TRAIN WITH HINDSIGHT: POST-HOC CHECKPOINT SELECTION ATOP DETERMINISTIC COMPUTING

Anonymous authors
Paper under double-blind review

ABSTRACT

The training of large language model(LLM) involves substantial steps, yet the optimal checkpoint for evaluation or next training stage can exist inside the training trajectory rather than at its endpoint. Fixed-interval checkpointing couples the set of selectable states to a priori what to save before that trajectory is observed, while denser storage and evaluation increase checkpoint I/O, storage, and validation costs. We introduce Post-hoc Checkpoint Selection (PhCS), a "*train with hindsight*" framework that decouples checkpoint selection from physical checkpoint storage. In the original training, PhCS saves parameters and optimizer states periodically as anchors and records lightweight trajectory metadata and input per step, after which post-hoc selection policies identify candidate states on the realized trajectory. Atop deterministic computation, reproduction of candidate states begins from the nearest preceding anchor and replay the previous input along the same numerical path. We instantiate PhCS in SFT-RL pipeline: static selection in SFT covers the entire trajectory, combining loss, token accuracy, and entropy, whereas dynamic selection in GRPO conducts recursive exploration on validation set. PhCS thereby turns checkpointing from a precommitted storage schedule into a post-hoc decision over the realized training trajectory. We evaluate PhCS in ReTool framework, a representative SFT-Agentic RL scenario; the experiments demonstrate that PhCS achieves higher pass@4 than conventional training paradigms on multiple datasets such as AIME24/25/26 and AMC23 , and achieves superior equivalent storage efficiency.

1 INTRODUCTION

Large language model post-training commonly combines supervised fine-tuning (SFT) with reinforcement learning (RL) to establish task behavior and then optimize by reward signals(Christiano et al., 2017; Ouyang et al., 2022; Bai et al., 2022; Schulman et al., 2017). In such multi-stage pipelines, the optimal state for downstream evaluation or subsequent training may not coincide with the final training step or interval checkpoints. As illustrated in Figure 1, the state with highest reward can occur at a middle step, which is proved with a 100% accuracy in subsequent evaluation. In the case the optimal state can exist in neither a fixed-interval checkpoint nor the final checkpoint. This case demonstrates why checkpoint value may become apparent only after the realized training trajectory is available.



Figure 1: GRPO training reward of Ministral-3-8B-Instruct-2512 on Sudoku[1]. The highest-reward step is neither a fixed-interval saved checkpoint nor the final checkpoint.

Existing training systems typically select from checkpoints saved at fixed intervals, at the end of training, or when an early-stopping is triggered. Increasing the saving frequency expands the set of selectable checkpoints, but incurs additional storage and checkpoint I/O costs; evaluating more checkpoints also increases validation cost. Conversely, sparse checkpointing is efficient but prevents

[1]Training Recipe: https://unsloth.ai/docs/zh/mo-xing/tutorials/ministral-3

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

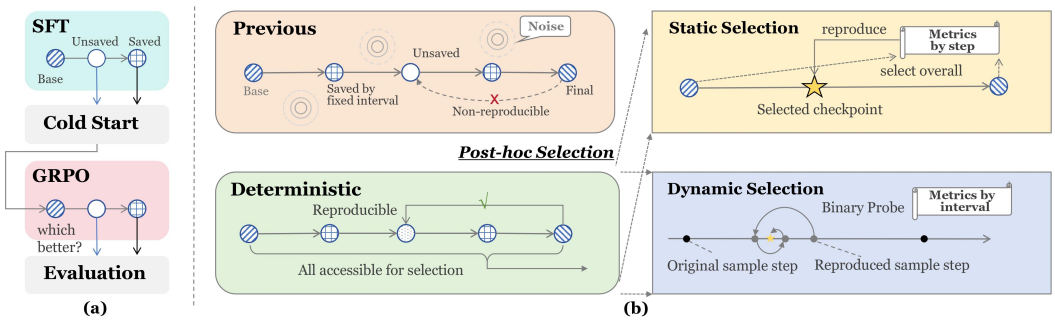

Figure 2: (a) Conventional training paradigms may miss better intermediate checkpoints because they are not saved. (b) PhCS builds the training system atop deterministic computing, enabling precise reproduction of unsaved checkpoints and post-hoc selection of candidates to reproduce.

unsaved intermediate states from being considered. This creates a fundamental coupling between checkpoint storage and checkpoint selection: the states that can be selected are restricted by what was saved during training.

We propose **Post-hoc Checkpoint Selection (PhCS)**, a "*train with hindsight*" paradigm that decouples checkpoint storage from checkpoint selection. As shown in Figure 2, instead of storing every potentially useful state, PhCS stores periodic full-state anchors and lightweight step-level trajectory records during the original training run. After the trajectory is observed, a post-hoc selection procedure identifies candidate steps from the recorded trajectory. Selected candidates can then be reproduced from the nearest preceding anchor, allowing PhCS to materialize checkpoints that were not stored during the original run. Importantly, expanding the set of recoverable candidate steps does not require evaluating every step in the trajectory.

The key enabler of PhCS is **deterministic computation**. Reproducing an unsaved checkpoint requires the replayed updates to follow the same numerical path as the original training run. PhCS therefore imposes deterministic computation across the training and rollout paths, including fixed reduction order, aligned computation kernels, data order, random state, and parallel configuration(Whitehead & Fit-Florea, 2011; Yuan et al., 2025; He & Thinking Machines Lab, 2025; Zhang et al., 2026). Under this numerical contract, a candidate step on the recorded trajectory can be reproduced from an anchor and used for downstream evaluation or continued training. We instantiate PhCS in a cold-start SFT-RL pipeline: SFT uses lightweight per-step training metrics for static selection, while GRPO uses validation results for dynamic selection over the recorded trajectory.

Our contributions are as follows:

- We propose PhCS, a post-hoc checkpoint selection paradigm built atop deterministic computing. It transforms checkpoint selection from an a priori decision into a post-hoc one and we empirically demonstrates its effectiveness in improving model performance.
- We implement a deterministic training system that supports full fine-tuning, Low-Rank Adaptation (LoRA) (Hu et al., 2021), Fully Sharded Data Parallel (FSDP) (Zhao et al., 2023) and Activation Checkpointing, with availability under resource constraints and compatibility for mainstream open-source model architectures such as Qwen3/3.5 and Ministral-3.
- We instantiate PhCS across a SFT-agentic RL pipeline, with stage-specific selection policies for supervised fine-tuning and agentic reinforcement learning, verifying the feasibility of integrating PhCS into universal training procedures.

2 Related Work

2.1 SFT and RL in LLM Post-Training

Supervised fine-tuning (SFT) followed by reinforcement learning (RL) is a common recipe for LLM post-training. SFT provides behavioral and instruction-following capabilities, while RL further optimizes the policy using human, AI, rule-based, or task outcome rewards (Christiano et al., 2017;

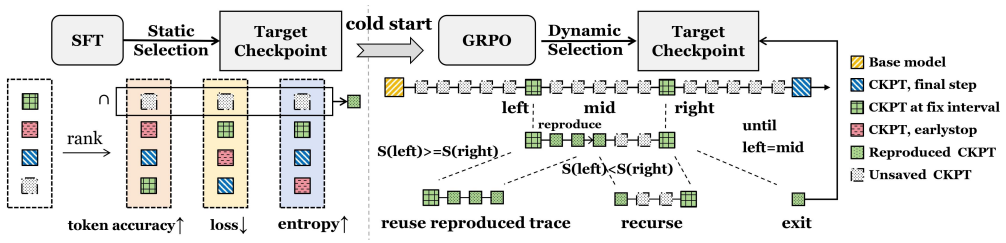

Figure 3: Method overview of PhCS. PhCS designs two selection policies based on the granularity of training-time metrics: For metrics with a step-level granularity, such as loss, entropy, and token accuracy in SFT, PhCS can directly reproduce the optimal checkpoint(or CKPT) within the selection policy; for metrics that are measured at fixed intervals, such as the performance on the GRPO validation set, PhCS recursively reproduces and probes the checkpoints by a binary-style interval-refinement heuristic.

Ouyang et al., 2022; Bai et al., 2022; Schulman et al., 2017; Shao et al., 2024; Guo et al., 2025; Liu et al., 2026b; Feng et al., 2025). Recent work shows that the SFT–RL handoff affects subsequent RL performance: excessive SFT can reduce plasticity, while insufficient SFT may leave the policy under-prepared for RL (Liu et al., 2026b; Li et al., 2026). Agentic and tool-use pipelines similarly use SFT to establish interaction formats before RL optimizes tool use and reasoning behavior (Wei et al., 2022; Chen et al., 2023; Gao et al., 2023; Pan et al., 2023; Wang et al., 2023; Luong et al., 2024; Jin et al., 2025; Li et al., 2025; Feng et al., 2025). We study checkpoint selection in this SFT-agentic RL scenario.

