Molt: A Scalable PyTorch-Native Training Framework for Agentic Reinforcement Learning

Jian Hu, Huiying Li, Hao Zhang, Binfeng Xu, Yifan Zhang, Shaokun Zhang, Hemil Desai, Michael Demoret, Pavlo Molchanov, Jan Kautz, Yi Dong

NVIDIA

{jianh,yidong}@nvidia.com

Abstract

Agentic reinforcement learning research is constant algorithm modification, new estimators, new pipeline stages, new rollout schemes, and in mainstream frameworks each change threads through layers of trainer, distributed backend, and rollout glue: the cost lands on the researcher at every iteration. Molt is a PyTorch-native training framework built to keep that cost small: a codebase compact and clean enough for a researcher to hold in their head, and for an AI coding assistant to read and reason about in its entirety, so the algorithm flow can be traced and changed end to end. The agent is an ordinary program, and one asynchronous loop trains multimodal and mixture-of-experts policies while never training on a token it did not generate, consistent in tokens, policy versions, and model semantics. Leanness does not cost performance: under a matched, fully asynchronous protocol, Molt is statistically comparable to a state-of-the-art Megatron-based stack. Molt is open source and provides recipes and containers at https://github.com/NVIDIA-NeMo/labs-molt.

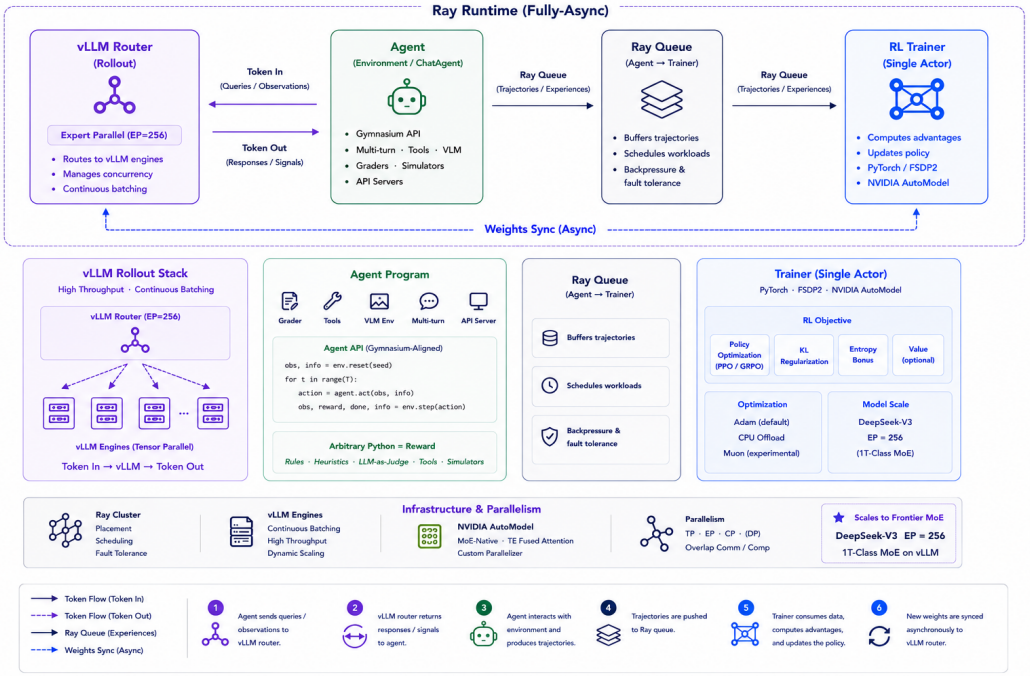

Figure 1: **The whole system: three components and one loop.** Molt composes user agents in plain Python, vLLM rollout engines behind a request router, and a single FSDP2 policy actor on NeMo AutoModel around one Ray asynchronous queue that implements the streaming pool and partial rollout. Trajectories flow token-exactly from the engines through the queue into training, and weight refit returns over NCCL directly to the engines, bypassing the request router.

