MOLT: A SCALABLE PYTORCH-NATIVE TRAINING FRAMEWORK FOR AGENTIC REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic reinforcement learning requires rapid experimentation with agents and learning algorithms, yet large policies and long, multimodal trajectories demand substantial distributed infrastructure. We present MOLT, a lightweight, PyTorch- and Hugging Face-native framework that brings these goals together through four contributions. MOLT combines direct loading of Hugging Face models with experimentally validated trillion-parameter scalability in approximately 9.2K lines of framework code. Unified OpenAI- and Anthropic-compatible interfaces integrate existing agents with automatic handling of context compaction. Fully asynchronous training overlaps agent rollouts and policy optimization, accommodating variable agent execution times. Distributed experience storage removes centralized rollout-memory bottlenecks for long, multimodal trajectories. We experimentally validate the complete RL training pipeline on a one-trillion-parameter policy and demonstrate sustained learning with a 30B mixture-of-experts agent, establishing MOLT as a lightweight foundation for large-scale agentic RL research.

1 INTRODUCTION

Agentic reinforcement learning (RL) turns model training into an iterative process of modifying agents, rewards, and optimization algorithms. Advances in reasoning policies (DeepSeek-AI et al., 2025) and interactive multimodal tasks (Xie et al., 2024) increase both the scale of the policy and the complexity of the interactions used to train it. The systems challenge is to support changes to tools, rewards, and estimators while preserving distributed execution at trillion-parameter scale.

Existing RL frameworks provide distributed trainers, serving engines, orchestration, and algorithm implementations through different integration choices. HybridFlow/verl (Sheng et al., 2025b) emphasizes flexible distributed execution, OpenRLHF (Hu et al., 2024) provides a Ray-based training stack, and AReaL (Fu et al., 2025) develops asynchronous RL at scale. For agentic research, model scale must be paired with a small, understandable implementation and explicit interfaces for changing the experiment.

We present MOLT, a lightweight PyTorch-native and Hugging Face-native framework for agentic RL research. Researchers can load Hugging Face models directly and configure parallel training within the same workflow. Its framework-owned RL implementation comprises approximately 9.2K Python code lines, and the complete rollout, weight-refit, and training-update path has executed on a one-trillion-parameter policy. Agent behavior and algorithmic changes remain in Python modules and functions.

Agent pipelines must also accommodate context compaction, large visual trajectories, and variable tool latency. MOLT connects standard SDK calls, automatic trajectory segmentation, and distributed experience delivery within one asynchronous training path.

1. **A lightweight PyTorch- and Hugging Face-native framework at 1T scale.** Direct Hugging Face model loading and configurable parallel training combine research flexibility with large-model scalability.

