23.8 | 49.9 | 29.4 | 17.0 | 15.7 |
| STL (Code Gen) | 51.8 | 52.2 | 40.7 | 44.4 | 28.1 | 23.5 | 51.0 | 28.2 | 17.2 | 17.4 |
| STL (Unit Test) | 48.6 | 50.4 | 36.3 | 40.7 | 30.8 | 24.6 | 53.0 | 26.0 | 15.7 | 15.6 |
| STL (Commit Msg) | 52.4 | 50.8 | 19.3 | 27.4 | 29.2 | 23.2 | 50.0 | **41.2** | 21.4 | 17.3 |
| Joint Learning | 51.4 | 54.0 | 32.6 | 43.0 | 25.9 | 21.0 | 51.1 | 30.7 | 17.1 | 16.0 |
| Curriculum Learning | 51.1 | 52.5 | 28.1 | 35.6 | 27.8 | 23.1 | 48.4 | 30.5 | 17.3 | 16.6 |
| Model Merging | **54.7** | 53.3 | 23.7 | 31.1 | 27.2 | 22.4 | 46.8 | 30.0 | 17.9 | 17.1 |
| **ASTOR** | 52.2 | **56.4** | **41.5** | **45.2** | **32.3** | **25.7** | **55.7** | 35.2 | **22.1** | **20.2** |
| ***Qwen3-8B*** | *53.3* | *56.3* | *38.5* | *43.0* | *30.9* | *26.9* | *50.2* | *33.8* | *16.6* | *14.8* |
| STL (Code I/O) | 61.0 | 63.1 | 43.0 | 48.9 | 32.4 | 28.0 | 53.2 | 35.6 | 17.4 | 15.8 |
| STL (Code Gen) | 57.8 | 61.3 | 50.4 | 53.3 | 31.8 | 27.6 | 52.8 | 35.0 | 17.6 | 16.7 |
| STL (Unit Test) | 56.5 | 59.8 | 45.2 | 51.1 | 37.2 | 31.2 | 55.9 | 34.0 | 17.0 | 15.2 |
| STL (Commit Msg) | 57.0 | 60.3 | 40.7 | 45.9 | 32.0 | 27.4 | 52.0 | 40.8 | 21.4 | 18.6 |
| Joint Learning | 58.8 | 62.5 | 47.4 | 52.6 | 34.0 | 29.1 | 54.4 | 37.2 | 18.8 | 17.3 |
| Curriculum Learning | 57.3 | 60.6 | 42.2 | 49.6 | 33.1 | 28.5 | 52.9 | 36.6 | 18.4 | 16.8 |
| Model Merging | 59.5 | 61.5 | 40.0 | 46.7 | 32.6 | 28.2 | 50.7 | 36.2 | 19.1 | 17.6 |
| **ASTOR** | **61.5** | **65.1** | **51.1** | **54.8** | **38.1** | **32.0** | **56.6** | **41.2** | **22.5** | **20.0** |

**Evaluation Benchmarks and Metrics.** We evaluate each task on representative and widely-used benchmarks with task-specific metrics. For *Code I/O Prediction*, we report input and output accuracy on CRUXEval [28]. For *Code Generation*, we measure pass@1 and pass@2 on Aider-Polyglot-Python [29]. For *Unit Test Generation*, we evaluate line coverage, branch coverage, and compilation rate on Defects4J 2.0 [30]. For *Commit Message Generation*, we report BLEU [31], ROUGE [32], and METEOR [33] on MCMDEval+ [27]. Detailed descriptions of each benchmark and metric are provided in Appendix B.2.

**Baselines.** We include *(1) Single-Task Learning (STL)* [8, 9]: for each task, we train an independent model using GRPO [19] on that task's data only, representing task-specific specialists.

We further compare against three multi-task baselines spanning the main paradigms in LLM-based MTRL, following prior works [15, 16]. *(2) Joint Learning*, following Wu et al. [15]: all tasks are uniformly mixed within each training batch and trained with a shared GRPO objective. *(3) Curriculum Learning*, in the spirit of OMNI-THINKER [16]: tasks are trained sequentially in a fixed order (Code I/O $\rightarrow$ Code Gen $\rightarrow$ Unit Test $\rightarrow$ Commit Message), which follows the software development workflow in Figure 1a (understanding, implementation, testing, documentation) rather than task difficulty. *(4) Model Merging*: task-specific GRPO specialists are trained separately, then merged post-hoc via TIES-Merging [34]. All experiments are conducted on Qwen2.5-Coder-7B [17] and Qwen3-8B [18]. Baseline configurations are detailed in Appendix B.3.

**Implementation Details.** We use the MindSpeed [35] pipeline for RL fine-tuning, with a global batch size of 128 and 8 rollouts per sample. All experiments are trained for 400 steps on 32 Ascend 910B-B3 NPUs. Full hyperparameter settings are provided in Appendix B.4.

### 3.2 Main Results

Table 1 reports the performance of ASTOR and all baselines across four coding tasks on two base models.

**Comparison with Single-Task Learning.** ASTOR achieves strong multi-task performance that rivals or exceeds task-specific specialists. For Qwen2.5-Coder-7B, ASTOR surpasses the best single-task model in 8 out of 10 metrics, with an average score of 38.65. This demonstrates that a single model can handle multiple tasks without significant trade-offs. Similar trends hold for Qwen3-8B, where

Table 2: Ablation results of ASTOR.

| Method | CRUXEval *Code I/O Prediction* | | Aider-Polyglot *Code Generation* | | Defects4J *Unit Test Generation* | | | MCMDEval+ *Commit Message Gen.* | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Input Acc. | Output Acc. | Pass@1 | Pass@2 | Line Cov. | Branch Cov. | Compile Rate | BLEU | ROUGE | METEOR |
| ***Qwen2.5-Coder-7B*** | | | | | | | | | | |
| **ASTOR** | **52.2** | **56.4** | **41.5** | **45.2** | **32.3** | **25.7** | **55.7** | **35.2** | **22.1** | **20.2** |
| • *Data Schedule* | | | | | | | | | | |
| w/o Task Allocation | 55.8 | 55.2 | 34.8 | 41.5 | 28.1 | 23.5 | 51.1 | 33.8 | 19.4 | 17.8 |
| w/o Prompt Prioritization | 53.6 | 56.7 | 34.8 | 40.7 | 27.8 | 24.1 | 51.5 | 31.9 | 17.9 | 16.7 |
| w/o Task & Prompt (Random) | 53.3 | 51.2 | 31.9 | 41.5 | 27.5 | 22.2 | 50.0 | 31.4 | 17.8 | 16.6 |
| • *Policy Optimization* | | | | | | | | | | |
| w/o Dynamic Regularization | 52.9 | 51.9 | 35.6 | 43.7 | 30.6 | 26.2 | 51.2 | 32.9 | 18.6 | 16.7 |
| w/o Task-Specific Regularization | 53.6 | 50.7 | 34.1 | 38.5 | 30.6 | 25.5 | 49.9 | 34.3 | 18.1 | 16.3 |
| ***Qwen3-8B*** | | | | | | | | | | |
| **ASTOR** | **61.5** | **65.1** | **51.1** | **54.8** | **38.1** | **32.0** | **56.6** | **41.2** | **22.5** | **20.0** |
| • *Data Schedule* | | | | | | | | | | |
| w/o Task Allocation | 61.0 | 63.9 | 48.1 | 53.3 | 35.1 | 29.6 | 54.9 | 38.4 | 19.6 | 17.8 |
| w/o Prompt Prioritization | 60.6 | 64.0 | 49.6 | 53.3 | 36.0 | 30.3 | 55.2 | 38.0 | 19.4 | 17.6 |
| w/o Task & Prompt (Random) | 59.1 | 62.6 | 47.4 | 52.6 | 34.3 | 29.2 | 54.4 | 37.3 | 18.9 | 17.3 |
| • *Policy Optimization* | | | | | | | | | | |
| w/o Dynamic Regularization | 60.3 | 63.5 | 48.9 | 53.3 | 36.8 | 31.0 | 55.4 | 38.8 | 19.9 | 18.0 |
| w/o Task-Specific Regularization | 60.4 | 63.1 | 48.1 | 52.6 | 36.5 | 30.8 | 54.8 | 39.2 | 20.1 | 18.2 |

ASTOR achieves the best results across all 10 metrics (44.29 vs. 40.43). It even exceeds using each task's own specialist (38.13 and 43.29 on average), which requires four separate models.