1. Introduction

Prototyping a new RL algorithm, a different advantage estimator, an extra filtering stage in the experience pipeline, a modified rollout scheme, should be an afternoon's edit. In mainstream frameworks it means tracing the change through several layers of trainer, distributed backend, and rollout-engine glue, and that cost lands on the researcher at every iteration, because agentic RL research is constant algorithm modification. The layers exist for good reason: as workloads moved from single-step preference tuning to agentic ones, multi-turn tool use, code execution, vision-language environments, long-horizon interaction (Guo et al., 2025; Shao et al., 2024; Xie et al., 2024), mainstream stacks were architected for ultra-large-scale training, and their multi-backend structure, separate rollout engines, distributed trainers, controllers, registries, configuration layers, is the price of hyperscale scalability. That is rational engineering for its target. For research it is a poor default: the researcher inherits hyperscale complexity without needing its specialization, and understanding or changing the framework comes to cost more than expressing the hypothesis under study.

Research infrastructure has a different objective: minimize the distance from an idea, about an agent, a reward, an algorithm, to a trustworthy experiment, while retaining the performance that large policies demand. Source code then becomes part of the user interface. A researcher should be able to trace one sample from agent invocation to policy loss; the AI coding assistants that increasingly work alongside that researcher (e.g., Claude Code) should be able to navigate the same path without reconstructing hidden registries or backend-specific control flow. Molt therefore adopts an explicit priority: human readability is the primary design criterion, with navigability by AI coding assistants as a secondary one (Sec. 2).

Readability alone is insufficient because agentic online RL has unusually quiet failure modes. The serving engine and actor evaluate nominally the same policy, yet tokenization, sampling transforms, multimodal rendering, weight versions, or mixture-of-experts (MoE) routing can differ without raising an error. The symptom is merely a biased or rejected gradient. Agentic workloads add another boundary: existing harnesses own context management, tools, and control flow, while trainers usually expect framework-specific environments. Recent work identifies a step-granular trajectory protocol as a missing primitive for online agent learning (Yan et al., 2026); harness-side systems capture such trajectories at the model API (Luo et al., 2025; Xu et al., 2026).

Molt is a lean, high-performance, PyTorch-native framework built for this research loop. It composes Ray (Moritz et al., 2018), vLLM (Kwon et al., 2023), and NeMo AutoModel around one disaggregated asynchronous loop (Fig. 1); because none of them is forked, every upstream improvement, new models, kernels, serving features, arrives the day it ships. Environments remain ordinary Python: the framework can drive an `Env`, or an existing agent can use a stock OpenAI/Anthropic SDK through a loopback capture server. A persistent prompt-group pool overlaps generation and training without draining the engines. The actor remains standard PyTorch while FSDP2 composes with native tensor, expert, and context parallelism.

The small surface is organized around three correctness invariants. *Token identity*: the sampled token ids, rather than a retokenized transcript, define the trajectory. *Policy-version semantics*: trainable tokens retain their behavior-policy log-probabilities, and asynchronous use is explicitly corrected. *Forward consistency*: rollout and actor execution must agree on model semantics, including multimodal expansion and MoE routing. These invariants connect readability to technical correctness: each has one data representation, one implementation path, and a fail-fast check at unsupported combinations. Their net effect is simple to state: every trained token is exactly the token that was generated, so experiments mean what they say.

The thesis is that complexity is not the price of capable RL infrastructure, it is a choice inherited from hyperscale. Molt bets that the entire algorithm flow can stay small enough to read as a whole, by a researcher and by the AI coding assistants that now share the work, and the evaluation shows the bet costs nothing in throughput. What distinguishes Molt from prior systems is therefore not any single mechanism but the combination: a codebase several times smaller than the Megatron-based production stacks it matches in throughput, and a design a researcher can read, modify, and extend in place, hackable in the way research code needs to be.