2.2 Checkpoint Selection

Checkpoint selection is typically performed over a sparse set of stored checkpoints, through fixed evaluation intervals, final-checkpoint selection, or early stopping (Prechelt, 2002). These approaches couple the candidate set to checkpoints stored during training. This limitation is particularly relevant to RL fine-tuning, where checkpoint quality can vary substantially and validation is costly (Christiano et al., 2017; Cobbe et al., 2021; Lightman et al., 2023; Stiennon et al., 2020; Nguyen et al., 2025). Recent uncertainty-guided checkpoint selection (UGCS) reuses per-sample rewards and log-likelihoods from RL logs to score fixed stored checkpoints (Gal & Ghahramani, 2016; Ren et al., 2019; Swayamdipta et al., 2020; Farquhar et al., 2024; Nguyen et al., 2025). PhCS is complementary: rather than changing how fixed stored checkpoints are scored, it decouples checkpoint selection from full-state checkpoint storage by allowing candidate steps from the recorded trajectory to be selected and subsequently reproduced.

2.3 Deterministic Inference and Training

Deterministic computation is increasingly important in LLM evaluation, inference, and training pipelines, where numerical differences can alter generations, rewards, and subsequent updates. Prior work identifies floating-point non-associativity, batching, kernel implementations, training–inference mismatch, and tensor-parallel reductions as major sources of nondeterminism (Whitehead & Fit-Florea, 2011; Yuan et al., 2025; He & Thinking Machines Lab, 2025; Zhang et al., 2026). Recent systems address these issues with deterministic kernels and batch- or tensor-parallel-invariant computation (Kwon et al., 2023; The SGLang Team, 2025; Shoeybi et al., 2019; He & Thinking Machines Lab, 2025; Yuan et al., 2025; Zhang et al., 2026). PhCS builds on this line of work by extending deterministic computation from inference consistency to training trajectory reproduction: a selected candidate step can be reproduced from an anchor and evaluated after the original training.

3 Method

Training produces a state at every step, but the conventional training paradigm can access only the states retained by an interval-based saving schedule. PhCS operates in two phases: the original run

stores sparse anchors and step-level trajectory information, and replay runs later reconstruct selected states. We first formalize this setting and then describe reproduction, selection, and implementation.

3.1 Definition and Goal

Let the original training trajectory be $\mathcal{T} = \{z_t\}_{t=0}^{T}$, where z_t is the full training state at step t. We store $z_t \subseteq (\theta_t, \omega_t, \xi_t)$, comprising model parameters θ_t, optimizer state ω_t, and additional trainer state ξ_t required by the next update. Let $\mathcal{A}$ denote anchors written to persistent storage, $\mathcal{R}$ the reproducible candidate states, and $\mathcal{E}$ the states actually materialized and evaluated:

$$\mathcal{A} \subseteq \mathcal{R} \subseteq \mathcal{T}, \qquad \mathcal{E} \subseteq \mathcal{R}. \tag{1}$$

These sets distinguish physical storage, reproduction capability, and the subset chosen for materialization and evaluation. Conventional stored-checkpoint workflows satisfy $\mathcal{R} = \mathcal{A}$ and $\mathcal{E} \subseteq \mathcal{A}$, because an unsaved state cannot be materialized for evaluation. When every stored checkpoint is evaluated, $\mathcal{E} = \mathcal{A}$; otherwise, $\mathcal{E}$ is the evaluated subset of $\mathcal{A}$.

Goal. Against this conventional setting, PhCS selects from $\mathcal{R}$ rather than only $\mathcal{A}$, without writing every training state as a full checkpoint. Given trajectory signals $I_{\mathcal{T}}$ and evaluations on validation sets for states in $\mathcal{E}$, a selection policy π proposes

$$\hat{t} = \pi(I_{\mathcal{T}}, \{S(t) : t \in \mathcal{E}\}), \qquad z_{\hat{t}} \in \mathcal{R}. \tag{2}$$

The design goal is step-level access to the recorded trajectory. When the reproduction conditions in Section 3.3.1 hold throughout the trajectory, PhCS yields $\mathcal{R} = \mathcal{T}$, so any training step can be materialized as a checkpoint. Because $\mathcal{E}$ remains a policy-dependent subset of $\mathcal{R}$, recovery granularity and evaluation cost remain independently controllable.

3.2 Key Idea

Accessing an unsaved training state requires re-executing its intervening updates, but ordinary parallel GPU execution may not reproduce the same state. Floating-point reductions are non-associative, so different accumulation orders can change an update and propagate differences to later steps (Whitehead & Fit-Florea, 2011; Yuan et al., 2025; He & Thinking Machines Lab, 2025). PhCS therefore fixes the execution configuration, including kernel implementations, reduction order, data order, random state, and parallel topology, so that each update becomes deterministic given its starting state and recorded input. A full-state anchor provides the starting state, step-level trajectory information provides the intervening inputs, and the selection policy specifies which candidate step to reproduce.

However, deterministic inference alone is insufficient for PhCS: reproduction must replay the forward and backward passes as well as the parameter, gradient, and optimizer update. The rollout and training paths must also remain consistent despite their different model-loading and kernel semantics. We therefore extend fixed-reduction kernels from inference to training and align model loading and computation semantics across rollout and training workers. These changes establish end-to-end training-state reproduction, which is necessary to **extend selection beyond physically stored checkpoints to a broader set of reproducible candidate states.**

3.3 Framework Design

The detailed design realizes the core idea through deterministic training, reproduction, and selection. Deterministic State Recovery establishes whether a requested state can be reproduced from the original run. Checkpoint Reproduction materializes that state from a full-state anchor, and Selection Policies decide which states warrant this cost.

3.3.1 Deterministic Training System

Kernel-level determinism. Kernel-level determinism keeps the transition F_e fixed at reduction-related computation. Floating-point reductions are non-associative, and GPU kernels can alter accumulation through tiling or split-K partitioning (Whitehead & Fit-Florea, 2011; Yuan et al., 2025; He & Thinking Machines Lab, 2025). Atomic updates and launch scheduling introduce further order

variation. For a tiled matrix multiplication, we fix the reduction partition and accumulate tiles in a fixed sequential order: $C_{ij} = \sum_{p=1}^{P_{\text{tile}}} \sum_{k \in \mathcal{K}_p^{\text{BLOCK_K}}} A_{ik} B_{kj}$. Here $\mathcal{K}_p^{\text{BLOCK_K}}$ is the reduction-index subset induced by the fixed tile size, and P_{tile} is the number of tiles. Furthermore, other reduction-related kernels we have also modified in the same way include functions such as softmax and attention. Fixing both the partition and accumulation order removes scheduler-dependent accumulation order within the validated kernel path.

Training-Inference Alignment Reproducibility also depends on training and inference exposing the same computation semantics(Zhang et al., 2026). PhCS therefore uses shared computation kernels on the rollout and training backends instead of separate implementations. To keep rollout and training aligned, the rollout and training backends are linked through compatible model loading, so the parameters consumed by one backend are equivalent exactly to the parameters produced by the other. This makes rollout generation, loss computation, and checkpoint reproduction operate on the same effective model state. With fixed data, seed, and optimizer state, the update at each step is deterministic, which allows PhCS to observe the original trajectory, select checkpoints by post-hoc policies, and later reproduce the candidates.

3.3.2 Checkpoint Reproduction

PhCS stores anchor parameters and optimizer states at a fixed interval, and records the data input at step-level granularity. These records preserve the information needed for reproduction later, including the data input, hyperparameters, and the metrics used by post-hoc selection policies. When a candidate step is selected, PhCS reproduces the checkpoint from the nearest preceding checkpoint. Reproduction runs the updates under the same deterministic computation settings and with the same input. The reproduced weights therefore correspond to the selected step in the original training.

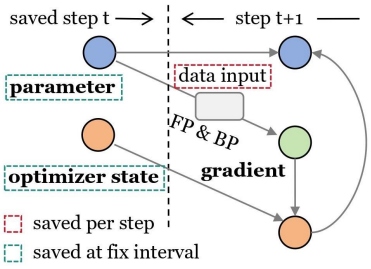

Figure 4: Dataflow graph of parameters, gradient, and optimizer state in reproduction.