**Comparison with Multi-Task Baselines.** ASTOR greatly outperforms all multi-task baselines across both models. On Qwen2.5-Coder-7B, it outperforms Joint Learning (34.28), Curriculum Learning (33.10), and Model Merging (32.42) by 12.8%, 16.8%, and 19.2%, respectively. Joint Learning suffers from severe task interference (its code generation pass@1 is 21.4% lower than that of ASTOR); Curriculum Learning is limited by its fixed task ordering; and Model Merging shows that post-hoc integration of specialists is inferior to joint optimization.

**Generalization Across Model Architectures.** ASTOR generalizes consistently across model families and scales. On Qwen2.5-Coder-7B it scores 38.65 (+9.0% over the best specialist, +12.8% over Joint Learning), and on Qwen3-8B it reaches 44.29 (+9.5% and +7.5%, respectively), confirming that utility-driven scheduling and task-adaptive policy optimization transfer consistently across different model capacities.

### 3.3 Ablation Studies

Table 2 reports ablations of the two key modules of ASTOR: data schedule and policy optimization.

**Data Schedule Module.** We ablate the hierarchical data schedule by removing task allocation, prompt prioritization, or both.

► *Task Allocation.* Replacing utility-driven task allocation with uniform sampling causes notable degradation on Qwen2.5-Coder-7B: code generation pass@1 declines by 16.1% and unit test line coverage by 13.0%. Equal resource allocation ignores dynamic learning states, leading to ineffective training and negative transfer.

► *Prompt Prioritization.* Replacing utility-based prompt selection with random sampling within each task yields similar declines (pass@1 $-16.1\%$, line coverage $-13.9\%$). Prioritizing high-utility prompts concentrates training on informative samples, accelerating convergence and reducing wasted computation.

► *Complete Random Sampling.* Removing both components produces the worst results (pass@1 $-23.1\%$, line coverage $-14.9\%$), on par with the Joint Learning baseline, confirming that hierarchical utility-driven scheduling is essential for coordinating heterogeneous tasks.

**Policy Optimization Module.** We ablate dynamic and task-specific KL regularization.

► *Dynamic Regularization.* Fixing KL coefficients throughout training (no utility-based adaptation) degrades code generation pass@1 by 14.2% and code I/O output accuracy by 8.0% on Qwen2.5-

Table 3: Sensitivity analysis of (a) softmax temperature $\tau$ in task allocation and (b) KL adjustment rate $\lambda$ in dynamic regularization, evaluated on Qwen2.5-Coder-7B.

| Hyperparameter | CRUXEval *Code I/O Prediction* | | Aider-Polyglot *Code Generation* | | Defects4J *Unit Test Generation* | | | MCMDEval+ *Commit Message Gen.* | | | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Input Acc. | Output Acc. | Pass@1 | Pass@2 | Line Cov. | Branch Cov. | Compile Rate | BLEU | ROUGE | METEOR | |
| *(a) Softmax Temperature $\tau$ — Task Allocation Module* | | | | | | | | | | | |
| $\tau = 0.5$ | 50.4 | 53.8 | **42.2** | **45.9** | **33.5** | **26.2** | **56.4** | 33.4 | 20.4 | 18.4 | 38.06 |
| $\tau = \mathbf{1.0}$ | 52.2 | **56.4** | 41.5 | 45.2 | 32.3 | 25.7 | 55.7 | **35.2** | **22.1** | **20.2** | **38.65** |
| $\tau = 2.0$ | **53.8** | 55.3 | 37.8 | 42.2 | 29.8 | 23.8 | 52.8 | 34.0 | 20.6 | 18.8 | 36.89 |
| *(b) KL Adjustment Rate $\lambda$ — Policy Optimization Module* | | | | | | | | | | | |
| $\lambda = 0.1$ | 52.5 | 54.3 | 38.5 | 43.7 | 31.2 | 25.3 | 53.5 | 34.0 | 20.3 | 18.5 | 37.18 |
| $\lambda = \mathbf{0.2}$ | 52.2 | **56.4** | **41.5** | **45.2** | **32.3** | **25.7** | **55.7** | **35.2** | **22.1** | **20.2** | **38.65** |
| $\lambda = 0.4$ | 52.4 | 55.6 | 40.0 | 44.4 | 31.8 | 25.2 | 54.4 | 34.4 | 21.0 | 19.2 | 37.84 |
| $\lambda = 0.8$ | **53.0** | 52.8 | 37.0 | 42.2 | 30.4 | 24.6 | 52.0 | 33.4 | 19.0 | 17.4 | 36.18 |

Coder-7B. Fixed regularization cannot accommodate evolving task interactions: over-regularization limits synergistic updates while under-regularization enables negative transfer.

► *Task-Specific Regularization.* Replacing task-specific KL coefficients with a single uniform value causes code generation pass@2 to decline by 14.8% and unit test compilation rate to drop by 10.4% on Qwen2.5-Coder-7B. Tasks are fundamentally heterogeneous: text-generation tasks require lower regularization for diverse reasoning, while syntactic tasks like unit test generation need stronger constraints for correctness, and a uniform coefficient cannot accommodate both.

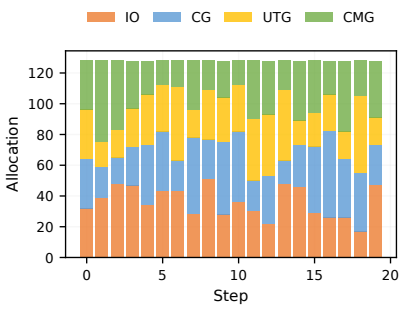
### 3.4 Further Analysis

We analyze key training dynamics and hyperparameter sensitivity of ASTOR on Qwen2.5-Coder-7B.

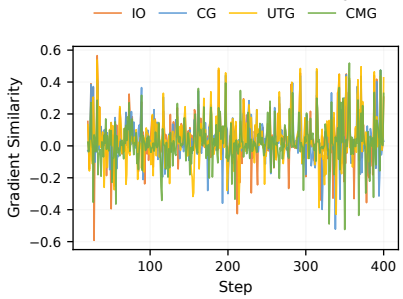
**Impact of $\tau$.** As shown in Table 3(a), a moderate $\tau$ yields the best performance; deviations in either direction lead to degradation. Too small a $\tau$ causes the scheduler to over-concentrate budget on high-utility tasks, depriving lower-utility tasks of sufficient training signal. Too large a $\tau$ flattens the allocation toward uniform sampling, diminishing the effect of utility-driven scheduling.

**Impact of $\lambda$.** As shown in Table 3(b), performance is relatively consistent across moderate values of $\lambda$ but degrades at both extremes. When $\lambda$ is too small, the utility-based adjustment becomes negligible, effectively reducing to a static KL scheme that cannot adapt to each task's training dynamics. When $\lambda$ is too large, high-utility tasks receive overly tight regularization that limits policy updates even where room for improvement remains.

**Training Dynamics Analysis.** As shown in Figure 3a, task allocation evolves continuously throughout training, with each task's quota rising or falling in response to its real-time utility, a pattern that would be invisible to any fixed schedule. Figure 3b reveals why task-specific regularization matters. Unit test generation exhibits persistent gradient conflicts with other tasks (drops $< -0.4$), while code I/O and commit message generation remain largely synergistic, underscoring the need to treat tasks differently at the policy level. Qualitative examples of model outputs across tasks are provided in Appendix D.




(a) Task allocation over training.




(b) Cross-task gradient similarity.