The paper makes four contributions:

- **A principled, readability-first framework design.** We codify five design principles, readability for humans and AI coding assistants, minimal single-backend code, performance parity as a constraint, modularity that follows the RL algorithm, and correctness in the details, and realize them in a compact RL path with a single entry point, a single agent module, and localized mechanisms instead of backend abstractions (Sec. 2 and 4.1).
- **A token-first agent boundary.** An agent that runs against a standard OpenAI or Anthropic SDK trains as-is, with no integration code: SDK traffic is captured as token ids and log-probabilities across two plain-Python agent forms sharing one chat-format data path, and trajectories segment automatically when context compaction rewrites a prefix (Sec. 3.2 and 3.3).
- **A fast asynchronous PyTorch path.** Persistent prompt-group streaming, pause/refit/resume, direct NCCL weight synchronization, FSDP2-native tensor, expert, and context parallelism, optimizer offload, and rollout routing replay carry multimodal MoE training without changing the dense-model programming model (Sec. 3.1, 3.4 and 3.5).
- **An open implementation and a matched-protocol evaluation.** We quantify serving acceleration and memory trade-offs on a 35B multimodal MoE workload, show throughput statistically comparable to a state-of-the-art Megatron-based stack under an explicit matched protocol with every asymmetry disclosed, and carry the same asynchronous training path end to end onto a 700B MoE at expert parallelism 256, one lean loop unchanged from a 4B dense model (Sec. 3.5 and 4).

The paper follows this arc: Sec. 2 derives design principles from the problem above, Sec. 3 realizes them as a system, Sec. 4 tests whether the resulting leanness costs performance, and Sec. 5 and 6 position the result and lay out what remains.

2. Design Principles

Sec. 1 located the cost of agentic RL infrastructure in a regime mismatch: hyperscale stacks are optimized for the largest training jobs, while research iteration is optimized for the rate of algorithm change. MOLT's position is that reaching large scale does not require hyperscale-specific layering: composing components that are separately hardened at frontier scale covers the same band while staying readable, so scalability is inherited rather than re-implemented (Sec. 3.5). Five principles govern the implementation; each states what it forbids or requires in the codebase and points to where the paper delivers it.

Principle 1: readable by humans and by AI coding assistants. A researcher must be able to read a function once and understand its control and data flow; an AI coding assistant (e.g., Claude Code) must be able to trace a feature from CLI flag to executed branch, tensor, metric, and test without reconstructing hidden control flow. This forbids unnecessary indirection outright, code that needs a second pass is treated as a defect even when it executes correctly, and it is what keeps the framework hackable with assistants in the loop. Sec. 4.1 shows the resulting workflow; the trace in Sec. 3.6 shows the property in use.

Principle 2: minimal code, deliberately one backend. Redundant code is a defect: deletion is preferred over addition, though never at the cost of performance knobs or observability, and a helper must earn its existence with repeated nontrivial use. Most consequentially, MOLT supports exactly one training backend (AutoModel) and one serving engine (vLLM), neither forked, so upstream improvements arrive with nothing to rebase. Multi-backend abstraction is where layered indirection comes from; refusing it removes the layer instead of hiding it. The trade is deliberate, narrower deployment choice, bought for directness, and it is what produces the small codebase of Tab. 1; scale remains configuration on the same backend (Sec. 3.5).

Principle 3: performance on par with the state of the art, as a constraint. Leanness is admissible only if it costs no throughput. Parity with a state-of-the-art Megatron-based stack under a matched protocol is a

design requirement: it forbids simplifications that shift time onto the training path and requires the engines' own optimizations to remain usable through the composed path. Sec. 4.2 and 4.3 verify the requirement.

Principle 4: modularity follows the RL algorithm, not the infrastructure. Components map one-to-one onto the algorithm's own objects, the agent/environment interface, the rollout, the advantage estimator, the loss, rather than onto infrastructure layers. This forbids adapter layers and plugin registries, and it is why an algorithmic edit touches exactly one component: the agent contract (Sec. 3.2) and estimators-as-pure-functions (Sec. 3.6) are the delivered examples, and the concept inventory opening Sec. 3 is this principle's system-level shape.

Principle 5: correctness in the details. Numerical fidelity between generation and training is a first-class guarantee, not a debugging afterthought. The codebase is required to train on exactly the tokens that were generated (token-in/token-out capture, Sec. 3.2), to keep log-probabilities consistent between engine and trainer, and to monitor the residual training–inference mismatch (Liu et al., 2025a) at every step behind a hard sequence-level gate (Sec. 3.6); silent divergence is treated as a bug wherever it arises, including in MoE routing (Sec. 3.5).