Formally, let one original-run update under fixed execution configuration e be $z_t = F_e(z_{t-1}, u_t)$. Here, u_t denotes the ordered, step-specific input consumed by the update and recorded in the step-level trajectory information. A replay run starts from the nearest preceding anchor $a(t)$ and applies the same transition to obtain $\tilde{z}_t$:

$$\tilde{z}_{a(t)} = z_{a(t)}, \quad \tilde{z}_j = F_e(\tilde{z}_{j-1}, u_j), \quad j = a(t)+1, \ldots, t. \tag{3}$$

Reproduction first requires a full-state anchor and every intervening update input. It also requires all trainer state affecting F_e and reuse of the same fixed execution configuration. Under these conditions, induction over j gives $\tilde{z}_j = z_j$ for every step in the reproduced segment, rather than only at its endpoint. If these conditions hold for every segment, every state on the realized training trajectory is reproducible and $\mathcal{R} = \mathcal{T}$. Thus, any training step in the covered trajectory can be materialized as a checkpoint for evaluation or continued training. If an update input is missing or the execution configuration changes, the reproduction may diverge, and this guarantee no longer applies.

3.3.3 Selection Policies

With reproducibility separated from storage, selection policies can match the signal granularity available at each training stage. PhCS instantiates two policies that share the same deterministic training mechanism. Static Selection uses signals recorded at every SFT step, whereas Dynamic Selection probes additional GRPO states because validation scores are observed only at intervals. Both policies select from $\mathcal{R}$, and a requested state enters $\mathcal{E}$ only after materialization and evaluation.

Static Selection for SFT. Static Selection first filters recoverable SFT steps by loss and token accuracy, then uses mean entropy to rank the remaining candidates. For each step $t \in \mathcal{C}_{\text{SFT}} \subseteq \mathcal{R}$, let ℓ_t, a_t, and H_t denote loss, token accuracy, and mean entropy. For a fraction $p \in (0, 1]$, define

$$\mathcal{L}_p = \{t \in \mathcal{C}_{\text{SFT}} : \ell_t \text{ in the lowest } p \text{ fraction}\}, \mathcal{Q}_p = \{t \in \mathcal{C}_{\text{SFT}} : a_t \text{ in the highest } p \text{ fraction}\}, \tag{4}$$

Algorithm 1 Static Selection for SFT

Require: $\mathcal{C}_{\mathrm{SFT}}$, $\{(\ell_t, a_t, H_t)\}$, P, δ
Ensure: $\hat{t}_{\mathrm{SFT}}$
$p \leftarrow P$
while $p \leq 1$ **do**
  $\mathcal{L}_p \leftarrow \mathrm{Low}_p(\ell)$
  $\mathcal{Q}_p \leftarrow \mathrm{High}_p(a)$
  $\mathcal{S} \leftarrow \mathcal{L}_p \cap \mathcal{Q}_p$
  if $\mathcal{S} \neq \emptyset$ **then**
    return $\arg\max_{t \in \mathcal{S}} \mathrm{key}(t)$
  end if
  $p \leftarrow \min(p + \delta, 1)$
end while

Algorithm 2 Dynamic Selection for GRPO

Require: $\mathcal{C}_{\mathrm{GRPO}}$, $[l_0, r_0]$, S
Ensure: $\hat{t}_{\mathrm{GRPO}}$
Validate $l_0, r_0 \in \mathcal{C}_{\mathrm{GRPO}}$, $l_0 < r_0$
$[l, r] \leftarrow [l_0, r_0]$
while $r - l > 1$ **do**
  $m \leftarrow \lfloor (l + r)/2 \rfloor$
  Reproduce/retrieve l, m, r; cache $[l, m]$
  Evaluate $S(l), S(m), S(r)$ if unavailable
  $[l, r] \leftarrow [l, m]$ **if** $S(l) \geq S(r)$ **else** $[m, r]$
  Discard cached checkpoints outside $[l, r]$
end while
return m

and write $\mathrm{Low}_p(\ell) = \mathcal{L}_p$, $\mathrm{High}_p(a) = \mathcal{Q}_p$, with the lexicographic key $\mathrm{key}(t) = (H_t, -\ell_t, a_t, t)$. Starting from $p = P$, the policy increases p by δ until $\mathcal{L}_p \cap \mathcal{Q}_p \neq \emptyset$, and stops at the first feasible intersection. It then selects

$$\hat{t}_{\mathrm{SFT}} = \arg \max_{t \in \mathcal{L}_p \cap \mathcal{Q}_p} \mathrm{key}(t), \tag{5}$$

where the key is compared lexicographically. The entropy-first order is a configurable, stage-specific heuristic intended to retain policy diversity before RL. After entropy, the key prefers lower loss and then higher accuracy. A later step resolves any remaining tie, without implying optimality. Algorithm 1 summarizes the procedure.

Dynamic Selection for GRPO. GRPO presents a coarser signal regime because validating every recoverable candidate state would make evaluation cost scale with $|\mathcal{R}|$. Dynamic Selection therefore uses a cached interval-refinement heuristic over a user-specified initial interval. Let $\mathcal{C}_{\mathrm{GRPO}} \subseteq \mathcal{R}$ be the GRPO candidate steps, and let $S(t)$ be the validation score of step t. PhCS takes an initial selection interval $[l_0, r_0]$ with $l_0, r_0 \in \mathcal{C}_{\mathrm{GRPO}}$ and $l_0 < r_0$, which can be supplied manually or obtained from a coarse pre-scan. The algorithm first validates these endpoints and then initializes $[l, r] \leftarrow [l_0, r_0]$. At each iteration, it sets $m = \lfloor (l + r)/2 \rfloor$, materializes or retrieves l, m, r, caches the trajectory from l to m, and evaluates $S(l), S(m), S(r)$ if unavailable. It then updates the retained interval by

$$[l, r] \leftarrow \begin{cases} [l, m], & S(l) \geq S(r), \\ [m, r], & \text{otherwise}, \end{cases} \tag{6}$$

and discards cached checkpoints outside the retained interval. The search stops when $r - l \leq 1$ and returns the terminal midpoint, as shown in Algorithm 2. Because the branch rule compares endpoint scores and assumes no unimodality, the method is not binary search and provides no guarantee of recovering the global maximizer of S. Its purpose is to focus a limited evaluation budget on recoverable candidate states within the realized training trajectory.

3.4 Implementation

The implementation maps the preceding reproduction requirements to shared deterministic execution paths across the rollout and training backends. Anchor spacing bounds reproduction overhead without changing the recoverable candidate set. For a requested step t, the reproduction distance is $d(t) = t - a(t)$, and reproduction compute is linear in $d(t)$. If the save interval is K steps, one reproduction requires fewer than K updates. States along the same reproduced segment are cached to reduce overhead further and can be hit by later requests.

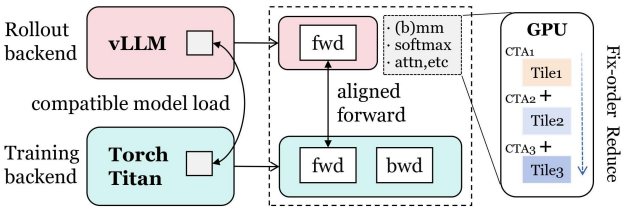

Figure 5: Top-down deterministic system implementation

Table 1: Overall results of Ministral-3-8B-Instruct-2512, Qwen3-4B-Instruct-2507, Qwen3.5-4B across mathematical datasets AIME24/25/26, AMC23, MATH500, Olympiad and Minerva.

Method	validation set AIME24	AIME25	AIME26	AMC23	MATH500	Olympiad	Minerva
			Ministral-3-8B-Instruct-2512				
Base Model	0.100	0.167	0.067	0.400	0.634	0.389	0.301
Best Interval	0.067	0.100	0.067	0.375	0.672	0.347	0.323
Final Step	0.033	0.033	0.067	0.225	0.472	0.163	0.224
PhCS	**0.167**	**0.167**	**0.100**	**0.425**	**0.690**	**0.391**	**0.408**
			Qwen3-4B-Instruct-2507				
Base Model	0.200	0.233	0.133	0.650	0.642	0.478	0.305
Best Interval	0.267	0.333	0.233	0.675	0.746	0.525	0.308
Final Step	0.200	0.300	0.233	0.675	0.750	0.527	0.301
PhCS	**0.400**	**0.367**	**0.333**	**0.700**	0.748	**0.531**	**0.327**
			Qwen3.5-4B				
Base Model	0.267	0.200	0.200	0.600	0.802	0.491	0.518
Best Interval	0.500	0.400	0.433	0.925	0.890	0.691	0.618
Final Step	0.400	0.400	0.567	0.850	0.886	0.696	0.526
PhCS	**0.733**	**0.500**	0.467	**0.975**	**0.926**	**0.721**	0.614

Figure 5 shows how our implementation connects vLLM rollouts (Kwon et al., 2023) with TorchTitan policy updates (Liang et al., 2025) through the experimental `deterministic_vllm_rl` stack.[2] The implementation supports both full fine-tuning and LoRA. It also integrates FSDP with activation checkpointing. The experiments in this paper use tensor parallelism 1 and FSDP over two GPUs. We provide fixed-order forward and backward kernels for matrix multiplication and mean reduction. We also patch softmax/log-softmax and RMSNorm, while attention uses a fixed split policy. The rollout and training workers share these patched operator paths. During deterministic replay, the data order and random state remain fixed. The parallel topology and optimizer state are also preserved. Appendices A and B describe the compatibility paths evaluated for Qwen3.5 Gated Delta Network (Qwen Team, 2026) and Ministral-3 attention (Liu et al., 2026a). Together, these choices define the fixed execution configuration validated in our experiments. The recovery guarantee is confined to the validated model and kernel paths. It also assumes the same software stack, device configuration, and parallel topology.