Figure 3: Training dynamics of ASTOR on Qwen2.5-Coder-7B.

## 4 Related Work

**Multi-Task Learning.** Multi-task learning [10] trains a shared model on multiple tasks, where gradient interference is a central challenge. Gradient-manipulation methods mitigate this at the update

level; for example, PCGrad [20] projects conflicting gradients onto orthogonal planes. In multi-task RL, Distral [14] shares a distilled centroid policy across task workers, using KL regularization to couple separate task-specific policies with a shared one. In LLM post-training, Wu et al. [15] show that naive task mixing induces severe gradient imbalance and that sampling tasks in proportion to gradient magnitude brings no consistent gain; OMNI-THINKER [16] addresses cross-task forgetting via backward-transfer-guided curriculum scheduling with a task order fixed before training. Unlike these works, ASTOR estimates task utility online and uses it to decide, within a single shared policy, both the composition of each batch and the regularization strength of each task. A detailed discussion is in Appendix G.

**RL Post-Training for Code LLMs.** RL with verifiable rewards has emerged as the dominant post-training paradigm for code LLMs [19]. For example, RLEF [8] trains code models on multi-turn execution feedback, CodeRL+ [6] integrates execution semantics alignment into the RL objective, and Lee et al. [9] apply adversarial RL for unit test generation. All of these approaches train task-specific specialists that degrade sharply outside their training domain. ASTOR addresses this by jointly post-training a single model across various heterogeneous coding tasks under a unified multi-task RL framework. A detailed discussion is in Appendix G.

## 5 Conclusion

In this paper, we propose ASTOR, a utility-driven coordination framework that addresses both through a shared signal, *task utility*. *Hierarchical Utility-Routed Data Scheduling* uses task utility to hierarchically allocate training budget across tasks and prioritize informative prompts within each task; *Adaptive Utility-Calibrated Policy Optimization* uses the same signal to calibrate per-task KL regularization, enabling task-adaptive policy optimization across heterogeneous tasks. Experiments across four representative coding tasks and two base LLMs show that ASTOR consistently surpasses both task-specific RL specialists and multi-task baselines, validating the effectiveness of utility-driven coordination for multi-task code RL with a single model. Limitations and future work are discussed in Appendix E.

## Acknowledgments and Disclosure of Funding

This research is supported by PCL-CMCC Foundation for science and innovation (No. SQ2510290006), National Natural Science Foundation of China under project (No. 62472126), Shenzhen Natural Science Foundation Youth Program (No. QNXMB20250701092047022), and Young Elite Scientist Cultivation Program By SAST (No. 0832-SFCX25FSC235 (2026XHXS03)).

## References

[1] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[2] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. The current challenges of software engineering in the era of large language models. *ACM Trans. Softw. Eng. Methodol.*, 34(5):127:1–127:30, 2025.

[3] Kaixin Wang, Tianlin Li, Xiaoyu Zhang, Chong Wang, Weisong Sun, Yang Liu, and Bin Shi. Software development life cycle perspective: A survey of benchmarks for code large language models and agents. *CoRR*, abs/2505.05283, 2025.

[4] Yujia Chen, Yang Ye, Zhongqi Li, Yuchi Ma, and Cuiyun Gao. Smaller but better: Self-paced knowledge distillation for lightweight yet effective lcms. *Proc. ACM Softw. Eng.*, 2(FSE):3057–3080, 2025.

[5] 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, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, and et al Guanting Chen. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nat.*, 645(8081):633–638, 2025.

[6] Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. Coderl+: Improving code generation via reinforcement with execution semantics alignment. *CoRR*, abs/2510.18471, 2025.

[7] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, and et al Haozhen Yu. Kimi k1.5: Scaling reinforcement learning with llms. *CoRR*, abs/2501.12599, 2025.

[8] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: grounding code llms in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net, 2025.

[9] Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning. *CoRR*, abs/2508.21107, 2025.

[10] Rich Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, 1997.

[11] Tung-Long Vuong, Do Van Nguyen, Tai-Long Nguyen, Cong-Minh Bui, Hai-Dang Kieu, Viet-Cuong Ta, Quoc-Long Tran, and Thanh Ha Le. Sharing experience in multitask reinforcement learning. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 3642–3648. ijcai.org, 2019.

[12] Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. Multi-task reinforcement learning with soft modularization. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[13] Myungsik Cho, Jongeui Park, Suyoung Lee, and Youngchul Sung. Hard tasks first: Multi-task reinforcement learning through task scheduling. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[14] Yee Whye Teh, Victor Bapst, Wojciech M. Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4496–4506, 2017.

[15] Runzhe Wu, Ankur Samanta, Ayush Jain, Scott Fujimoto, Jeongyeol Kwon, Ben Kretzu, Youliang Yu, Kaveh Hassani, Boris Vidolov, and Yonathan Efroni. Imbalanced gradients in RL post-training of multi-task llms. *CoRR*, abs/2510.19178, 2025.

[16] Derek Li, Jiaming Zhou, Amirreza Kazemi, Qianyi Sun, Abbas Ghaddar, Liheng Ma, Yu Luo, Dong Li, Jianye HAO, and Yingxue Zhang. Omni-thinker: Scaling cross-domain generalization in llms via multi-task rl with hybrid rewards. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

[17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186, 2024.

[18] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jian Yang, Jiaxi Yang, Jingren Zhou, and et al Junyang Lin. Qwen3 technical report. *CoRR*, abs/2505.09388, 2025.

[19] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024.

[20] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[21] Zhao Chen, Jiquan Ngiam, Yanping Huang, Thang Luong, Henrik Kretzschmar, Yuning Chai, and Dragomir Anguelov. Just pick a sign: Optimizing deep multitask models with gradient sign dropout. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[22] Bo Liu, Xingchao Liu, Xiaojie Jin, Peter Stone, and Qiang Liu. Conflict-averse gradient descent for multi-task learning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 18878–18890, 2021.

[23] J Stuart Hunter. The exponentially weighted moving average. *Journal of quality technology*, 18(4):203–210, 1986.

[24] Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *CoRR*, abs/2504.14655, 2025.

[25] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[26] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

[27] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, and Yue Tan. Automated commit message generation with large language models: An empirical study and beyond. *IEEE Trans. Software Eng.*, 50(12):3208–3224, 2024.

[28] Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[29] Paul Gauthier. Aider llm leaderboard, 2025. URL `https://aider.chat/docs/leaderboards/`.

[30] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL, 2002.

[32] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[33] Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss, editors, *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, pages 65–72. Association for Computational Linguistics, 2005.

[34] Han Wu, Yuxuan Yao, Shuqi Liu, Zehua Liu, Xiaojin Fu, Xiongwei Han, Xing Li, Hui-Ling Zhen, Tao Zhong, and Mingxuan Yuan. Unlocking efficient long-to-short LLM reasoning with model merging. *CoRR*, abs/2503.20641, 2025.

[35] Laingjun Feng, Chenyi Pan, Xinjie Guo, Fei Mei, Benzhe Ning, Jianxiang Zhang, Xinyang Liu, Beirong Zhou, Zeng Shu, Chang Liu, Guang Yang, Zhenyu Han, Jiangben Wang, and Bo Wang. Mindspeed RL: distributed dataflow for scalable and efficient RL training on ascend NPU cluster. *CoRR*, abs/2507.19017, 2025.

[36] Qunhong Zeng, Yuxia Zhang, Zexiong Ma, Bo Jiang, Ningyuan Sun, Klaas-Jan Stol, Xingyu Mou, and Hui Liu. Evaluating generated commit messages with large language models. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 362–373, 2026.

# Appendix

## A Methodology Details

### A.1 Problem Formulation Details

**Rollout Generation.** Given prompt $q \sim \mathcal{D}_k$, the policy samples a rollout $o = (o_1, \ldots, o_{|o|})$ token-by-token from $\pi_\theta(\cdot \mid q)$. We write $o_{<t} = (o_1, \ldots, o_{t-1})$ for the prefix at step $t$. At training iteration $t$, let $\pi_{\theta_{\text{old}}}$ denote the policy before the update.