Sec. 3 shows the five principles as one running system, anchoring each part of the design to the principle it delivers.

3. The System: Four Concepts, One Loop

This section shows how the principles of Sec. 2 become a system. MOLT has four load-bearing concepts, each mapping one-to-one onto code (Fig. 1): an *agent*, plain Python that produces actions and rewards; a *generator*, token-exact capture against the serving engines; a *trainer*, one visible training loop over a single FSDP2 policy actor; and *estimators and losses*, pure functions of rewards, groups, and the token trace. An algorithmic change touches exactly one of the four. In a stack architected for hyperscale, the same change threads through trainer, distributed backend, engine glue, and configuration indirection; MOLT keeps the concept count small and the mapping direct. The subsections follow one trajectory from agent invocation to policy loss, each anchored to the principle of Sec. 2 it delivers: architecture (Sec. 3.1; Principles 1–2), agent boundary and data path (Sec. 3.2 and 3.3; Principle 4), rollout transport (Sec. 3.4; Principle 5), distributed actor (Sec. 3.5; Principle 3), and algorithm layer (Sec. 3.6; Principle 4).

3.1. Architecture: One Asynchronous Loop

The entire runtime is three components and one loop (Fig. 1), small enough to read and deliberately single-backend (Principles 1–2). Ray (Moritz et al., 2018) provides placement and the asynchronous queue connecting an agent pool, a set of vLLM (Kwon et al., 2023) rollout engines, and a single trainable policy actor built on NVIDIA AutoModel with FSDP2 (Zhao et al., 2023)/EP/CP; reference workers and a PPO critic are optional additional groups. There is no hybrid controller, no per-backend adapter layer, and no separate parameter server. The contract between components is *token-first*: token ids, per-token log-probabilities, action ranges, rewards, and multimodal tensors remain aligned from the engine's sampler to the loss, and no component re-derives tokens from text. One invariant anchors the whole loop, MOLT never trains on a token it did not generate, and around it the token-first contract keeps generation and training consistent without a reconciliation layer.

Streaming pool: asynchrony without new concepts. The pool keeps prompt *groups*, all samples of one prompt, the unit group-baseline estimators need, in flight at all times and emits a training batch as soon as enough groups complete, so the engines never drain while the actor trains; a configurable queue depth decouples training throughput from generation latency, which is heavy-tailed in agentic workloads. The loop is instrumented end to end: every optimizer step reports per-stage timings and rollout statistics, so the effect of an algorithmic change is visible in the same step's logs (Sec. 4.1).

Partial rollout: no discarded requests. A weight update need not discard in-flight requests: MOLT pauses the engines, broadcasts actor shards, and resumes the retained requests. A resumed request can mix policy versions, so every action token keeps the log-probability returned when it was sampled and the loss applies the per-token correction of Sec. 3.6 (Liu et al., 2025a); MOLT refuses to run partial rollout without that correction enabled.

3.2. Agent Contract: The Agent Is Ordinary Python

The agent contract is modularity along the algorithm's own boundary (Principle 4): it imposes exactly one obligation, an RL run names one Python module that exports an `AgentRunner`, and everything else is ordinary code: the reward is arbitrary Python, from graders and sandboxed tools to LLM-as-judge calls and full vision-language environments.

`Env`: the framework owns the LLM loop. In the Gym-aligned (Brockman et al., 2016) form, the framework drives generation, tokenization, multimodal accounting, and per-turn budgets, invoking the user's `step()` after each model action. A complete single-turn environment follows; multi-turn environments return the next observation from the same `Result`, and the framework chains turns and maintains the token trace.

A complete Env agent (framework owns the loop)

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

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

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

`ChatAgent`: the user owns the loop. If existing agent code runs against a stock OpenAI or Anthropic SDK, it trains as-is, no integration code. MOLT launches a loopback chat server in front of the engines; `ctx.base_url` carries a session identifier, and every request through it decodes server-side into one token-exact accumulation, *token-in/token-out* (TITO) capture, the capability that harness-side systems provide at a proxy boundary (Luo et al., 2025; Xu et al., 2026). Retokenization drift is eliminated because token space is never exited, and the agent needs no `extra_body`, no `logprobs=true`, and no session plumbing:

A complete ChatAgent (user owns the loop via a stock SDK)

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

class MyAgent(ChatAgent):
    async def run(self, ctx: ChatContext) -> Result:
        # ctx.base_url carries the session id and auto-captures the
        # token trace -- no extra_body, no logprobs, no plumbing.
        client = AsyncOpenAI(base_url=ctx.base_url, api_key=ctx.api_key)
        resp = await client.chat.completions.create(
            model=ctx.model_name,
            messages=[{"role": "user", "content": ctx.prompt}],
            max_tokens=ctx.sampling_params.max_tokens,
            temperature=ctx.sampling_params.temperature,
        )
        return Result(reward=grade(resp.choices[0].message.content,
                                   ctx.label))

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

The same server exposes both the OpenAI and Anthropic wire protocols, so external harnesses, browser automation, evaluation frameworks, OSWorld-style agents (Xie et al., 2024), drive the policy without modification.

Context compaction as segmentation. Long-horizon agents *compact*: they summarize or drop earlier turns, rewriting the prompt prefix that would otherwise extend one monotonic token-exact trajectory. The chat server detects the rewrite, seals the current segment, and opens a fresh token-exact one; group baselines still see one reward per rollout. No agent-side change is required, so even harnesses whose compaction behavior is opaque stay trainable.

3.3. One Data Path for Both Agent Forms

The data path keeps that modularity intact (Principle 4): one dataset serves both agent forms, each consumes the same chat-format data (`-data.apply_chat_template`), and a single runner attribute decides where the chat template is applied, pre-rendered by the dataset for `Env`, rendered exactly once by the chat server for `ChatAgent`. System turns, tool schemas, and inline images are preserved on both paths. Because the template is applied exactly once either way, the two forms present identical inputs to the rollout engine, and experiments move between agent styles without a data-path confound.

3.4. Transport: Token-Exact by Construction

Transport is where correctness in the details (Principle 5) meets the one-backend rule (Principle 2).

No engine forks. MOLT carries no vLLM patches, so an engine upgrade is a container pin rather than a rebase and every upstream serving improvement is immediately usable; every transport requirement is met client-side against documented endpoints. The constraint doubles as a design forcing function, whenever token-exactness appears to require engine internals, the requirement is relocated to a stable interface instead.

Token-exact transport. Trainer-facing generation uses the engines' token-level interface: prompts enter as token ids, completions return as token ids with per-token log-probabilities, and text never passes through a tokenizer mid-episode. The request router keeps all requests of one rollout, including render calls, on the engine that holds its prefix cache, and vision-language prompts are rendered server-side and realigned once, so an image traverses the transport exactly once and cannot be silently dropped by a text-only endpoint.

Weight refit bypasses the router. Refit is an NCCL broadcast of the parameters from the actor directly to every rollout engine, each loading only the shards it owns; the router carries inference traffic only.

3.5. Scale Is Configuration, Not Migration

The production features below landed as small, local extensions of the same visible loop rather than as new layers, the machinery behind the performance constraint of Principle 3, verified in Sec. 4.

Composable parallelism: 1T-class MoE by configuration. FSDP2 composes with AutoModel-native tensor, expert, and context parallelism, with matching knobs on the vLLM side (Tab. 4 in the appendix). The launch script that trains a dense 4B model also expresses 1T-class MoE: a DeepSeek-V3-class configuration (DeepSeek-AI, 2024) is written as `-fsdp.ep_size 256`, not as a backend migration, the parallel layouts these components validate at frontier scale upstream are expressible here directly. That expressiveness is exercised, not asserted: we have run the full asynchronous loop, rollout, weight refit, and optimizer step, end to end on a 700B MoE at expert parallelism 256, so a change of scale is a change of configuration, not of design (Sec. 6). Unsupported combinations, packed batches under CP among them, are rejected at configuration time by a conservative MOLT-side guard.

MoE consistency. MoE RL has a failure mode dense models lack: the rollout and training routers select experts independently, and small numerical differences can make the two sides evaluate different sparse computation graphs. Rollout routing replay (Ma et al., 2025) closes it locally, the engine returns its per-token expert choices and the actor replays them during training, without touching any other part of the loop.