4 Experiment

4.1 Experiment Settings

Base Models. The base models include Qwen3-4B-Instruct-2507(Yang et al., 2025), Qwen3.5-4B(Qwen Team, 2026), and Ministral-3-8B-Instruct-2512(Liu et al., 2026a). Qwen3-4B-Instruct-2507 and Ministral-3-8B-Instruct-2512 are non-thinking models, while Qwen3.5-4B is a thinking model. Qwen3-4B-Instruct-2507 and Qwen3.5-4B possess basic mathematical and programming skills, while Ministral-3-8B-Instruct has weak mathematical and programming abilities and is just trained for instruction-following.

Training Scenario. The training, consisting of SFT and GRPO, is conducted based on the ReTool framework(Feng et al., 2025). In ReTool, when given a math problem, the LLM can dynamically call the tool "Code Interpreter", convert the calculation steps into code and obtain execution results, and then feed the execution results back into the subsequent reasoning. The model needs to autonomously decide whether to call the tool, when to call it, which code to call, and how to use the execution results to correct the subsequent reasoning, and finally output the answer.

Datasets The training in the SFT stage uses the ReTool-SFT(Feng et al., 2025). The GRPO training stage uses the dataset DAPO-Math-17k(Yu et al., 2025). The evaluation dataset in-

[2]https://github.com/bwasti/torchtitan/tree/main/torchtitan/experiments/deterministic_vllm_rl

cludes AIME24, AIME25, AIME26, AMC23, Math500, Minerva(Lewkowycz et al., 2022) and Olympiad(He et al., 2024). AIME24 works as a validation set for scoring in Dynamic Selection.

4.2 Overall Results

We evaluated PhCS using the pass@4 metric. The baseline included the **Base Model**, which is the untrained model, **Best Interval**, which refers to the best of checkpoints saved at fixed intervals and can be seen as exhaustive search in conventional training paradigm, and **Final Step**, which is the checkpoint reached the total number of training steps. **Best Interval** and **Final Step** represent the checkpoints that the conventional training paradigm could obtain. In the experiment, the interval was set to 25 and the total steps were set to 250. For Qwen3-4B-Instruct-2507, the SFT step selected by PhCS is 119, and GRPO step is 174. For Qwen3.5-4B, the SFT step selected by PhCS is 91, and GRPO step is 149. For Ministral-3-8B-Instruct-2512, the SFT step selected by PhCS is 144, and GRPO step is 133. **Almost all checkpoints fall out of the Best Interval or the Final Step**.

Table 1 summarizes the results. PhCS achieves the best or comparable performance to both Best Interval and Final Step across all three models and the majority of benchmarks. Notably, Final Step often underperforms Best Interval, indicating that training to the final step can lead to overfitting or degradation, whereas PhCS's adaptive checkpoint selection avoids this issue. For Qwen3.5-4B, PhCS achieves 0.733 on AIME24, a 46.6% relative improvement over 0.5 of Best Interval and 83.3% over 0.4 of Final Step. On AMC23, PhCS reaches 0.975, compared to 0.925 for Best Interval and 0.85 for Final Step. Similar trends hold for Qwen3-4B-Instruct-2507 and Ministral-3-8B-Instruct-2512, where PhCS yields the best or tied-best scores on nearly every benchmark. The few exceptions are marginal and do not diminish the overall superiority of PhCS.

4.3 Tradeoff Analysis

We conducted statistical experiments on equivalent storage space and the dilation of training time caused by the introduction of deterministic kernels. This part involves two parameters: checkpoint interval and the total number of training steps. The PhCS training additionally saves the optimizer state and training metadata. However, the advantage is that all checkpoints can be retrieved. The conventional training paradigm would need to save checkpoints at every step to achieve the same effect. The "equivalent storage" metric is the ratio of the storage space required to obtain any checkpoint by the PhCS to the storage space needed by the conventional training paradigm. The "time dilation" is the ratio of the total time of training in PhCS plus the average reproduction time to the total training time of the conventional paradigm. The I/O time is hidden through asynchronous operations and is not taken into account. As shown in Figure 6, the experiments show that PhCS achieves a storage efficiency of over 51.4× with a time dilation of less than 1.98×.

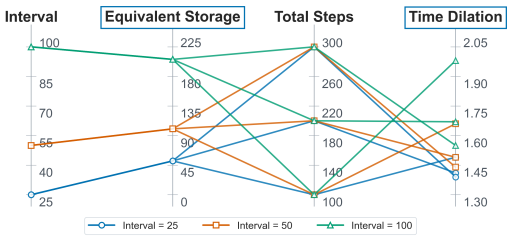

Figure 6: Equivalent storage and time dilation under different save intervals and total steps

4.4 Ablation Analysis

Figure 7 shows the ablation experiment that replaces the PhCS in the two-stage SFT-GRPO training of Qwen3-4B-Instruct-2507 with early stop, fixed-interval saving, and final-step checkpoints from the conventional training. The condition for Earlystop is that, in SFT, within $patience = 5$ steps the loss is not lower than that of the current step; in GRPO, within $patience = 10$ steps the reward is not higher than that of the current step. In Figure 8, both stages achieved the optimal result using PhCS on multiple datasets. Any one of the stages being ablated would result in a performance decline.

4.5 Determinism Fidelity

The determinism checklists in Appendix H cover Ministral-3 and Qwen3.5 GDN. The Ministral-3 checklist verifies bitwise reproducibility across CPU/CUDA forward, backward, optimizer updates,

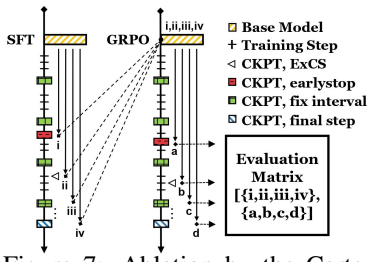

Figure 7: Ablation by the Cartesian product of the 4 checkpoint strategies in SFT and GRPO

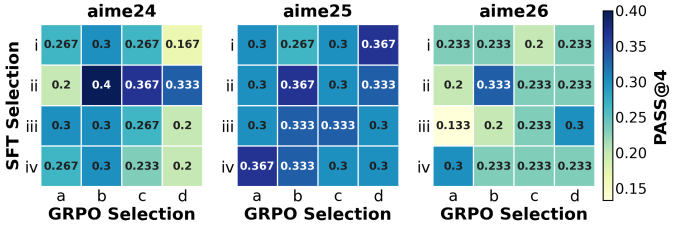

Figure 8: The corresponding results of the 16 ablation experiments. The checkpoint used in both SFT and GRPO stages is located in the performance hotspots area.

and cross-process vLLM rollout. The Qwen3.5 GDN checklist validates high-precision mathematical correctness, differentiability, batch/request isolation, prefill/decode equivalence, and repeatability. Both require strict element-wise equality, showing that determinism holds across execution paths, training and inference backends, and process boundaries rather than only within tolerance.

4.6 Reproduction Case Study

Figure 9 illustrates the two selection procedures on the Qwen3.5-4B training trajectory. In the SFT, PhCS first forms a candidate set by intersecting the checkpoints in the lowest 50% of training loss with those in the highest 50% of token accuracy. This yields 83 candidates. We then select the candidate with the highest mean entropy, which is 91. Its raw loss, token accuracy, and mean entropy are 0.3094, 0.8931, and 0.3264, respectively. In the GRPO, PhCS specifies the initial search interval as steps [125, 175] and performs dynamic selection using AIME24 pass@4. At each round, we evaluate the interval endpoints and midpoint, then retain the half indicated by the comparison of the two endpoints. After five rounds, the interval contracts to [149, 150], and the procedure selects step 149. This checkpoint obtains a pass@4 of 0.60, corresponding to 18 of 30 prompts with at least one correct response among four generations.