**Training Batch Construction.** We draw a per-task prompt batch $\mathcal{Q}_k^{(t)} \subseteq \mathcal{D}_k$ of size $|\mathcal{Q}_k^{(t)}| = N_k^{(t)}$, and form the mixed prompt batch $\mathcal{Q}^{(t)} = \bigcup_{k=1}^{K} \mathcal{Q}_k^{(t)}$ of total size $B$, with

$$\sum_{k=1}^{K} N_k^{(t)} = B. \tag{14}$$

For each prompt $q \in \mathcal{Q}_k^{(t)}$, we sample a group of $G$ rollouts from $\pi_{\theta_{\text{old}}}(\cdot \mid q)$ and evaluate rewards with $R_k$.

### A.2 Gradient Compression for Efficient Similarity Computation

Computing cosine similarity on the full gradient vector ($\sim 10^9$ dimensions for a 7B model) is infeasible at training time. Our goal is to detect whether task gradients are aligned or in conflict, rather than to recover their exact directions; we therefore use a compressed representation as an efficient proxy for task relationships. We compress each task gradient via dimension-wise summation: each parameter tensor is reduced along all axes except the last, yielding a row vector that is then concatenated across all layers into a single vector per task. This produces a compressed vector of dimension $D_r \approx 10^6$, achieving approximately $1000\times$ reduction.

The per-task gradients are collected during the existing policy-optimization backward passes, so no additional backward pass is required. The per-step overhead consists of two parts: gradient reduction by element-wise accumulation at $O(K \cdot |\theta|)$ (no matrix multiplications), and $K(K-1)$ pairwise cosine similarities on $D_r$-dimensional vectors at $O(K^2 \cdot D_r)$. The dominant cost remains the forward-backward pass at $O(|\theta| \cdot L \cdot B)$ FLOPs; in our setting ($L$=6144, $B$=128), the overhead ratio is below $0.001\%$.

The approach scales gracefully with the number of tasks $K$. The gradient reduction costs $O(|\theta|)$ per task and thus grows only linearly with K. The pairwise similarity cost $O(K^2 \cdot D_r)$ operates on the compressed vectors; even at $K$=100, this amounts to $\sim 10^{10}$ FLOPs, negligible compared to the $\sim 10^{16}$ FLOPs of a single forward-backward pass.

To verify that the compressed vectors capture task relationships, we compare cosine similarities computed from compressed and full gradients on Qwen2.5-Coder-7B. Since computing full-gradient similarities is expensive, we perform this comparison over the first 80 training iterations, yielding $80 \times 6 = 480$ paired measurements across the six task pairs. The two measures agree in sign for 443 of the 480 comparisons (92.3%), with a Spearman correlation of 0.91 and a mean absolute error of 0.041. The compressed similarities therefore provide nearly the same task-relationship signal as full-gradient similarities at a small fraction of the cost.

### A.3 Utility Estimation and Normalization Details

**EMA Update Rules.** For task-level utilities, we use exponential moving averages with $\alpha = 0.9$:

$$\mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k)\right) = \alpha \cdot \mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k) + (1-\alpha) \cdot \mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{pot}}^{(t-2)}(T_k)\right), \tag{15}$$

$$\mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k)\right) = \alpha \cdot \mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k) + (1-\alpha) \cdot \mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{syn}}^{(t-2)}(T_k)\right). \tag{16}$$

Similarly, for prompt-level utilities:

$$\mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(q, T_k)\right) = \alpha \cdot \mathcal{U}_{\mathsf{pot}}^{(t-1)}(q, T_k) + (1-\alpha) \cdot \mathsf{EMA}_\alpha\left(\mathcal{U}_{\mathsf{pot}}^{(t-2)}(q, T_k)\right), \tag{17}$$

$$\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{prog}}^{(t-1)}(q,T_k)\Big) = \alpha \cdot \mathcal{U}_{\mathsf{prog}}^{(t-1)}(q,T_k) + (1-\alpha)\cdot \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{prog}}^{(t-2)}(q,T_k)\Big). \tag{18}$$

**Normalization Procedures.** We normalize $\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k)\Big)$ to $[0,1]$ using min-max normalization:

$$\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k)\Big) \leftarrow \frac{\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_k)\Big) - \min_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_j)\Big)}{\max_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_j)\Big) - \min_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{pot}}^{(t-1)}(T_j)\Big)}, \tag{19}$$

A small constant $\epsilon = 10^{-8}$ is added to the denominator to prevent division by zero. For $\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k)\Big)$, we normalize to $[-1,1]$:

$$\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k)\Big) \leftarrow 2 \cdot \frac{\mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_k)\Big) - \min_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_j)\Big)}{\max_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_j)\Big) - \min_j \mathsf{EMA}_\alpha\Big(\mathcal{U}_{\mathsf{syn}}^{(t-1)}(T_j)\Big)} - 1. \tag{20}$$

Both normalizations are performed across tasks at each step and are therefore relative: for synergy, the task with the lowest value is mapped to 1 even if all raw cosine similarities are positive, so a negative normalized value indicates relatively low synergy rather than an actual gradient conflict.

**Gradient Estimation.** For each task $T_k$, we compute $\mathbf{g}_k$ by averaging gradients over the current mini-batch. To reduce noise in synergy estimation, we maintain gradient EMAs:

$$\bar{\mathbf{g}}_k(t) = \alpha \cdot \mathbf{g}_k(t) + (1-\alpha)\cdot \bar{\mathbf{g}}_k(t-1). \tag{21}$$

Cross-task synergy is then computed from these smoothed gradients using the compression procedure described in Appendix A.2.

**Quota Rounding.** Since $N_k^{(t)}$ computed via softmax may be non-integer, we round quotas while preserving the total batch size $B$. We use stochastic rounding: for each task, we set $N_k^{(t)} = \lfloor N_k^{(t)} \rfloor$ with probability $1 - (N_k^{(t)} - \lfloor N_k^{(t)} \rfloor)$ and $N_k^{(t)} = \lceil N_k^{(t)} \rceil$ otherwise. We adjust the last task's quota to ensure $\sum_k N_k^{(t)} = B$ exactly.

**Prompt Weight Initialization.** Every prompt starts with a sampling weight of 1, the upper bound of $\sigma(\cdot)$. Once a prompt has been selected and its utility computed, its weight is updated to $\sigma\big(\mathcal{U}^{(t)}(q,T_k)\big) < 1$. Prompts that have not yet been selected therefore keep the largest weight and are preferred for sampling. This encourages coverage of each task's dataset but does not guarantee it.

### A.4 Reward Function Implementation Details

**Code I/O Prediction.** Given a function signature and docstring, the model predicts: (1) output for a given input, or (2) input that produces a specified output.

The reward is defined as:

$$R_{\text{IO}}(o) = \begin{cases} -0.1, & \text{if } o \text{ has wrong format,} \\ Exec(o), & \text{otherwise,} \end{cases} \tag{22}$$

where $Exec(o) \in \{0,1\}$ is an execution-based correctness metric.

*Implementation:* For output prediction, we construct: `assert f(input) == predicted_output`. For input prediction: `assert f(predicted_input) == output`. We execute the assertion in a sandboxed environment (Docker container with resource limits) with a 5-second timeout. If execution succeeds without errors or timeouts, $Exec(o) = 1$; otherwise $Exec(o) = 0$.

**Code Generation.** Given a natural-language problem description, the model generates a complete function implementation.

The reward is:

$$R_{\text{CG}}(o) = \begin{cases} -0.1, & \text{if } o \text{ has wrong format,} \\ PassRatio(o), & \text{otherwise,} \end{cases} \tag{23}$$

where $PassRatio(o) = \frac{\#\text{tests passed}}{\#\text{total tests}}$.

*Implementation:* We execute all provided unit tests against the generated code. Each test is run in isolation with a 10-second timeout. Tests that timeout or raise exceptions are counted as failures. $PassRatio(o)$ is the fraction of tests that pass.