Engine features arrive as flags. Speculative decoding, prefix caching, and CUDA graphs arrive as engine flags through the composed path; optimizer CPU offload holds Adam states in host memory for the largest actors, compatibly with distributed checkpointing. Sec. 4.2 quantifies the supported paths, and combinations for which the engine does not preserve alignment fail fast rather than degrade silently.

3.6. Algorithm Layer: Estimators Are Functions

The algorithm layer applies Principle 4 directly: estimators are plain functions, and one loss path consumes the canonical token trace.

Estimators. Advantage estimators in MOLT are plain functions of rewards and groups, selected by name with no strategy classes or inheritance hierarchy, a new estimator is one function, not a subclass. The default follows the critic-free REINFORCE++ approach (Hu et al., 2025); REINFORCE with a group-mean baseline, REINFORCE Leave-One-Out (RLOO) (Ahmadian et al., 2024), Group Relative Policy Optimization (GRPO) (Shao et al., 2024), Dr. GRPO (Liu et al., 2025b), Generalized Advantage Estimation (GAE) (Schulman et al., 2016) with a PPO critic (Schulman et al., 2017), and on-policy distillation are each selected by one flag.

One loss normalization. Losses are normalized by a *global whole-batch token mean*: a single denominator, the number of unmasked tokens in the optimizer-step window, is shared by the policy-gradient, KL, and entropy terms, making the update invariant to data-parallel size and gradient-accumulation depth, so changing the cluster layout does not silently change the objective.

Off-policy correction and batch shaping. For asynchronous rollout, the loss applies a per-token importance correction with a sequence-level gate, in the lineage of masked importance sampling for the training–inference mismatch (Liu et al., 2025a). Dynamic filtering in the style of DAPO (Yu et al., 2025) removes degenerate groups and backfills with complete ones, and a force-on-policy option maps one complete multi-turn rollout to exactly one optimizer step when strict on-policy training takes priority over utilization.

One modification, end to end. Consider the algorithmic edit of Sec. 1: a new advantage estimator. In MOLT this is the afternoon's edit the introduction asked for, one pure function of rewards and groups, selected by

	Molt	**OpenRLHF**	**verl**	**slime**
Training backend	AutoModel	DeepSpeed ZeRO-3	FSDP(2)/Megatron	Megatron
Rollout engine	vLLM	vLLM	vLLM/SGLang/TRT	SGLang
RL topology	actor (+critic)	actor+critic+RM	actor+critic+RM	actor+critic+RM
Reward source	agent Python	agent/endpoint/RM	agent/RM/endpoint	rollout fn/RM
Parallelism	TP/EP/CP, MoE-native	ZeRO-3/FSDP	TP/PP/EP/SP	TP/PP/DP/CP/EP
Multimodal	VLM, multi-turn tools	VLM RL	Qwen2.5-VL, Kimi-VL	geo3k VLM
Config surface	CLI flags	CLI + scripts	Hydra + YAML	CLI + YAML
RL code (LOC)[1]	∼8.6K	∼7.2K	∼62K	∼25K
Design center	readable agentic research	RLHF coverage	production breadth	Megatron throughput

Table 1: **Framework comparison: the lean point in the design space.** Molt trains with FSDP2/EP/CP on NVIDIA AutoModel; slime's FSDP backend is experimental; Molt and OpenRLHF orchestrate vLLM under Ray. [1]RL code counts every Python file used by the RL entry path, trainer, rollout, orchestration, experience/advantage/loss, and imported model/utility/parallelism code, excluding pure SFT/DPO trainers, reward-model training, vendored code, tests, examples, and docs; counts trace the import graph from each RL entry point. Measured at verl `86e8123`, slime `243773c`, OpenRLHF `b3d2927` (2026-06-16); Molt 2026-07-07. LOC measures implementation footprint, not usability or correctness.

name with `-algo.advantage.estimator`; its single call site sits in the visible training loop, its effect appears in the same step's logged reward and loss statistics, and a unit test sits beside the existing estimators. Four artifacts, function, flag, metric, test, and no layer crossed. Inserting a filtering stage in the experience pipeline has the same shape: a function over the batch at a visible point in the loop, plus a flag. This is the property Principle 1 buys (Sec. 2), and it is what an AI coding assistant needs to make the same edit unassisted: the path from flag to executed branch to tensor to metric to test never leaves code a human can read in one pass.