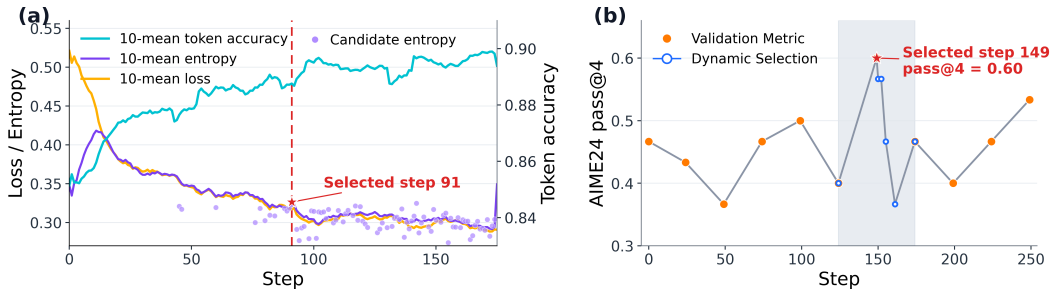

Figure 9: (a) Static selection in SFT. Lines show 10-step means of loss, token accuracy, and mean entropy; dots show the raw entropy values of the candidates. (b) Dynamic selection in GRPO by validation set AIME24 pass@4 across measured anchor and subsequent probe.

5 Conclusion

The optimal training checkpoint can exist at neither the save interval nor the final step, which are set before the training. However, in the conventional training paradigm, these intermediate checkpoints are not accessible because reproducibility is not granted. We propose the Post-hoc Checkpoint Selection(PhCS) framework to address this issue. PhCS implements a deterministic training system from the bottom up to ensure reproducibility of training. Additionally, PhCS incorporates various policies for selecting the candidate checkpoints for reproduction, covering all granularities of training metrics. Thereby, PhCS achieves the "*train with hindsight*" post-hoc training paradigm, which significantly reshaped the landscape of LLM training. The experiments conducted on an SFT-Agentic RL training scenario demonstrate that PhCS achieves state-of-the-art on multiple datasets and achieves superior equivalent storage efficiency.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation, manuscript writing and polishing, experiment feedback, visualization, coding, debugging, and reference proofreading. Generative AI was not used to formulate mathematical claims, provide critical ingredients for mathematical proofs, or propose or refine research hypotheses. Synthetic data generation and the development of theoretical models or conceptual frameworks were not applicable to this work. All AI-assisted outputs were reviewed and verified by the authors. In particular, the mathematical derivations and algorithm workflows were developed by the authors rather than proposed by AI, and the key related work was identified by the authors. AI-generated code was executed and verified by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the anonymous source code for the Deterministic Training System, Checkpoint Reproduction, and Selection Policies implementation at `https://anonymous.4open.science/r/test-AE86/`, including the necessary software library dependencies. Meanwhile, in Appendix G, we list the hyperparameters for training and evaluation, and in Appendix H, we have presented the checklist for verifying the system's determinism. It is necessary to clarify that the current determinism only guarantees the run-to-run level within the same hardware environment, and does not guarantee cross-hardware determinism across GPUs. Our hardware environment consists of 2×NVIDIA H100 with 80GB HBM3 memory, and the CUDA version is 12.8.

ETHICS STATEMENT

PhCS aims to drive the innovation of the AI training system through the new features of the infrastructure system, without involving studies on human subjects. potentially harmful findings, research methods and applications, potential conflicts of interest and sponsorship relationships, issues of discrimination/bias/equity, privacy and security and legal compliance.

REFERENCES

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback, 2022. URL `https://arxiv.org/abs/2204.05862`.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023. URL `https://arxiv.org/abs/2211.12588`.

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. In *arXiv preprint arXiv:2110.14168*, 2021. URL `https://arxiv.org/abs/2110.14168`.

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630, 2024.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL `https://arxiv.org/abs/2504.11536`.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pp. 1050–1059, 2016.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023. URL https://arxiv.org/abs/2211.10435.

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, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems, 2024. URL https://arxiv.org/abs/2402.14008.

Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference, 2025. URL https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient triton kernels for llm training, 2025. URL https://arxiv.org/abs/2410.10989.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.09516.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models, 2022. URL https://arxiv.org/abs/2206.14858.

Xinran Li, Guangda Huzhang, Siqi Shen, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Jun Zhang. Getting your llms ready for reinforcement learning with lightweight sft. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=yezWGJmODg.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. Torl: Scaling tool-integrated rl, 2025. URL https://arxiv.org/abs/2503.23383.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready llm pre-training, 2025. URL https://arxiv.org/abs/2410.06511.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, Alexandre Sablayrolles, Amélie Héliou, Amos You, Andy Ehrenberg, Andy Lo, Anton Eliseev, Antonia Calvi, Avinash Sooriyarachchi, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Clémence Lanfranchi, Corentin Barreau, Cyprien Courtot, Daniele Grattarola, Darius Dabert, Diego de las Casas, Elliot Chane-Sane, Faruk Ahmed, Gabrielle Berrada, Gaëtan Ecrepont, Gauthier Guinet, Georgii Novikov, Guillaume Kunsch, Guillaume Lample, Guillaume Martin, Gunshi Gupta, Jan Ludziejewski, Jason Rute, Joachim Studnia, Jonas Amar, Joséphine Delas, Josselin Somerville Roberts, Karmesh Yadav, Khyathi Chandu, Kush Jain, Laurence Aitchison, Laurent Fainsin, Léonard Blier, Lingxiao Zhao, Louis Martin, Lucile Saulnier, Luyu Gao, Maarten Buyl, Margaret Jennings, Marie Pellat, Mark Prins, Mathieu Poirée, Mathilde Guillaumin, Matthieu Dinot, Matthieu Futeral, Maxime Darrin, Maximilian Augustin, Mia Chiquier, Michel Schimpf, Nathan Grinsztajn, Neha Gupta, Nikhil Raghuraman, Olivier Bousquet, Olivier Duchenne, Patricia Wang, Patrick von Platen, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Quentin Torroba, Romain Sauvestre, Roman Soletskyi, Rupert Menneer, Sagar Vaze, Samuel Barry, Sanchit Gandhi, Siddhant Waghjale, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Teven Le Scao, Théo Cachet, Theo Simon Sorg, Thibaut Lavril, Thiziri Nait Saada, Thomas Chabal, Thomas Foubert, Thomas Robert, Thomas Wang, Tim Lawson, Tom Bewley, Tom Bewley, Tom Edwards, Umar Jamil, Umberto Tomasini, Valeriia Nemychnikova, Van Phung, Vincent Maladière, Virgile Richard, Wassim Bouaziz, Wen-Ding Li, William Marshall, Xinghui Li, Xinyu Yang, Yassine El Ouahidi, Yihan Wang, Yunhao Tang, and Zaccharie Ramzi. Ministral 3, 2026a. URL https://arxiv.org/abs/2601.08584.

Runze Liu, Jiashun Liu, Xu Wan, Yuqian Fu, and Ling Pan. When rl fails after sft: Rejuvenating model plasticity for robust sft-to-rl handoff, 2026b. URL https://arxiv.org/abs/2606.09932.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. Reft: Reasoning with reinforced fine-tuning, 2024. URL https://arxiv.org/abs/2401.08967.

Manh Nguyen, Dung Nguyen, Dai Do, Svetha Venkatesh, and Hung Le. Uncertainty-guided checkpoint selection for reinforcement finetuning of large language models, 2025. URL https://arxiv.org/abs/2511.09864.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744, 2022.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, 2023. URL https://aclanthology.org/2023.findings-emnlp.248/.

Lutz Prechelt. Early stopping–but when? In *Neural Networks: Tricks of the Trade*, pp. 55–69. Springer, 2002.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Jie Ren, Peter J. Liu, Emily Fertig, Jasper Snoek, Ryan Poplin, Mark DePristo, Joshua Dillon, and Balaji Lakshminarayanan. Likelihood ratios for out-of-distribution detection. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019. URL https://arxiv.org/abs/1909.08053.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F. Christiano. Learning to summarize with human feedback. In *Advances in Neural Information Processing Systems*, volume 33, pp. 3008–3021, 2020.

Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, Yizhong Wang, Hannaneh Hajishirzi, Noah A. Smith, and Yejin Choi. Dataset cartography: Mapping and diagnosing datasets with training dynamics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 9275–9293, 2020.