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

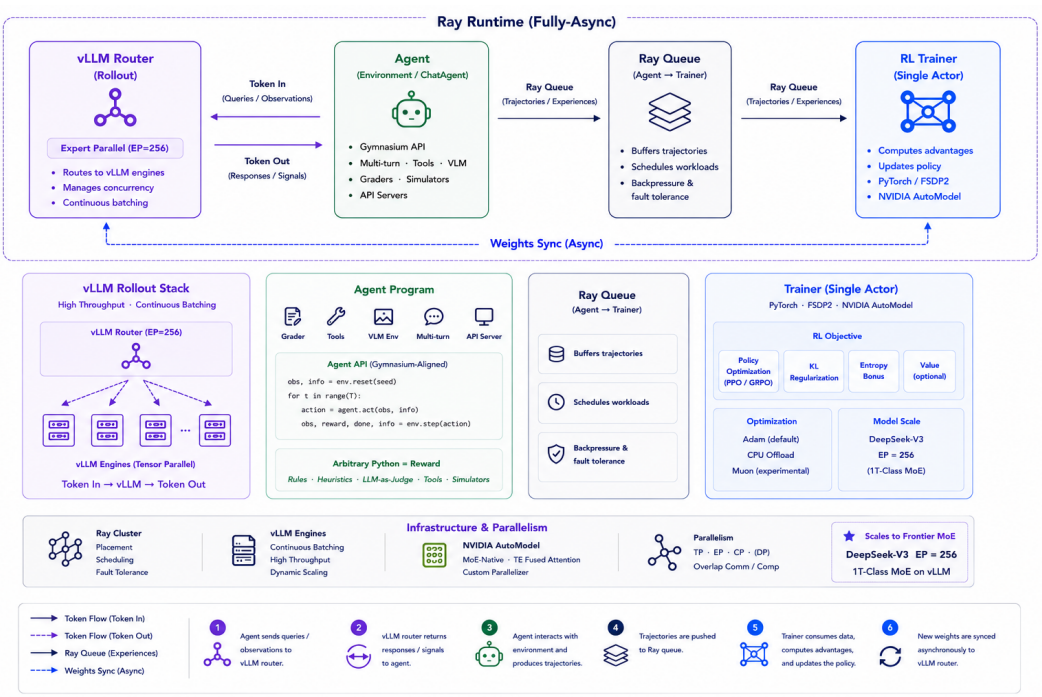


Figure 1: **MOLT architecture and research workflow.** Python agents, vLLM rollout engines, and a PyTorch/FSDP2 actor are coordinated through Ray. The overview, component details, and execution sequence show how agent interactions become training trajectories and how policy updates return to the rollout stack.

2. **Unified black-box agent integration with automatic context-compaction handling.** OpenAI and Anthropic clients share one capture path that preserves generated tokens and splits trajectories when the agent rewrites its context.
3. **Fully asynchronous training.** Persistent rollout dispatch, completed-group batch assembly, and direct policy-weight refit connect agent execution to optimization while retaining generation-time behavior probabilities.
4. **Distributed rollout experience storage.** Workers publish heavy trajectory payloads to Ray's object store, and trainer ranks retrieve their assigned experiences by reference, removing the centralized full-batch gather from the training handoff.

2 FRAMEWORK OVERVIEW

MOLT separates agent execution, rollout serving, and policy optimization through Ray (Moritz et al., 2018). Python agents interact with vLLM (Kwon et al., 2023); their trajectories feed a PyTorch actor built on NeMo AutoModel and FSDP2. Updated weights return directly to the serving engines (Figure 1).

A `ChatAgent` controls model calls and returns a task reward; an `Env` supplies transitions while the framework manages generation. Both produce the same training representation. Rewards, advantage estimators, and policy losses are separate Python extension points that reuse the shared transport and runtime.

The controller schedules work using lightweight records; heavy trajectory payloads reside in the distributed object store. Sections 3 and 4 describe agent integration and execution.

3 AGENT INTEGRATION EXAMPLES

3.1 USER-DRIVEN AGENT: A STANDARD OPENAI CLIENT

A user-driven agent invokes a standard SDK through `ChatAgent.run` and returns a task reward. The session endpoint captures its training trace. Below, `grade` supplies the reward, and `AgentRunner` connects the agent to training.

```python
from openai import AsyncOpenAI
from molt.agents import ChatAgent, ChatAgentRunner, Result

class MyAgent(ChatAgent):
    async def run(self, ctx):
        client = AsyncOpenAI(base_url=ctx.base_url,
                             api_key=ctx.api_key)
        response = await client.chat.completions.create(
            model=ctx.model_name,
            messages=list(ctx.messages),
            max_tokens=ctx.sampling_params.max_tokens,
            temperature=ctx.sampling_params.temperature,
        )
        text = response.choices[0].message.content
        return Result(reward=grade(text, ctx.label))

class AgentRunner(ChatAgentRunner):
    def __init__(self):
        super().__init__(MyAgent)
```

The agent can extend this call into its existing tool loop. The endpoint records token IDs, behavior probabilities, and segment transitions after context compaction; user code supplies the reward.

3.2 THE SAME CAPTURE PATH THROUGH ANTHROPIC

The Anthropic client points to `ctx.session_url`, the session root without the `/v1` suffix. Its SDK appends `/v1/messages`; the OpenAI base URL includes `/v1`. For a text request, the corresponding model call is:

```python
from anthropic import AsyncAnthropic

client = AsyncAnthropic(base_url=ctx.session_url,
                        api_key=ctx.api_key)
response = await client.messages.create(
    model=ctx.model_name,
    messages=[{"role": "user", "content": ctx.prompt}],
    max_tokens=ctx.sampling_params.max_tokens,
)
text = response.content[0].text
```

Both interfaces share the same model, reward path, and automatic trajectory capture.