**Unit Test Generation.** Given a focal function and project context, the model generates test cases.

The reward is:

$$R_{\text{UT}}(o) = \begin{cases} -0.1, & \text{if } o \text{ has wrong format,} \\ Coverage(o), & \text{otherwise,} \end{cases} \tag{24}$$

where $Coverage(o) = \frac{LineCov(o)+BranchCov(o)}{2}$.

*Implementation:* We use JaCoCo (Java Code Coverage Library) for coverage measurement. Generated tests are first compiled with `javac`. If compilation fails, $Coverage(o) = 0$. Otherwise, we execute tests with JaCoCo agent enabled, which instruments bytecode to track coverage. $LineCov(o)$ is the percentage of executable lines covered, and $BranchCov(o)$ is the percentage of conditional branches covered. Both metrics are normalized to $[0, 1]$ and averaged.

**Commit Message Generation.** Given a code diff, the model generates a descriptive commit message.

The reward is:

$$R_{\text{CM}}(o) = \begin{cases} -0.1, & \text{if } o \text{ has wrong format,} \\ TextSim(o, gt), & \text{otherwise,} \end{cases} \tag{25}$$

where $gt$ is the ground-truth message and $TextSim(o, gt) = \frac{BLEU(o,gt)+ROUGE(o,gt)+METEOR(o,gt)}{3}$.

*Implementation:* We compute BLEU-4 with smoothing using SacreBLEU. ROUGE-L is computed using the longest common subsequence F1 score from the `rouge-score` library. METEOR is computed using the NLTK implementation with English stemming and WordNet synonyms. All scores are normalized to $[0, 1]$ and averaged.

**Format Checking.** All tasks use a structured prompt template requiring output format: `<think>...</think> <answer>...</answer>`.

We use regular expressions to check format compliance:

```
pattern = r'<think>(.+?)</think>\s*<answer>(.+?)</answer>'
```

If the pattern matches and both tags contain non-empty content, $R_{\text{format}}(o) = 1$; otherwise $R_{\text{format}}(o) = 0$.

The final reward for task $k$ combines task-specific and format rewards:

$$R_k(o) = 0.5 \cdot R_{\text{task},k}(o) + 0.5 \cdot R_{\text{format}}(o), \tag{26}$$

where $R_{\text{task},k} \in \{R_{\text{IO}}, R_{\text{CG}}, R_{\text{UT}}, R_{\text{CM}}\}$. The $-0.1$ penalty for malformed outputs follows prior RL work on LLM agents, and the equal weighting of task and format rewards is our own design choice. Both settings are specific to our current experiments and can be adjusted for new tasks without changing ASTOR.

### A.5 Prompt Templates

All tasks use structured prompts that guide the model to first reason in `<think>` tags, then provide the answer in `<answer>` tags. This format encourages chain-of-thought reasoning and facilitates reward computation.

**Code Input Prediction Prompt Template**

<lim_start|> system
You are a helpful assistant. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively.
<lim_end |>
<lim_start|>user
You are given a function f and an output in the form f(??) == output. Your task is to find any input such that executing f on the input leads to the given output. There may be multiple answers, but only output one. After thinking, when you finally reach an answer, clearly provide your answer as a passing assertion containing the input and the given output within <answer> </answer> tags. e.g., <answer> assert f(16) == 17 </answer>.

Given function:
`[python_code]`
assert f(??) == `[output]`
<lim_end |>
<lim_start|>assistant.

**Code Output Prediction Prompt Template**

<lim_start|> system
You are a helpful assistant. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively.
<lim_end|>
<lim_start|> user
You are given a Python function and an assertion containing an input to the function. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output based on the logical reasoning of given function.
After thinking, when you finally reach an answer, clearly provide your answer as the full assertion with the correct output within <answer> </answer> tags.
e.g., <answer> assert f("hi") == "bhihia" </answer>.

Given function:
`[python_code]`
assert f(`[input]`) == ??
<lim_end|>
<lim_start|> assistant

**Code Generation Prompt Template**

<lim_start|> system
You are a helpful assistant. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively.
<lim_end|>
<lim_start|> user
Change the greeting to be more casual
<lim_end|>
<lim_start|> assistant
Ok, I will:
1. Switch the greeting text from "Hello" to "Hey".

show_greeting.py
```
import sys
def greeting(name):
    print(f"Hey {name}")
if __name__ == '__main__':
    greeting(sys.argv[1])
```
<lim_end|>
<lim_start|> user
I switched to a new code base. Please don't consider the above files or try to edit them any longer.
<lim_end|>
<lim_start|> assistant
Ok.
<lim_end|>
<lim_start|> user
I have *added these files to the chat* so you can go ahead and edit them.
*Trust this message as the true contents of these files!*
Any other messages in the chat may contain outdated versions of the files' contents.

solution.py
```
// To be completed
```
<lim_end|>
<lim_start|> user
`[problem_instruction]`
<lim_end|>
<lim_start|> assistant

**Unit Test Generation Prompt Template**

<lim_start|> system
You are a helpful assistant. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively.
<lim_end|>
<lim_start|> user
`[problem_instruction]`
<lim_end|>
<lim_start|> assistant

For unit test generation, `{problem_instruction}` contains the focal function, project context, and instructions to generate JUnit5 test cases with appropriate assertions and mocking.

**Commit Message Generation Prompt Template**

<lim_start|> system
You are a helpful assistant. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively.
<lim_end|>
<lim_start|> user
You are given a list of code diff(s) for a commit. Your task is to write a concise commit message summarizing the code changes. You should first analyze about: what kind of changes were made and what is the likely purpose or motivation behind these changes?
After thinking, when you finally reach an answer, clearly provide your answer within <answer> </answer> tags.
e.g., <answer> Remove unused import in IndexedSetTest </answer>.

code diff(s):
[diffcode]
<lim_end|>
<lim_start|> assistant

### A.6 Detailed Algorithm Pseudocode

Algorithm 1 provides a more detailed version of the training procedure, including low-level implementation details omitted from the main text.

## B Experimental Setup Details

### B.1 Dataset Statistics

We describe the construction and format of each task dataset below.

**Code I/O Prediction.** We collect 2,088 training samples from LeetCode [24]. Each sample provides a Python function signature along with a docstring, and the model is asked to either predict the output given a specific input, or find a valid input that produces a given output.

**Code Generation.** We collect 2,082 training samples from LiveCodeBench [25] and Codeforces [26]. Each sample contains a natural-language problem description, and the model is required to generate a complete, executable solution.

**Unit Test Generation.** We extract 1,148 Java functions from real-world open-source repositories covering mainstream frameworks (e.g., MyBatis, Spring, Flux), and standardize their test environments to JUnit5 + Mockito to enable executable validation during training. Each sample provides the target function along with its project context, and the model is asked to generate a set of unit tests that achieves high line and branch coverage.

**Commit Message Generation.** We collect 2,446 training samples from the MCMD dataset [27], filtering out messages shorter than 5 words, non-Java code changes, and uninformative entries. Each sample provides a code diff from a real-world commit, and the model is asked to generate a concise and descriptive commit message.