The SGLang Team. Towards deterministic inference in sglang and reproducible rl training., 2025. URL https://lmsys.org/blog/2025-09-22-sglang-deterministic/.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning, 2023. URL https://arxiv.org/abs/2310.03731.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Nathan Whitehead and Alex Fit-Florea. Precision & performance: floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21, 01 2011.

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, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and mitigating numerical sources of nondeterminism in llm inference, 2025. URL https://arxiv.org/abs/2506.09501.

Ziyang Zhang, Xinheng Ding, Jiayi Yuan, Rixin Liu, Huizi Mao, Jiarong Xing, and Zirui Liu. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch, 2026. URL https://arxiv.org/abs/2511.17826.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL https://arxiv.org/abs/2304.11277.

A Deterministic Gated Delta Network

The central design is to make the forward and backward computation of the Qwen3.5 Gated Delta Network(GDN) deterministic by treating it as a fixed state-transition system with fixed reduction order, rather than relying only on framework-level deterministic switches. The main sources of nondeterminism are recurrent state updates, causal convolution, vector reductions, batch packing, and fused kernels that may reorder floating-point operations. The implementation therefore first defines an explicit mathematical execution order, and then aligns the training and GPU-accelerated paths to that order.

The forward pass is decomposed into deterministic substeps: token-ordered depthwise causal convolution, Q/K/V and gate projection splitting, Gated DeltaNet recurrent update, and gated RMSNorm. Each step avoids leaving reduction order to the framework. Request states are isolated from one another; tokens are processed in sequence within each request; convolution taps are accumulated in a fixed order; Q/K normalization sums dimensions in a fixed order; and the delta-rule memory computation, state update, and output reduction all traverse the key dimension in a fixed order. As a result, full-sequence prefill and token-by-token decode share the same state-transition semantics.

The GPU forward path is not a different approximation of the algorithm. It moves the same fixed-order semantics into fixed-shape kernels. Kernel dimensions, convolution width, warp and stage counts, fusion policy, and key float32 round-to-nearest operations are explicitly constrained so that acceleration does not change the state-transition rule. The packed path optimizes cache access, but still follows the same principle: request offsets, state slots, and initial-state flags are explicit metadata, so batch layout does not implicitly define the computation.

The backward pass avoids depending on opaque upstream fused backward kernels. During training, the forward path uses the deterministic GDN state-update operator and saves the required inputs. During backward, it rebuilds a differentiable version of the same forward computation with a fixed chunk size, then asks PyTorch to compute gradients under deterministic Torch/CUBLAS settings. Chunk size, padding, per-batch sequence layout, convolution recomputation, Q/K normalization, and delta-rule recurrence order are all fixed, so the gradient path is repeatable for the same inputs and runtime environment.

B Deterministic Ministral Attention

The deterministic support for Ministral 3 is best understood as a constrained forward/backward computation pipeline. Determinism here is not only about fixing random seeds; it is about fixing tensor layouts, attention visibility, head mapping, RoPE value sources, reduction order, and the backward path.

In the forward pass, the adapter first makes every operator input semantically unambiguous. Q, K, and V projections are shaped according to Ministral's GQA rules, so the mapping from query heads to KV heads does not depend on batch shape or incidental broadcasting. RoPE values are determined before attention, so the same position and configuration produce the same rotation factors. The sliding window is represented as an explicit left-causal visibility range rather than being inferred implicitly by the kernel. As a result, attention receives inputs with fixed layout, fixed mask, fixed window, and fixed scaling.

Attention is the most important operator boundary in the Ministral adapter. The adapter reorganizes contiguous sequences into the block layout expected by the shared runtime while preserving Ministral's GQA and sliding/full attention semantics. Determinism comes from two sides: the adapter makes block partitioning, sequence-length alignment, and visibility windows deterministic; the shared attention kernel makes softmax, weighted value accumulation, and log-sum-exp statistics reproducible for that fixed semantic problem.

The backward pass follows the same semantic boundary. The forward pass saves the Q, K, V tensors, output, and normalization statistics required by the backward computation. The backward pass then uses the same causal/sliding window, the same softmax scale, and the same head grouping to compute gradients. This avoids reconstructing a slightly different attention problem during backpropagation and keeps window boundaries, GQA expansion, and softmax normalization consistent across forward and backward.

Other operators follow the same rule: the Ministral-specific code defines model semantics and weight layout, while numerical execution is delegated to the deterministic runtime. RMSNorm keeps residual and normalization ordering stable; SwiGLU merges the gate/up projections into a stable layout and then splits and activates them in a fixed order; output projection and tied embeddings are handled as deterministic weight references or layout conversions. Static FP8 checkpoint weights are dequantized into dense weights before execution, so quantization metadata does not leak into later operator selection.

C Training Recipe

This section describes the SFT–GRPO post-training recipe used in our experiments. The pipeline is decomposed into independent stages: run SFT, select and export an SFT checkpoint with PhCS, run GRPO from the exported checkpoint, and finally apply PhCS to select the final checkpoint from the GRPO trajectory.

C.1 SFT Recipe

Our SFT stage performs full-model supervised fine-tuning rather than adapter tuning. We use a Qwen3-family base model and a 2,000-example ReTool-SFT training set. Each row follows the ReTool-SFT message format: the input column is `messages` and the label column is `label`. We apply the model tokenizer's chat template before training, shuffle the dataset with seed 0, and filter long completions by keeping examples below the 90th completion-token percentile. This filter keeps 1,800 examples and caps retained completions at 3,255 tokens.

The SFT objective is standard completion-only next-token prediction. The trainer concatenates prompt and completion tokens, masks prompt positions with the ignore index, and computes cross-entropy only over completion tokens. The implementation optionally uses Liger's fused cross-entropy loss(Hsu et al., 2025) for memory efficiency, while keeping compatibility with vLLM-based rollout. For multi-GPU runs, the trainer uses PyTorch FSDP with full-shard strategy.

During SFT, PhCS records stored anchor checkpoints and lightweight per-step records. The recipe writes anchors every 50 optimizer steps and records scalar training signals at every step, including loss, perplexity, gradient norm, token accuracy, mean target-token probability and log-probability, mean entropy, top-1 probability, top-1 logit margin, and completion length statistics.

C.2 GRPO Recipe

The GRPO stage follows a ReTool-style training recipe adapted from the slime example.[3] It starts from the exported SFT checkpoint selected by PhCS. Training prompts are drawn from DAPO-Math-17k with `prompt` as the input field and `label` as the verifier target. The driver applies the chat template and shuffles examples with seed 0.

GRPO is implemented as a rollout-update loop. At each step, a vLLM-based rollout engine samples a prompt batch and generates a group of completions per prompt. The rollout bundle stores completions, token ids, prompt token ids, old token log-probabilities, loss masks, tool-call counts, rewards, and sample metadata. A distributed FSDP train worker then consumes the rollout bundle, recomputes current-policy token log-probabilities, and applies a PPO-style clipped policy gradient loss. Rewards are normalized within the rollout batch and transformed into group-relative advantages; by default the implementation uses the exponential GRPO advantage transform with `grpo_beta=0.1`, while a stable mean-centered variant is available as an option.

The ReTool-compatible GRPO reward checks final mathematical answers. Correct answers receive reward $+1$, incorrect answers receive reward -1, and incorrect ReTool trajectories receive a small tool-use shaping bonus based on the number of tool calls, capped so that incorrect trajectories remain negative. The PPO clip range is asymmetric, with lower epsilon 0.2 and upper epsilon 0.28. The default KL and entropy coefficients in the ReTool-style recipe are both 0.

The GRPO recipe enables a constant-reward group filter before each policy update. For a prompt with G sampled responses, the driver first recomputes the reward vector $r_{1:G}$ using the same reward

[3] https://thudm.github.io/slime/_examples_synced/retool/README.html

function used for training. It then computes the group reward span $\Delta = \max_i r_i - \min_i r_i$. A prompt group is kept only when $\Delta > \epsilon$, with $\epsilon = 0.1$ in our configuration. This removes all-correct and all-incorrect groups, whose within-group relative advantages would otherwise carry little useful preference signal. Filtering is performed at the group level: when a prompt is kept, all of its sampled responses, token ids, log-probabilities, loss masks, tool-call counts, rewards, and metadata are kept together; when it is dropped, the whole group is removed from the rollout bundle used by the train worker.

The filter is implemented as a pre-update rollout selection step rather than an online stopping criterion. If the current prompt batch contains no group with $\Delta > \epsilon$, the driver records the failed attempt, advances the dataset cursor, and samples the next prompt batch. It may retry up to one full pass over the prompt set, i.e., $\lceil N/B \rceil$ attempts for N prompts and rollout batch size B. If no non-constant group is found after these attempts, the implementation falls back to the last raw rollout so that training can continue. The filter summary, including the attempted dataset indices, reward ranges, kept and dropped group indices, kept fraction, cursor range, and fallback flag, is retained with the training record for later analysis and reproduction.