3.3 FRAMEWORK-DRIVEN ENVIRONMENT

The `Env` path complements the user-driven SDK interfaces above and follows the environment-oriented pattern of Gym (Brockman et al., 2016). Here the framework manages model generation, and the task implements its transition and reward logic. The task supplies `grade`.

```python
from molt.agents import Env, Result, StepEnvRunner

class MathEnv(Env):
    async def step(self, state):
        reward = grade(state["action_text"], state["label"])
        return Result(reward=reward, terminated=True)

class AgentRunner(StepEnvRunner):
    def __init__(self):
        super().__init__(MathEnv)
```

3.4 Algorithm Extensions and Experiment Workflow

Change an estimator or loss. An advantage estimator takes rewards, prompt groups, and the context needed for token alignment. A new estimator is expressed as a function selected by name; its outputs pass through the shared loss path. Batch filters act before training and preserve complete groups for group-based estimators.

Launch and inspect. Single-node and Slurm recipes use the same CLI and agent modules. Per-step logs include reward means, within-group reward variation, response lengths, evaluation pass@n, and generation, actor-training, weight-broadcast, and total-step timings. The logs also expose the fraction of samples retained by the consistency gate.

Table 1: Research extension points. Agent behavior and algorithmic changes enter through explicit Python interfaces.

Research change	Interface	Quantities to inspect
Environment behavior	`Env.step(state)`	Action, observation, termination
Agent control flow	`ChatAgent.run(ctx)`	Model calls, tools, segment boundaries
Task reward	Python grading and `Result`	Rollout reward and group statistics
Advantage estimator	Function of rewards, groups, context	Per-token advantages and returns
Policy objective	Shared policy-loss path	Ratios, masks, normalization, gradient

4 System Design

4.1 A PyTorch-Native Path to Trillion-Parameter Policies

MOLT builds the actor on NeMo AutoModel and FSDP2 (PyTorch Contributors, 2025), using the sharding approach developed by PyTorch FSDP (Zhao et al., 2023). Tensor, expert, and context parallelism compose with this path; corresponding vLLM controls configure rollout execution (Kwon et al., 2023). Optimizer CPU offload adds a memory-capacity control without changing the agent interface.

Training and serving parallelism are configured independently (Table 2), preserving the agent interface at 1T scale (Section 5.2).

For MoE models, rollout routing replay (Ma et al., 2025) records expert choices during generation and reuses them in the actor. Token alignment, route alignment, and numerical consistency diagnostics accompany distributed execution. Prefix caching, CUDA graphs, and speculative decoding enter through the serving engine, and per-stage measurements expose their effect on the research workload.

4.2 Unified Black-Box Interfaces and Context Compaction

Both SDK interfaces in Section 3 feed the same local capture path. The agent retains control over its execution and context management, while the server records token IDs and log-probabilities at generation time. The training trace therefore comes from the serving engine's outputs without requiring agent-side tracing calls or explicit SDK requests for log-probabilities.

Table 2: Principal configuration controls described by the implementation. Actor parallelism composes with FSDP2, and serving controls configure each rollout engine.

Control	Purpose
`--fsdp.tp_size`	Actor tensor parallelism
`--fsdp.ep_size`	Actor expert parallelism
`--fsdp.cp_size`	Actor context parallelism
`--fsdp.offload optimizer`	Adam states in host memory
`--vllm.tensor_parallel_size`	Tensor parallelism per serving engine
`--vllm.enable_expert_parallel`	Expert-parallel serving
`--vllm.data_parallel_size`	Serving data parallelism
`--vllm.max_num_batched_tokens`	Serving scheduler token budget
`--vllm.mtp_num_speculative_tokens`	Number of speculative tokens; zero disables
`--train.routing_replay`	Reuse rollout expert choices in the actor
`--actor.freeze_moe_router`	Freeze the router; distinct from replay

The canonical trace contains token IDs, action ranges, behavior log-probabilities, rewards, and multimodal tensors. For a recorded token x_i and policy-action mask m_i, the training contract is

$$m_i = 1 \quad \Longrightarrow \quad x_i \text{ is the token ID returned by the rollout engine.} \tag{1}$$

Prompts, images, and tool observations provide the context for those actions. A shared chat-format data path applies the model template once, preserving the input structure across framework-driven environments and user-driven agents.