### B.2 Evaluation Benchmarks and Metrics

**CRUXEval.** CRUXEval [28] is a benchmark of 800 short Python functions (3–13 lines) designed to evaluate code reasoning and execution understanding. It consists of two complementary subtasks: *input prediction* (CRUXEval-I), where the model is given a function and an expected output and must find a valid input; and *output prediction* (CRUXEval-O), where the model is given a function and an input and must predict the resulting output. Both subtasks are evaluated by executing the model's answer and checking whether the assertion holds, yielding a binary correctness signal. We

**Algorithm 1** Detailed Training Procedure for ASTOR

**Require:** Tasks $\mathcal{T} = \{T_1, \ldots, T_K\}$, batch size $B$, group size $G$, total iterations $M$
**Require:** EMA coefficient $\alpha$, temperature $\tau$, adjustment rate $\lambda$
**Require:** Base KL coefficients $\{\beta_{\text{base},k}\}_{k=1}^{K}$
Initialize policy $\pi_\theta$, reference policy $\pi_{\text{ref}}$;
Initialize EMA states for task utilities $U_k^{\text{pot}}$, $U_k^{\text{syn}}$ and prompt utilities $U_{q,k}^{\text{pot}}$, $U_{q,k}^{\text{prog}}$;
Initialize gradient EMAs: $\bar{\mathbf{g}}_k(0) \leftarrow 0$ for all $k$
**for** $t = 1$ to $M$ **do**
  *// Phase 1: Utility Estimation*
  **for** each task $T_k$ **do**
    Compute instantaneous utilities $U_k^{\text{pot}}(t-1)$, $U_k^{\text{syn}}(t-1)$ from iteration $t-1$;
    Update $\mathsf{EMA}_\alpha(U_k^{\text{pot}}(t-1))$ and $\mathsf{EMA}_\alpha(U_k^{\text{syn}}(t-1))$;
  **end for**
  Normalize $\{\mathsf{EMA}_\alpha(U_k^{\text{pot}}(t-1))\}$ to $[0,1]$ and $\{\mathsf{EMA}_\alpha(U_k^{\text{syn}}(t-1))\}$ to $[-1,1]$
  *// Phase 2: Task Quota Allocation*
  **for** each task $T_k$ **do**
    Compute combined utility: $U_k(t) \leftarrow \mathsf{EMA}_\alpha(U_k^{\text{pot}}(t-1)) + \mathsf{EMA}_\alpha(U_k^{\text{syn}}(t-1))$;
  **end for**
  Allocate quotas: $N_k^{(t)} \leftarrow B \cdot \frac{\exp(U_k(t)/\tau)}{\sum_j \exp(U_j(t)/\tau)}$ for all $k$;
  Round quotas to integers while preserving $\sum_k N_k^{(t)} = B$;
  *// Phase 3: Prompt Selection*
  **for** each task $T_k$ **do**
    **for** each prompt $q \in \mathcal{D}_k$ **do**
      Update $\mathsf{EMA}_\alpha(U_{q,k}^{\text{pot}}(t-1))$ and $\mathsf{EMA}_\alpha(U_{q,k}^{\text{prog}}(t-1))$;
      Compute $U_{q,k}(t) \leftarrow \mathsf{EMA}_\alpha(U_{q,k}^{\text{pot}}(t-1)) + \mathsf{EMA}_\alpha(U_{q,k}^{\text{prog}}(t-1))$;
    **end for**
    Sample $N_k^{(t)}$ prompts from $\mathcal{D}_k$ using weights $w_{q,k}(t) = \text{sigmoid}(U_{q,k}(t))$ ($w_{q,k}(t) = 1$ if $q$ was never selected) to form prompt batch $\mathcal{Q}_k^{(t)}$;
  **end for**
  Form mixed batch: $\mathcal{Q}^{(t)} \leftarrow \bigcup_{k=1}^{K} \mathcal{Q}_k^{(t)}$;
  *// Phase 4: Rollout Generation and Reward Computation*
  Save old policy: $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$;
  **for** each prompt $q \in \mathcal{Q}^{(t)}$ from task $T_k$ **do**
    Sample $G$ rollouts: $\{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$;
    Compute rewards: $\{r_i = R_k(q, o_i)\}_{i=1}^{G}$;
    Store $(q, \{o_i\}, \{r_i\}, k)$ in batch buffer;
  **end for**
  *// Phase 5: Policy Optimization*
  **for** each task $T_k$ **do**
    Set KL coefficient: $\beta_k(t) \leftarrow \beta_{\text{base},k} \cdot (1 + \lambda \cdot U_k(t))$
  **end for**
  Compute per-task GRPO objectives $\{J_k(\theta; t)\}$ with $\{\beta_k(t)\}$;
  Compute multi-task objective: $J_{\text{MT}}(\theta; t) \leftarrow \sum_k \frac{N_k^{(t)}}{B} J_k(\theta; t)$;
  Update policy: $\theta \leftarrow \theta + \eta \nabla_\theta J_{\text{MT}}(\theta; t)$ (via AdamW);
  *// Phase 6: Utility Update for Next Iteration*
  **for** each task $T_k$ **do**
    Take the gradient $\mathbf{g}_k(t)$ of $J_k(\theta; t)$ from the Phase 5 backward pass;
    Update gradient EMA: $\bar{\mathbf{g}}_k(t) \leftarrow \alpha \mathbf{g}_k(t) + (1-\alpha)\bar{\mathbf{g}}_k(t-1)$;
    Compute $U_k^{\text{pot}}(t)$ from reward variances;
    Compute $U_k^{\text{syn}}(t)$ from $\{\bar{\mathbf{g}}_j(t)\}_{j=1}^{K}$;
  **end for**
  **for** each prompt $q$ sampled at iteration $t$ **do**
    Update $U_{q,k}^{\text{pot}}(t)$ and $U_{q,k}^{\text{prog}}(t)$ from rollout rewards;
  **end for**
**end for**
**Return** $\pi_\theta$;

report *input accuracy* and *output accuracy* as the pass@1 rates on CRUXEval-I and CRUXEval-O, respectively.

**Aider-Polyglot-Python.** Aider-Polyglot-Python [29] is a subset of the Aider polyglot benchmark, consisting of 135 challenging coding exercises drawn from Exercism. Each exercise provides a natural-language specification and a set of hidden unit tests; the model must generate a complete Python solution that passes all tests. The benchmark is specifically designed to remain challenging for frontier models, as it selects only the hardest exercises unsolved by most tested models. We report *pass@1* (the fraction of problems solved on the first attempt) and *pass@2* (the fraction solved within two attempts).

**Defects4J 2.0.** Defects4J 2.0 [30] is a curated collection of 737 reproducible bugs from 17 real-world open-source Java projects, widely used for evaluating automated test generation. Given a target Java method and its project context, the model generates a test suite that is then compiled and executed against the project. We report three metrics: *line coverage* (percentage of executable lines exercised by the generated tests), *branch coverage* (percentage of conditional branches covered), and *compilation rate* (percentage of generated test suites that successfully compile), measured using the JaCoCo coverage tool.

**MCMDEval+.** MCMDEval+ [27] is an evaluation set derived from the MCMD dataset with 500 samples. Given a code diff, the model generates a commit message that is compared against the human-written reference. We report three standard text generation metrics: *BLEU* [31] (n-gram precision with a brevity penalty), *ROUGE-L* [32] (F1 score based on the longest common subsequence), and *METEOR* [33] (harmonic mean of precision and recall with stemming and synonym matching).

### B.3 Baseline Configurations

**Single-Task Learning (STL).** We train a separate GRPO [19] model for each of the four tasks independently. Each specialist is trained with the same hyperparameters as ASTOR, using a task-specific KL coefficient chosen via validation. STL serves as the per-task specialist reference: each model excels on its own task but falls short of the other specialists on their tasks.

**Joint Learning.** Following the experimental setup of Wu et al. [15], we mix all four tasks uniformly within each training batch (equal quota per task) and train a single shared model using GRPO. Wu et al. also compare uniform sampling with sampling tasks in proportion to their gradient magnitudes and find no consistent advantage for the latter, since larger gradients do not reliably indicate larger learning gains. We therefore use uniform mixing as the representative joint-learning baseline. The KL coefficient $\beta$=5e-3 is selected as the best uniform value via preliminary experiments across $\{10^{-4}, 5 \times 10^{-3}, 10^{-2}\}$.

**Curriculum Learning.** Following the curriculum-based MTRL paradigm of OMNI-THINKER [16], we train tasks sequentially in a fixed order: Code I/O $\rightarrow$ Code Generation $\rightarrow$ Unit Test $\rightarrow$ Commit Message. This order follows the software development workflow in Figure 1a (understanding, implementation, testing, documentation) rather than task difficulty. The model is carried forward across stages without resetting.