PhCS uses sparse checkpoint storage during GRPO. Anchor steps keep stored policy checkpoints and optimizer state, while intermediate candidate steps keep lightweight records. In our configuration, anchors are written every 25 steps and candidate statistics are recorded every step.

D ReTool Framework

We follow ReTool (Feng et al., 2025) as the tool-integrated rollout pipeline for the GRPO stage. ReTool augments long-form mathematical reasoning with real-time code execution: the policy model alternates between natural-language reasoning, tool invocation, sandbox feedback, and final-answer generation. In our implementation, this pipeline is instantiated as follows:

1. **Tool-aware prompt construction.** Each math problem is rendered by `format_conversation_with_tools` as a chat prefix with a system message, a `# Tools` block, a JSON signature for `code_interpreter` inside `<tools>...</tools>`, the user problem, and an assistant generation marker. The tool schema has one required string argument, `code`, and the model is instructed to emit calls as JSON inside `<tool_call>...</tool_call>`. The exact template is shown in the next appendix section.
2. **Interleaved rollout.** For each prompt, the rollout engine creates G sampled trajectories. At each ReTool turn, unfinished trajectories are batched into vLLM generation with temperature sampling, optional `top_p`, seed 42, and token log-probabilities enabled. Generated text is then parsed as either a tool call, a final answer, or an invalid action.
3. **Sandbox execution and feedback.** A valid `<tool_call>` to `code_interpreter` is executed in the sandbox. The result, including either output or error messages, is appended as `<interpreter>...</interpreter>` and fed back into the next generation turn. A final `Answer: \boxed{...}` terminates the trajectory; invalid actions receive a short format reminder and continue if budget remains.
4. **Training interface.** Tool feedback remains in the context but is masked out of the policy loss, following the interpreter-feedback masking idea in ReTool. Concretely, observation tokens receive log-probability 0 and `loss_mask=0`, while model-generated reasoning, tool-call, and answer tokens receive `loss_mask=1`. The rollout bundle stores completions, token ids, log-probabilities, loss masks, tool-call counts, rewards, and metadata for reward computation, GRPO update, and deterministic reproduction.

E Sandbox Setting

The sandbox exposes one tool, `code_interpreter`, whose only argument is a string field named `code`. Each tool call is executed in a fresh temporary directory with `PYTHONUNBUFFERED=1` and a sandbox-specific `PYTHONPYCACHEPREFIX`. The generated code is written to a temporary `code.py` file and launched in a child Python process with the temporary directory as its working

directory. Standard output and standard error are redirected to in-process buffers by a generated wrapper script and returned as the tool observation. If the child process exits with a nonzero code, the observation contains the exit code and standard error; if execution exceeds the timeout, the process is killed and an explicit timeout error is returned. A tool call times out after 60 seconds in the implemented default configuration.

The sandbox applies both syntactic safety checks and process-level isolation. Before execution, code is normalized by removing optional Markdown fences and is rejected if it contains dangerous patterns such as dynamic imports, `eval`, `exec`, file I/O, direct introspection helpers, or dunder access. It also parses the code with Python's AST module and rejects imports of blocked system modules such as `os`, `sys`, `subprocess`, `shutil`, `glob`, and `pathlib`. Imports outside the explicit allowlist are also rejected. The allowlist is limited to common mathematical and standard-library utilities: `math`, `fractions`, `decimal`, `statistics`, `itertools`, `collections`, `functools`, `operator`, `random`, `datetime`, and `sympy`. Heavy packages such as `numpy`, `scipy`, `pandas`, `torch`, and `tensorflow` are intentionally unavailable.

The process wrapper additionally attempts to set an address-space limit. The configured memory limit is 4GB. The parent process tracks its own resident memory with `psutil` when available: above 12GB it refuses new tool execution and performs aggressive garbage collection; lower thresholds trigger light or regular cleanup messages that are appended to tool result.

Concurrent tool execution is handled at two levels. At the rollout level, after each generation turn the engine collects all active trajectories and calls `asyncio.gather` over their parsed actions, so tool calls from different samples in the same batch can be processed concurrently instead of serially blocking the next rollout turn. At the sandbox level, each valid `code_interpreter` request is dispatched through `asyncio.to_thread` to run the synchronous sandbox executor without blocking the event loop. Before launching the child Python process, the executor acquires a global `threading.BoundedSemaphore`; the default capacity is 32. This bounds the number of simultaneous sandbox subprocesses across a rollout worker, preventing large prompt groups from creating unbounded process, CPU, and memory pressure. Timed-out subprocesses are killed independently, and completed calls release the semaphore automatically when their execution context exits. The implemented default interaction limits are 8 ReTool turns and 8 tool calls per trajectory.

F System Prompt

The ReTool rollout wrapper uses the following default system prompt:

```
<|im_start|>system
You are a helpful assistant that can use Python tools to solve mathematical problems.
     Use the code_interpreter proactively to assist mathematical reasoning, especially
     for arithmetic, enumeration, symbolic manipulation, and checking intermediate
     results. The code interpreter supports Python built-ins and these modules: math,
     fractions, decimal, statistics, itertools, collections, functools, operator,
     random, datetime, sympy; other packages, e.g. numpy and scipy, are unavailable.
# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "code_interpreter", "description": "A tool
     for executing Python code in a safe sandbox environment.", "parameters": {"type":
     "object", "properties": {"code": {"type": "string", "description": "The Python
     code to execute"}}, "required": ["code"]}}}
</tools>

For each function call, return a json object with function name and arguments within
     <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
<|im_end|>
<|im_start|>user
{problem prompt}
<|im_end|>
<|im_start|>assistant
```

Category	Hyperparameter	Value
Model/data	SFT data	`ReTool-SFT/train_2000.parquet`
Model/data	input / label keys	`messages` / `label`
Data processing	chat template / shuffle	enabled / enabled
Data processing	dataset seed	0
Training	epochs	4
Training	logical batch size	32
Training	micro batch size	2
Training	parallelism	FSDP on GPUs 0,1
Optimization	optimizer	Adam(Kingma & Ba, 2017)
Optimization	learning rate	1×10^{-5}
Optimization	schedule	cosine, min LR 1×10^{-6}
Optimization	warmup fraction	0.1
Optimization	weight decay	0.1
Optimization	Adam betas	(0.9, 0.95)
Checkpointing	anchor / candidate-record interval	50 / 1 steps

Table 2: SFT hyperparameters.

Category	Hyperparameter	Value
Data	RL data	`dapo-math-17k.jsonl`
Data	input / label keys	`prompt` / `label`
Rollout	rollout prompts	10,000
Rollout	prompt batch size	16
Rollout	samples per prompt	8
Rollout	max response length	4,096
Rollout	temperature / top-p	1.0 / 1.0
Rollout	max model length	16,384
ReTool	max turns / tool calls	8 / 8
Training	optimizer steps	1,000
Training	micro batch size	2
Training	rollout / train GPUs	0,1 / 0,1
Optimization	optimizer	Adam
Optimization	learning rate	2×10^{-6}
Optimization	schedule	constant
Optimization	weight decay	0.1
Optimization	Adam betas	(0.9, 0.98)
GRPO loss	beta	0.1
GRPO loss	KL / entropy coefficients	0 / 0
GRPO loss	PPO clip eps low / high	0.2 / 0.28
GRPO loss	max grad norm	1.0
Filtering	constant-reward group filter	enabled, eps 0.1
Checkpointing	anchor / record interval	25 / 1 steps

Table 3: GRPO training hyperparameters.

When tools are enabled, the wrapper appends a tool instruction block specifying that function calls must be returned as JSON objects inside `<tool_call>...</tool_call>` XML tags. This makes the tool interface deterministic to parse and allows the recorded trajectory to preserve the full model–tool interaction.

G HYPERPARAMETERS

Table 2 summarizes the SFT hyperparameters used in our experiments.

Table 3 summarizes the GRPO training configuration.

Table 4 summarizes the default evaluation configuration. Training-time GRPO evaluation uses AIME 2024 every 50 steps in the local run script, while the evaluation script evaluates a broader benchmark suite.