Automatic handling of context compaction. An agent can summarize or remove earlier turns while retaining control of its context-management policy. The capture server checks whether each request extends the recorded message history and whether the rendered conversation preserves its prefix. A continuation appends the new context and sampled actions to the current segment; a rewritten history or template prefix starts a new segment. Each segment records the generated token IDs, action ranges, and behavior log-probabilities associated with its context.

All segments from one rollout retain the same rollout identifier and terminal reward. Group-based estimators count that reward once and map the rollout-level result back to its segments, whose generated actions remain available for optimization. Thus, an agent that replaces a long conversation with a summary can continue the same rollout while the trainer retains the contexts under which its earlier and later actions were generated.

4.3 Fully Asynchronous Training

The persistent rollout pool dispatches individual rollouts to agent workers and gathers their results into prompt groups, each containing the samples required by a group-based estimator. The trainer assembles batches from completed groups while further rollouts remain in flight on disaggregated GPUs. This separates optimizer scheduling from a global rollout round and accommodates differences in tool and environment latency. Queue depth controls the amount of buffered work and its policy staleness; reference-model workers and a PPO critic are enabled when required by the objective.

For partial rollout, the runtime pauses generation, broadcasts updated actor parameters over NCCL directly to the serving engines, and resumes retained requests. The request router handles inference traffic, while weight transfer follows the actor-to-engine path. Retaining requests preserves their generated prefixes across refits. Because a continuation may use a newer policy, every action token retains the log-probability recorded when it was sampled, and partial rollout requires importance correction to be enabled. The resulting training path combines continuous agent dispatch, complete-group consumption, and explicit weight-refit boundaries.

4.4 Distributed Rollout Experience Storage

Visual-agent rollouts can contain many screenshots, multimodal tensors, and long action histories. With many concurrent agents, gathering a full experience batch in one controller concentrates this

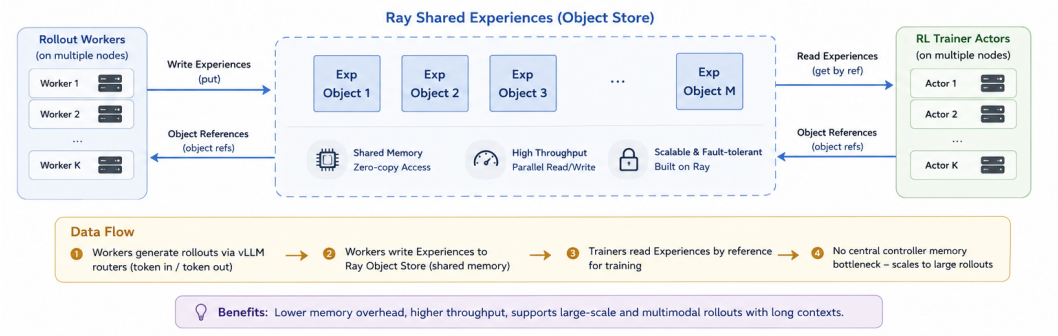


Figure 2: **Ray shared experiences for distributed rollout.** Rollout workers publish experience objects to Ray's object store and pass references through the orchestration path. Trainer actors retrieve experiences by reference. The schematic shows the worker-to-store-to-trainer data flow and the separation of experience payloads from their object references.

memory demand. OSWorld-style desktop interaction (Xie et al., 2024) motivates distributing the payloads together with rollout execution.

When a rollout completes, its worker converts each trajectory segment into an experience. The experience's `offload()` method places token sequences, attention masks, behavior log-probabilities, expert-route records, multimodal inputs, and images in the producing node's Ray object store. It clears these fields from the transported object and retains an object reference; the rollout queue carries that reference together with lightweight training fields. Length-aware assignment allocates experiences to data-parallel ranks, whose `reload()` calls retrieve their assigned payloads through Ray.

The storage unit is a trajectory segment, and trainer ranks materialize only their assigned experiences. Local shared-memory access and inter-node transfer use the same object-reference interface (Moritz et al., 2018) (Figure 2).