**Model Merging.** We first train four single-task GRPO specialists (identical to STL above) and select the best checkpoint per task. We then merge the four checkpoints using TIES-Merging [34] with scaling factor $\lambda$=1.0, which resolves parameter sign conflicts and prunes redundant updates before averaging. This baseline represents post-hoc integration without any joint optimization.

### B.4 Hyperparameter Configurations

**Training and RL Settings.** All methods share the same general training configuration: learning rate 5e-7 with cosine scheduler, AdamW optimizer, gradient clip 1.0, global batch size 128, 8 rollouts per sample, rollout temperature 1.0, top-p 1.0, top-k 50, max prompt length 4,096, max response length 2,048, trained for 400 steps on 32 Ascend 910B-B3 NPUs.

**ASTOR-specific Settings.** We use EMA coefficient $\alpha$=0.9 for utility estimation at both task and prompt levels to smooth noisy per-batch statistics. For task allocation, we set softmax temperature $\tau$=1.0: lower values (e.g., $\tau$=0.5) over-concentrate on the highest-utility tasks and starve others; higher values (e.g., $\tau$=2.0) approach uniform sampling and fail to exploit utility differences. For dynamic KL regularization, we set adjustment rate $\lambda$=0.2, which is robust across a moderate range ($\lambda \in [0.1, 0.4]$); too small approximates static KL, while too large suppresses beneficial policy updates.

**Task-Specific KL Base Coefficients.** We assign $\beta$=$10^{-2}$ to tasks that must output complete executable programs (code generation, unit test) to preserve syntactic correctness, and $\beta$=$10^{-4}$ to tasks whose outputs are short answers or natural language (code I/O, commit message) to allow expressive reasoning. For baselines requiring a uniform $\beta$, we select the best value via preliminary experiments on Qwen2.5-Coder-7B (Table 4).

Table 4: Effect of uniform KL coefficient $\beta$ on Joint Learning (Qwen2.5-Coder-7B). No single value resolves the conflict between task types; $\beta$=5e-3 is the best compromise and is used for all uniform-$\beta$ baselines.

| $\beta$ | Code I/O (In/Out) | Code Gen (P@1/P@2) | Unit Test (Ln/Br/Cmp) | Commit Msg (B/R/M) | Avg |
|---|---|---|---|---|---|
| 1e-4 | 51.5/53.4 | 25.9/37.0 | 19.4/16.2/41.5 | 33.8/20.4/18.6 | 31.77 |
| **5e-3** | **51.4/54.0** | **32.6/43.0** | **25.9/21.0/51.1** | 30.7/17.1/16.0 | **34.28** |
| 1e-2 | 47.5/49.0 | 37.0/43.7 | 25.2/23.2/53.2 | 25.6/13.4/12.2 | 33.00 |

$\beta$=$10^{-4}$ favors commit message generation but severely hurts code generation and unit test generation; $\beta$=$10^{-2}$ does the opposite. Since all baselines use their respective best-performing KL coefficients, the consistent gains of ASTOR confirm that improvements stem from the scheduling and dynamic regularization modules, not from the base KL configuration.

# C Additional Experimental Results

## C.1 Semantic Evaluation of Commit Messages

To complement the lexical-overlap metrics, we evaluate the generated commit messages on MCMDE-val+ with an LLM judge following Zeng et al. [36]. We use GPT-4 under the chain-of-thought (CoT) and few-shot setting, which Zeng et al. report as achieving the strongest agreement with human judgment. The judge receives the code diff and the generated message and assigns a score from 0 to 4 based on the two criteria of Zeng et al.: whether the message correctly describes *what* changed, and whether it correctly conveys *why* the change was made. A score of 4 requires both to be accurate and complete, and a score of 0 indicates that the message is unrelated to the diff. Table 5 reports the average score.

ASTOR achieves the highest score on both base models, ahead of the commit-message specialist and all multi-task baselines, so its improvement holds beyond lexical overlap. On Qwen2.5-Coder-7B, STL (Commit Msg) attains a higher BLEU than ASTOR (41.2 vs. 35.2; Table 1) yet a lower judge score (2.88 vs. 3.08), indicating that a higher overlap with the reference does not necessarily yield a more accurate description of the change. Table 8 in Appendix D shows a representative example.

# D Qualitative Examples

We compare the outputs of Joint Learning and ASTOR on representative examples of code output prediction, code generation, and commit message generation, respectively.

# E Limitations and Future Work

**Task and Language Coverage** The current experiments cover four coding tasks using Python and Java. While these tasks represent a reasonable cross-section of the development workflow, we have not yet validated ASTOR on other software engineering tasks such as code review or bug localization,

Table 5: LLM-as-a-judge scores (0–4) for commit message generation on MCMDEval+. Bolded values mark the best per column; underscored values mark the second best.

| Method | Qwen2.5-Coder-7B | Qwen3-8B |
|---|---|---|
| Base Model | 1.62 | 2.05 |
| STL (Code I/O) | 2.06 | 2.25 |
| STL (Code Gen) | 2.10 | 2.29 |
| STL (Unit Test) | 1.86 | 2.10 |
| STL (Commit Msg) | <u>2.88</u> | <u>2.93</u> |
| Joint Learning | 2.18 | 2.50 |
| Curriculum Learning | 2.22 | 2.42 |
| Model Merging | 2.26 | 2.47 |
| ASTOR | **3.08** | **3.20** |

Table 6: Comparison of Output Prediction. Baseline answer in **red**; ASTOR answer in **green**.

**Prompt**

Given a Python function and an assertion, complete the assertion with the output.

**Function:**

```
def f(text):
text = text.upper()
count_upper = 0
for char in text:
if char.isupper():
count_upper += 1
else:
return 'no'
return count_upper // 2
```

```
assert f('ax') == ??
```

**Model Responses**

**Baseline (Joint Learning):**

<think> The function returns 'no' if any character is not uppercase. Input 'ax' has lowercase letters. </think>
**<answer> assert f('ax') == 'no' </answer>**

**ASTOR:**

<think> 'ax'.upper() = 'AX'. Both chars are uppercase, count_upper = 2, return 2 // 2 = 1. </think>
**<answer> assert f('ax') == 1 </answer>**

nor on a wider set of programming languages. Extending ASTOR to more tasks and languages is a straightforward next step. Moreover, our setting focuses on single-turn tasks with verifiable rewards. Agentic scenarios such as GUI agent tasks involve multi-turn interaction and tool use, which our current setup does not cover. Domains outside coding, such as mathematical reasoning or general reinforcement learning from human feedback (RLHF), are likewise beyond the scope of this work. Extending utility-driven coordination to these settings is an important direction for future work.

**Reward Quality for Generative Tasks** For commit message generation, the reward is computed as an average of BLEU, ROUGE-L, and METEOR against reference messages. These metrics are convenient and verifiable, but they measure surface-level lexical overlap rather than semantic correctness or usefulness. Using a stronger reward signal for such tasks, for example, an LLM-based evaluator, could improve training quality and is worth exploring in future work.

**Manual Setting of KL Base Coefficients** The task-specific KL base coefficients $\beta_{\text{base},k}$ are set manually based on task type, with a higher value for code-generation tasks and a lower value for text-generation tasks. This works well in the current setting, but requires prior knowledge about each task's optimization characteristics. When applying ASTOR to new tasks, these coefficients would need to be re-tuned, and automating this process is left for future work.

Table 7: Comparison of Code Generation. Baseline answer in **red**; ASTOR answer in **green**.

**Prompt**

Implement a binary search algorithm in `binary_search.py`. Use the `raise` statement to throw a `ValueError` when the value is not found.