Category	Hyperparameter	Value
Training-time eval	dataset	AIME 2024
Training-time eval	interval	50 steps
Training-time eval	samples per prompt	4
Training-time eval	max response length	4,096
Training-time eval	top-p	1.0
Final eval	datasets	AIME24/25/26, AMC23, MATH500, OlympiadBench, MinervaMath
Final eval	samples per prompt	4
Final eval	max response length	8,192
Final eval	temperature / top-p	1.0 / 1.0
Final eval	rollout batch size	128
Final eval	max model length	16,384
Final eval	ReTool turns / calls	8 / 8
Final eval	vLLM GPU memory utilization	0.95

Table 4: Evaluation hyperparameters.

H DETERMINISM CHECKLIST

As shown in Table 5, the two checklists jointly confirm determinism for both Ministral-3 and Qwen3.5 GDN, covering forward and backward computation, optimizer updates, cross-backend rollout, mathematical correctness, batch/request isolation, prefill/decode equivalence.

Table 5: Determinism Checklists for Ministral-3 and Qwen3.5 Architecture

Architecture	Category	Check Item	Status
TheMinistral-3 determinism checklist uses a small Ministral configuration and performs element-wise bitwise comparisons for forward, backward, parameter updates, and vLLM rollout. The tests do not compare whether the error is below a threshold; instead, they directly require the corresponding tensors to satisfy strict element-wise equality conditions.			
The Qwen3.5 GDN deterministic ops checklist fixes the random seed; cases involving high-precision mathematical comparison and gradient checking use `float64` accumulation.			
Ministral3	Test Preconditions	Initialize the model and inputs with a fixed random seed.	✓
Ministral3	Test Preconditions	Use a fixed 2×16 token input to ensure that the CUDA attention sequence length satisfies the 16-token block alignment requirement.	✓
Ministral3	Test Preconditions	The small model simultaneously covers GQA, YaRN RoPE, sliding attention, and full attention.	✓
Ministral3	Test Preconditions	Enable deterministic runtime configurations such as Triton attention, batch-invariant, and training/inference alignment.	✓
Ministral3	Test Preconditions	CUDA tests perform deterministic patch initialization after importing the model and patches.	✓
Ministral3	CPU Forward	Run two consecutive forward passes with the same model parameters, same inputs, and same attention mask.	✓
Ministral3	CPU Forward	Check that every element of the output logits from the two runs is exactly identical.	✓
Ministral3	CPU Forward	This test verifies that GQA head repetition, sliding-window mask, RoPE, and the model forward path in the CPU fallback remain consistent across repeated executions.	✓
Ministral3	CUDA Forward	Construct a Ministral model in evaluation mode on CUDA with BF16.	✓
Ministral3	CUDA Forward	Run the same forward pass twice consecutively and synchronize CUDA after both executions.	✓
Ministral3	CUDA Forward	Check that the two logits are exactly element-wise identical.	✓

Architecture	Category	Check Item	Status
Ministral3	CUDA Forward	This test covers deterministic forward computation for CUDA unified attention, GQA head mapping, sliding-window, and full attention layers.	✓
Ministral3	CUDA Backward and Optimizer Update	Fix an initial model state and start two independent training steps from the same state.	✓
Ministral3	CUDA Backward and Optimizer Update	Both training runs perform a forward pass, mean-squared loss, backpropagation, and AdamW parameter update.	✓
Ministral3	CUDA Backward and Optimizer Update	Compare the two losses and require them to be exactly identical.	✓
Ministral3	CUDA Backward and Optimizer Update	Compare the parameter name sets of the two gradient tensors and require them to be exactly the same.	✓
Ministral3	CUDA Backward and Optimizer Update	Compare the gradient of each parameter and require exact element-wise identity.	✓
Ministral3	CUDA Backward and Optimizer Update	Compare the parameter name sets after the two updates and require them to be exactly the same.	✓
Ministral3	CUDA Backward and Optimizer Update	Compare each updated parameter and require exact element-wise identity.	✓
Ministral3	CUDA Backward and Optimizer Update	This test verifies that determinism exists not only in forward outputs but also throughout attention backward, MLP/RMSNorm gradient propagation, and optimizer update results.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Export an HF/vLLM-loadable weight layout from the deterministic model state.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Launch vLLM separately in two fresh Python subprocesses.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Use the same 4092-token prompt and generate 4 tokens, covering the boundary of `max_model_len = 4096`.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Use greedy sampling: `temperature = 0.0`, with a fixed sampling seed.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Disable the FlashInfer sampler and vLLM multiprocessing mode, and fix the attention backend and batch-invariant-related environment.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Compare the token ID sequences generated by the two runs and require them to be exactly identical.	✓
Ministral3	vLLM Rollout Across Fresh Processes	Compare the logprob corresponding to each generated token and require them to be exactly identical.	✓
Ministral3	vLLM Rollout Across Fresh Processes	This test verifies that determinism can span Python process restarts, model loading, long-context boundaries, and vLLM rollout, rather than only repeated forward passes within the same process.	✓
Qwen3.5 GDN	Gated Delta Rule Matches Independent High-Precision Mathematical Implementation	Randomly generate Q, K, V, decay, beta, and recurrent state.	✓
Qwen3.5 GDN	Gated Delta Rule Matches Independent High-Precision Mathematical Implementation	Use the reference operator to compute token outputs and the final state.	✓
Qwen3.5 GDN	Gated Delta Rule Matches Independent High-Precision Mathematical Implementation	Use an independent per-token mathematical implementation in the test as the baseline.	✓
Qwen3.5 GDN	Gated Delta Rule Matches Independent High-Precision Mathematical Implementation	Compare both outputs and final state using `atol=1e-12` and `rtol=1e-12`.	✓
Qwen3.5 GDN	Gated Delta Rule Matches Independent High-Precision Mathematical Implementation	Cover the overall semantics of Q/K normalization, decay, memory, delta update, and output reduction.	✓
Qwen3.5 GDN	Gated Delta Rule Supports Correct Backpropagation	Construct `float64`, `requires_grad=True` inputs for Q, K, V, decay, beta, and initial state.	✓

Architecture	Category	Check Item	Status
Qwen3.5 GDN	Gated Delta Rule Supports Correct Backpropagation	Use PyTorch numerical gradient checking.	✓
Qwen3.5 GDN	Gated Delta Rule Supports Correct Backpropagation	Numerical differentiation parameters are `eps=1e-6`, `atol=1e-5`, and `rtol=1e-4`.	✓
Qwen3.5 GDN	Gated Delta Rule Supports Correct Backpropagation	Verify that the explicit state recurrence implementation is not only correct in forward but also forms a continuous, differentiable computation graph.	✓
Qwen3.5 GDN	Batch Size Changes Do Not Affect the Target Request's Result	Place the same target sequence into packed request sets with batch sizes 1, 2, 4, 8, and 16 respectively.	✓
Qwen3.5 GDN	Batch Size Changes Do Not Affect the Target Request's Result	Use different lengths and random inputs for other requests, changing the target request's batch position and surrounding context.	✓
Qwen3.5 GDN	Batch Size Changes Do Not Affect the Target Request's Result	Isolate each request through cumulative sequence-length offsets and independent initial states.	✓
Qwen3.5 GDN	Batch Size Changes Do Not Affect the Target Request's Result	Use strict element-wise equality on the target request's output slice and final recurrent state.	✓
Qwen3.5 GDN	Batch Size Changes Do Not Affect the Target Request's Result	Verify that batch expansion, request permutation, and other sequences do not change the target request's state transition result.	✓
Qwen3.5 GDN	Prefill and Per-Token Decode Have Exactly Consistent State Semantics	Perform a full-sequence prefill on the same five-token input.	✓
Qwen3.5 GDN	Prefill and Per-Token Decode Have Exactly Consistent State Semantics	Then, starting from the same initial convolution state and recurrent state, perform five per-token decodes.	✓
Qwen3.5 GDN	Prefill and Per-Token Decode Have Exactly Consistent State Semantics	Compare the concatenated outputs, final convolution state, and final recurrent state from the two paths.	✓
Qwen3.5 GDN	Prefill and Per-Token Decode Have Exactly Consistent State Semantics	All three use strict element-wise equality checks.	✓
Qwen3.5 GDN	Prefill and Per-Token Decode Have Exactly Consistent State Semantics	Cover causal convolution, GDN recurrence, gated RMSNorm, and cross-step state passing.	✓
Qwen3.5 GDN	Causal Convolution Two Cold Calls Are Raw Equal	Call the reference convolution twice consecutively with the same input, convolution weights, bias, initial state, and request offsets.	✓
Qwen3.5 GDN	Causal Convolution Two Cold Calls Are Raw Equal	Use strict element-wise equality on the convolution outputs and final convolution state from the two calls.	✓
Qwen3.5 GDN	Causal Convolution Two Cold Calls Are Raw Equal	Verify that with no cache warm-up, no historical call state, and identical inputs, the reference convolution path does not drift between calls.	✓