4.5 Behavior Probabilities and Loss Normalization

The loss path distinguishes the policy used for sampling from the actor's recomputed probabilities. Asynchronous and training–inference corrections use recorded behavior log-probabilities and a configurable rejection or weighting rule (Liu et al., 2025a). A trajectory that mixes policy versions cannot be represented by one sequence-wide sampling policy without accounting for those versions. Per-token probability records preserve the quantities needed by the correction; masking and clipping trade estimator bias against the influence of mismatched samples.

For the reported token-mean configuration, all microbatches in an optimizer window use the same action-token denominator. If ℓ_i is a configured per-token loss contribution, including any correction or rejection weight, the normalization target is

$$\mathcal{L} = \frac{\sum_{i \in \mathcal{W}} m_i \ell_i}{\sum_{i \in \mathcal{W}} m_i}, \tag{2}$$

where $\mathcal{W}$ spans the whole distributed optimizer window and has a nonzero denominator. Rejection weights belong to ℓ_i; this expression does not renormalize by the number of accepted sequences. For a fixed optimizer window and objective, token weights are invariant to repartitioning across microbatches and data-parallel ranks.

Advantage estimators include REINFORCE++ (Hu et al., 2025), a group-mean baseline, RLOO (Ahmadian et al., 2024), GRPO (Shao et al., 2024), Dr. GRPO (Liu et al., 2025b), and GAE with a PPO critic (Schulman et al., 2016; 2017). Dynamic filtering can remove degenerate prompt groups and backfill complete groups, following the motivation of DAPO (Yu et al., 2025).

Table 3: Implementation footprint and integration structure. LOC counts framework-owned Python code in the RL execution path under the scope described in the text.

Framework	RL LOC	Training path	Rollout engine	Configuration
MOLT	∼9.2K	AutoModel / FSDP2	vLLM	CLI flags
OpenRLHF	∼7.2K	DeepSpeed ZeRO-3	vLLM	CLI + scripts
verl	∼62K	FSDP(2) / Megatron	vLLM / SGLang / HF	Hydra + YAML
slime	∼25K	Megatron	SGLang	CLI; optional YAML

5 EVALUATION

We evaluate implementation footprint, trillion-parameter execution, sustained learning with a 30B MoE agent, pipeline throughput, and 35B serving and memory efficiency.

5.1 FRAMEWORK FOOTPRINT

MOLT's framework-owned RL implementation comprises approximately 9.2K Python code lines.[1] The count follows the RL entry point through trainer, rollout, orchestration, experience processing, advantage estimation, loss computation, and imported model, utility, parallelism, and kernel code. Inference components used by RL are included; unrelated training paths, vendored dependencies, tests, examples, scripts, documentation, blank lines, and comments are excluded. Slime's scope includes its core package and in-repository model plugins.

Code footprint and structural complexity. Table 3 places MOLT alongside OpenRLHF (Hu et al., 2024), verl (Sheng et al., 2025b), and slime (Zhu et al., 2025). MOLT uses approximately 15% of verl's code and 37% of slime's code under this counting method. OpenRLHF has a comparable implementation footprint. MOLT combines this compact scale of implementation with PyTorch-native model parallelism and the demonstrated 1T execution.

MOLT concentrates on AutoModel and vLLM, with `Env` and `ChatAgent` as the agent extension points. OpenRLHF offers step-based `AgentInstance` and full-loop `AgentExecutor` interfaces; verl uses `AgentLoopBase.run`; slime accepts custom generation and reward functions. These entry points characterize integration structure alongside code size.

5.2 END-TO-END EXECUTION AT 1T PARAMETERS

MOLT has executed the complete asynchronous path on a 1T-parameter policy: agent rollout, weight refit, and a training update. The run retains the PyTorch-native actor and agent interfaces, with model parallelism and distributed execution supplied through configuration.

5.3 LEARNING AND ROLLOUT–TRAINER AGREEMENT

To validate sustained learning, we train Qwen3-30B-A3B as a Python-tool agent in MOLT using FlashREINFORCE (Hu et al., 2026a). The agent uses a Python tool for up to ten turns on 7.5K deduplicated DAPO-Math prompts. Each update uses 128 single-sample prompts at policy lag approximately eight, with routing replay and a sequence trust-region threshold of 10^{-3}. Training uses AdamW (initial learning rate 10^{-6}, cosine decay, weight decay 0.1), no reference KL, and 18,432-token contexts across 16 actor and 16 rollout GPUs. AMC23, Minerva, and AIME2025 are evaluated every 50 updates using avg@4, temperature 0.7, and top-$p = 0.7$.