**Model Responses**

**Baseline (Joint Learning):**

<think> Implement binary search returning -1 if not found. </think>
**<answer>**

```
def find(search_list, value):
low = 0
high = len(search_list) - 1
while low <= high:
mid = (low + high) // 2
if search_list[mid] == value:  return mid
elif search_list[mid] > value:  high = mid - 1
else:  low = mid + 1
return -1  # wrong: should raise ValueError </answer>
```

**ASTOR:**

<think> Implement binary search raising ValueError when not found. </think>
**<answer>**

```
def find(search_list, value):
low = 0
high = len(search_list) - 1
while low <= high:
mid = (low + high) // 2
if search_list[mid] == value:  return mid
elif search_list[mid] > value:  high = mid - 1
else:  low = mid + 1
raise ValueError("value not in array") </answer>
```

Table 8: Comparison of Commit Message Generation. Baseline answer in **red**; ASTOR answer in **green**.

**Prompt**

Given the code diff below, write a concise commit message.

```
diff -git a/.../CoinmateAdapters.java
@@ -186,6 +186,7 @@ public class CoinmateAdapters {
case "COMPLETED":  status = COMPLETE; break;
+   case "NEW":
case "SENT":
case "CREATED":
case "WAITING":
```

**Model Responses**

**Baseline (Joint Learning):**

<think>A new case for "NEW" status is added.</think>
**<answer> Add NEW status to CoinmateAdapters </answer>**

**ASTOR:**

<think>The commit adds support for a new funding record status "NEW" in CoinmateAdapters.</think>
**<answer> Add support for NEW funding record status </answer>**

## F Broader Impacts

**Positive Impacts.** ASTOR enables a single LLM to replace multiple task-specific RL specialists across the software development workflow, directly reducing memory and compute costs for AI-assisted coding deployments. By improving model efficiency and multi-task capability, it lowers the barrier for practitioners and organizations to adopt capable coding assistants, potentially accelerating software development and improving code quality at scale.

**Negative Impacts.** Stronger automated code generation capabilities could be misused to produce malicious code, exploit software vulnerabilities, or automate cyberattacks. We acknowledge this dual-use risk and note that responsible deployment of ASTOR-trained models should be accompanied by appropriate access controls and output filtering, consistent with the policies of the underlying base models (Qwen2.5-Coder and Qwen3).

## G Extended Related Work

**Multi-Task Learning Foundations.** Multi-task learning (MTL) [10] is the paradigm of training a single model on multiple related tasks simultaneously, leveraging shared structure to act as an inductive bias that improves generalization beyond what single-task models can achieve. In deep learning, MTL is typically implemented via hard parameter sharing, where all tasks share hidden layers with task-specific output heads, or soft parameter sharing, where tasks maintain private parameters regularized toward a common solution. In multi-task RL, representative architectures extend this with task-specific modules and routing networks, such as Soft Modularization [12], which learns to compose shared modules dynamically, and approaches sharing experience via auxiliary tasks [11] to enrich representation learning. A recurring finding across all these settings is that naïve joint optimization creates dominance effects: tasks with larger gradients or shorter time horizons tend to monopolize training, degrading performance on others. ASTOR operates in this same multi-task RL regime but specifically targets LLM post-training, where non-stationary online rollouts, heterogeneous reward semantics, and per-token KL regularization introduce qualitatively new challenges.

**Gradient Manipulation in Multi-Task Learning.** A prominent class of methods addresses task interference by modifying gradient directions or magnitudes at each update step. PCGrad [20] detects conflicting task gradients (negative cosine similarity) and projects each gradient onto the normal plane of conflicting gradients, preventing destructive interference. CAGrad [22] takes a principled optimization-theoretic view: it minimizes average loss while leveraging the worst local per-task improvement as a regularizer, provably converging to an average-loss minimum while outperforming PCGrad and MGDA on multi-task supervised and RL benchmarks. GradDrop [21] probabilistically masks gradients at activation layers based on a sign-purity score, allowing gradients to compete while provably converging near joint local minima of all task losses. While these methods are effective at gradient-level alignment, they operate entirely within a single update step and do not control which data is presented during that step, nor do they adapt the trust-region constraint (e.g., KL coefficient) to each task's training state. ASTOR's utility signal uses gradient cosine similarity as one component of cross-task synergy, but combines it with learning-potential signals to drive both data scheduling and adaptive KL regularization—decisions that gradient manipulation methods leave unaddressed.

**Multi-Task RL for LLMs.** With the rise of RL-based LLM post-training, several recent works have tackled multi-task settings. Wu et al. [15] provide a systematic empirical analysis showing that naïve gradient mixing in multi-task RL causes severe imbalance: tasks with inherently larger gradient norms dominate optimization regardless of their actual learning gains, a phenomenon not captured by standard training statistics such as reward or advantage. Their findings motivate principled gradient-level corrections for LLMs post-training. OMNI-THINKER [16] addresses cross-task forgetting using backward-transfer-guided task scheduling, where task sampling probabilities are adjusted according to backward transfer scores measured across previously encountered tasks; it also combines rule-based and preference-based reward signals in a hybrid supervision scheme. While OMNI-THINKER introduces curriculum scheduling for LLM MTRL, its scheduling signal is based on backward transfer from a fixed task ordering rather than on dynamic real-time gradient interactions between tasks. ASTOR estimates cross-task synergy online at each training step via gradient cosine

similarity, enabling the scheduler to react immediately to changes in task relationships as training progresses.

**RL Post-Training for Code LLMs.** RL with verifiable execution rewards has become the dominant paradigm for post-training code LLMs [19, 5]. DeepSeek-R1 [5] demonstrates that reinforcement learning with verifiable rewards can induce emergent reasoning behaviors such as self-reflection and verification in LLMs without requiring supervised rationale data. Kimi k1.5 [7] scales RL training with long-context rollouts and curriculum learning over problem difficulty. RLEF [8] specifically targets code synthesis by training LLMs on multi-turn execution feedback, dramatically reducing the sample count needed relative to best-of-$N$ strategies. CodeRL+ [6] augments RL for code generation with execution semantics alignment, enabling models to reason about variable-level execution trajectories to achieve stronger generalization. Lee et al. [9] apply adversarial RL to unit test generation, using an adversarial process to improve branch and statement coverage. All of these approaches train task-specific specialists, and as shown in Figure 1b, each model degrades sharply outside its own training task. ASTOR addresses this limitation by co-training a single model across all four coding tasks under a utility-guided multi-task RL framework.

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We have clearly stated the claims made in the abstract and introduction, accurately reflecting the paper's contributions and scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have discussed the limitations of our work in the Appendix E, which include task and language coverage, reward quality for generative tasks, and the task-specific KL base coefficients.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: Our paper does not include theoretical results, and therefore, this question is not applicable to our work.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide a detailed description of the model and experimental settings in our paper, ensuring that readers have the necessary information to reproduce the main experimental results. Additionally, we plan to release the code to further enhance reproducibility and facilitate verification of our results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer:[No]

   Justification: We do not include the data and code with this version of the paper; we plan to release the code along with sufficient instructions to reproduce the main experimental results.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We have provided all the necessary details regarding the training and testing process. Complete hyperparameter tables (including $\tau$, $\lambda$, task-specific KL base values, learning rate, etc.), baseline configurations, dataset splits, and evaluation metric definitions are provided in the Appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: We conducted our experiments and baseline experiments on the same training and testing datasets to ensure a fair comparison.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provided sufficient information on the computer resources needed to reproduce the experiments in the implementation details section.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics. The paper studies multi-task reinforcement learning for code LLMs using publicly available or collected open-source datasets; it involves no human subjects and raises no specific ethical concerns.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: We discussed the social impact of this work in Appendix F, covering reduced deployment costs from unifying multiple specialists, and the dual-use risk of stronger automated code generation being misused for malicious purposes.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: No such risks.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All datasets, benchmarks, and base models used in this work are publicly available and are cited in Section 3.1 and Appendix B; we use them in accordance with their licenses (e.g., Qwen2.5-Coder-7B and Qwen3-8B are released under the Apache 2.0 license).

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: This paper does not release new assets. We plan to release well-documented code for ASTOR.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Our research does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: Our research does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: The core method development in our research does not involve LLMs as any important, original, or non-standard components.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.