The run completes 1,464 updates. Mean evaluation accuracy rises from 46.38% at the first recorded evaluation (update 50) to 66.36% at update 1,450, and remains between 63.74% and 66.84% over updates 1,000–1,450 (Figure 3). Final AMC23, Minerva, and AIME2025 accuracies are 94.38%, 41.36%, and 63.33%, respectively. This run validates effective learning through MOLT's asynchronous agent-to-update path over approximately 1,500 updates.

[1]The MOLT count was recorded on July 20, 2026. Comparison snapshots: OpenRLHF `b3d2927`, verl `86e8123`, and slime `243773c`.

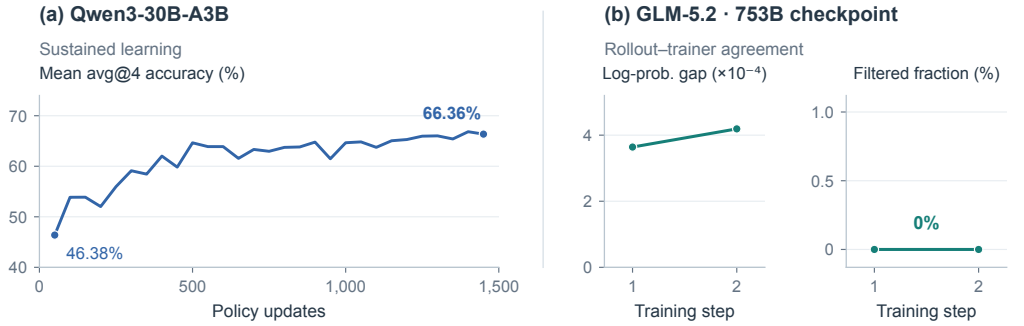


Figure 3: **Learning and rollout–trainer agreement with MOLT.** (a) Mean avg@4 accuracy on AMC23, Minerva, and AIME2025 for Qwen3-30B-A3B across 29 evaluations. (b) GLM-5.2 (753B checkpoint): logged rollout–trainer log-probability gap (`vllm_kl`, left) and filtering fraction (`is_filter_ratio`, right) at the first two recorded steps.

Rollout–trainer agreement on GLM-5.2. We also evaluate rollout–trainer agreement on the 753B-parameter GLM-5.2 MoE checkpoint (Z.ai, 2026). The first two recorded steps show a mean rollout-to-trainer log-probability gap below 5×10^{-4} and zero importance-sampling filtering (Figure 3b), demonstrating close probability agreement across serving and training at this model scale.

5.4 PIPELINE THROUGHPUT AGAINST A MEGATRON-BASED STACK

We compare MOLT (AutoModel + vLLM) with slime (Megatron-Core + SGLang) on Qwen3-30B-A3B in bf16. Both use two eight-H100 nodes, split into eight training and eight rollout GPUs, with asynchronous execution. The workload uses 2K deduplicated DAPO-Math prompts, 32 prompts with four samples per step, a 16,384-token context, and an 8,192-token response cap. Temperature and top-p are 1.0, and top-k is disabled. Both use Adam with identical hyperparameters, optimizer CPU offload, full recomputation, microbatch size one, and no reference model, packing, or dynamic batching.

MOLT uses FSDP2 DP8/EP8/TP1 with streaming asynchronous execution; slime uses TP4 with sequence parallelism, CP1/EP8, and its one-step asynchronous mode. Each rollout engine uses TP8, prefix caching, CUDA graphs, and 128 concurrent sequences.[2] Table 4 reports the original measured step times and workload-normalized throughput.

Table 4: Pipeline throughput on Qwen3-30B-A3B. Step time is the mean ± standard deviation of per-run mean step times (steps 2–10), over three independent runs. Tokens/GPU/s uses approximately 880K generated tokens per step divided by mean step time and 16 GPUs.

Configuration	Step (s)	Tokens/GPU/s
MOLT (AutoModel + vLLM)	119.4 ± 2.3	461
slime (Megatron-Core + SGLang)	109.5 ± 10.3	502

MOLT reaches approximately 92% of slime's workload-normalized throughput with its compact PyTorch-native stack. The long-run experiment in Section 5.3 complements this throughput comparison with sustained agent learning.

5.5 SERVING AND MEMORY ON A 35B MOE WORKLOAD

The single-framework measurements use the Qwen3.6-35B-A3B geo3k multi-turn tool-use recipe on two nodes with eight H100 training GPUs and eight H100 rollout GPUs. The configuration uses a 32K context, CP8/EP8/TP1 on the actor, four prompts with four samples each, temperature 1.0,

[2]Benchmark snapshots: MOLT `cb2cae11`; slime `5d7296a7`; Megatron-Core `1dcf0daf`. These are distinct from the footprint snapshots.

Table 5: Serving and memory measurements on the 35B workload. Timing results refer to the named execution stage.

Feature / quantity	Baseline	Enabled	Interpretation
MTP / generation time	329 s	64 s	$5.14\times$ stage speedup
Offload / peak actor memory	64.7 GB	46.4 GB	18.3 GB saved (28.3%)
Offload / actor train time	213 s	251 s	17.8% increase
Prefix cache / re-prefill	—	0.05 s	Cache-hit latency

routing replay, and the `seq-mask-tis` sequence-level importance gate. Table 5 summarizes the measurements and their scope.

Enabling the checkpoint's multi-token-prediction (MTP) head reduces generation time from 329 to 64 seconds ($5.14\times$). Optimizer offload exchanges 18.3 GB of GPU memory for 38 seconds of additional actor-training time. Session-consistent prefix caching yields a 0.05-second cache-hit re-prefill. The consistency gate retains all sequences. These controls expose complementary time and memory tradeoffs, measured through the same launch path.

6 Related Work

Research and production RL frameworks. HybridFlow/verl (Sheng et al., 2025b) coordinates distributed RL roles with a hybrid programming model. OpenRLHF (Hu et al., 2024), from which MOLT originated, established a Ray-based training and serving decomposition. Slime (Zhu et al., 2025) combines RL orchestration with Megatron-Core and SGLang (Zheng et al., 2024). MOLT combines a compact PyTorch training implementation with 1T execution, unified agent capture, and distributed experience delivery.

Asynchronous execution. AReaL (Fu et al., 2025) studies staleness-aware asynchronous RL. StreamRL (Zhong et al., 2025) addresses pipeline and skewness bubbles; Laminar (Sheng et al., 2025a) decouples generation and weight distribution; DORA (Hu et al., 2026b) uses multiple policy versions. RolloutPipe (Chen et al., 2026) overlaps prompt-group generation and training under on-policy constraints, while Relax (Zhang et al., 2026b) and RollArt (Gao et al., 2025) address multimodal and agentic execution. MOLT combines disaggregation, bounded buffering, and behavior-probability correction with distributed experience delivery in its asynchronous training path.

Agent integration and consistency. Agent Lightning (Luo et al., 2025) decouples agent execution from training. Polar (Xu et al., 2026), following ProRL Agent (Zhang et al., 2026a), captures trajectories at a provider-compatible proxy. MOLT integrates local capture, context segmentation, and a PyTorch trainer in one workflow. R3 (Ma et al., 2025) supplies the routing-replay mechanism used for MoE consistency; masked importance correction addresses another part of the training–inference mismatch (Liu et al., 2025a).

7 Conclusion

MOLT combines standard agent APIs, context segmentation, asynchronous training, and distributed experience storage in approximately 9.2K lines of PyTorch-native RL code. End-to-end execution at 1T parameters and sustained learning over approximately 1,500 updates on a 30B MoE agent demonstrate the implementation's scale and training capability. The explicit agent and algorithm interfaces preserve research flexibility across these workloads. Extending this design to 2T policies is the next scaling target.

AI Use Statement

Generative AI tools assisted software development in this project. During manuscript preparation, Codex assisted with manuscript drafting and editing, structural feedback, LaTeX formatting, literature retrieval and reference checking, interpretation of reported measurements, and suggestions for

additional experiments. The authors conducted the experiments reported in the paper. The authors are responsible for the final text, claims, code, and measurements.

REPRODUCIBILITY STATEMENT

The workload settings, measurement definitions, and footprint counting scope are described in Section 5. Section 3 gives the agent interfaces, and Table 2 lists the principal scaling controls. These descriptions identify the configuration and scope associated with each reported result.

ETHICS STATEMENT

MOLT supports agent research with explicit control over tool interfaces and trajectory collection. Responsible use requires task-appropriate execution isolation, permissions for tools and datasets, and privacy-aware handling of stored trajectories. These considerations apply to the environment and deployment configuration.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12248–12267, 2024. doi: 10.18653/v1/2024.acl-long.662. URL https://aclanthology.org/2024.acl-long.662/.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Rongjian Chen, Jianmin Hu, Kejiang Ye, and Minxian Xu. RolloutPipe: Overlapping pipelined rollout and training in disaggregated on-policy LLM reinforcement learning. In *Proceedings of the 2026 International Conference on Cognitive Computing (ICCC 2026)*, 2026. URL https://arxiv.org/abs/2606.26997.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, et al. RollArt: Disaggregated multi-task agentic RL training at scale. *arXiv preprint arXiv:2512.22560*, 2025.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 2024.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. REINFORCE++: Stabilizing critic-free policy optimization with global advantage normalization. *arXiv preprint arXiv:2501.03262*, 2025.

Jian Hu, Yifan Zhang, Hao Zhang, Binfeng Xu, Shaokun Zhang, Hongqing Peng, Zhiding Yu, Pavlo Molchanov, Jan Kautz, and Yi Dong. FlashREINFORCE: Critic-free single-rollout asynchronous RL for agentic language models, 2026a. URL https://github.com/yifanzhang-pro/FlashREINFORCE. Technical report.

Tianhao Hu, Xiangcheng Liu, Yuchun Miao, Youshao Xiao, Hongyu Zang, Yang Zheng, Xuan Huang, Jinrui Ding, Yufei Zhang, Yu Yang, Yi-Kai Zhang, Yueqing Sun, Chengcheng Han, Xiandi Ma, Wei Wang, Qi Gu, Yerui Sun, Yuchen Xie, and Xunliang Cai. DORA: A scalable asynchronous reinforcement learning system for language model training. *arXiv preprint arXiv:2604.26256*, 2026b. URL https://arxiv.org/abs/2604.26256.

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 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Jiacai Liu, Yingru Li, Yuqian Fu, Jiawei Wang, Qian Liu, and Yu Shen. When speed kills stability: Demystifying RL collapse from the training-inference mismatch. Online article, https://yingru.notion.site/When-Speed-Kills-Stability-Demystifying-RL-Collapse-from-the-Training-Inference-Mismatch-271211a558b7808d8b12d403fd15edda, 2025a.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025b.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. Agent lightning: Train ANY AI agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing MoE reinforcement learning by aligning training and inference routers. *arXiv preprint arXiv:2510.11370*, 2025.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

PyTorch Contributors. Getting started with fully sharded data parallel (FSDP2). https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html, 2025. PyTorch documentation; last updated September 2, 2025; accessed September 16, 2026.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, et al. Laminar: A scalable asynchronous RL post-training framework. *arXiv preprint arXiv:2510.12633*, 2025a.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, pp. 1279–1297, 2025b. doi: 10.1145/3689031.3696075.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, pp. 52040–52094, 2024. doi: 10.52202/079017-1650. URL

https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Binfeng Xu, Hao Zhang, Shaokun Zhang, Songyang Han, Mingjie Liu, Jian Hu, Shizhe Diao, Zhenghui Jin, Yunheng Zou, Michael Demoret, Jan Kautz, and Yi Dong. Polar: Agentic RL on any harness at scale. *arXiv preprint arXiv:2605.24220*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Z.ai. GLM-5.2, 2026. URL https://huggingface.co/zai-org/GLM-5.2. Official model card and checkpoint.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, Zhiding Yu, Jan Kautz, and Yi Dong. ProRL agent: Rollout-as-a-service for RL training of multi-turn LLM agents. *arXiv preprint arXiv:2603.18815*, 2026a.

Liujie Zhang, Benzhe Ning, Rui Yang, Xiaoyan Yu, Jiaxing Li, Lumeng Wu, et al. Relax: An asynchronous reinforcement learning engine for omni-modal post-training at scale. *arXiv preprint arXiv:2604.11554*, 2026b.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, et al. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime, 2025. GitHub repository; accessed September 16, 2026.