4. Evaluation: Does Leanness Cost Throughput?

Sec. 3 realized the principles of Sec. 2 as a small composed system; the evaluation now tests the parity constraint of Principle 3, that leanness costs no throughput. We answer three questions: (1) how large is the framework-owned RL surface, (2) do the engines' serving and memory optimizations keep arriving as configuration through the composed path, and (3) does the lean design sustain throughput comparable to a state-of-the-art Megatron-based stack under a matched protocol?

4.1. Footprint and Workflow: Three Steps, 8.6K Lines

A new experiment is three steps, author, launch, observe. *Author*: subclass `Env` or `ChatAgent` in one Python file and return `Result(reward=...)`; the `ChatAgent` listing in Sec. 3.2 is a complete trainable agent that points a stock OpenAI SDK at `ctx.base_url`, with no environment DSL or registry. *Launch*: one CLI command names the model, the prompt dataset, and the agent module; the shipped single-node and Slurm recipes use the same CLI and agent modules (Sec. A.2). *Observe*: every optimizer step logs reward statistics (`rollout/reward_mean`, per-group standard deviations), response-length distributions, evaluation pass@n, and a per-stage timing breakdown (`timing/generation`, `timing/policy_train`, `timing/broadcast`, `timing/step_total`) to Weights & Biases and TensorBoard through one logger configuration. Under the import-graph counting method of Tab. 1, the complete RL path behind this workflow is approximately 8.6K Python lines, against approximately 62K for verl and 25K for slime.

4.2. Throughput and Memory: Engine Features Arrive as Flags

Because the engines are composed rather than forked, their optimizations should reach Molt the day upstream ships them. The measurements below, taken on the shipped Qwen3.6-35B-A3B recipe, a multimodal MoE policy on a 32K multi-turn tool-use task across 2 nodes (8 training + 8 rollout GPUs), probe three points where a lean stack could pay a cost: multi-turn re-prefill, generation, and actor memory.

Setting	MOLT	slime
Model / precision	Qwen3-30B-A3B, bf16	
Hardware / topology	2 nodes × 8 H100; fully asynchronous, disaggregated: 8 training + 8 rollout GPUs	
Asynchrony	streaming asynchronous loop	one-step asynchronous (`train_async.py`)
Dataset	DAPO-Math prompts, deduplicated to 2K rows	
Batch	32 prompts × 4 samples = 128 sequences/step (global batch 128)	
Context / response	16,384-token context; 8,192-token response cap	
Sampling	temperature 1.0; top-p 1.0; top-k -1 (disabled); pinned identically	
Optimizer	Adam, identical hyperparameters; CPU offload on both	
Reference model	none on either side (KL coefficient 0; slime without `-use-kl-loss`)	
Recompute / micro-batch	full recompute; micro-batch 1; packing and dynamic batching disabled	
Training parallelism	DP8 (FSDP2), EP8, TP1	TP4+SP, CP1, EP8
Rollout engine	vLLM	SGLang
Rollout configuration	1 engine × TP8; prefix caching; CUDA graphs; 128 sequences in flight	
Commit	`cb2cae11`	`5d7296a7`; Megatron `1dcf0daf`

Table 2: **Head-to-head benchmark protocol.** Shared settings are pinned identically; the differing rows are the training backend with its parallelism layout and the rollout engine. Each stack runs its recommended training layout for this model and context length: MOLT its native FSDP2 data-parallel layout (context parallelism targets longer contexts), slime its native Megatron-Core 30B recipe. Every configuration is measured over three independent runs.

Prefix caching: 0.05 s re-prefill. With automatic prefix caching and session-consistent routing (Sec. 3.4), measured re-prefill of a growing conversation was 0.05 s on a cache hit. Absent a cache-miss baseline, this demonstrates the working path rather than an end-to-end speedup.

Speculative decoding: $5\times$ generation. With the checkpoint's MTP head enabled (Sec. 3.5), per-step generation time decreased from 329 s to 64 s: a single configuration change moves the recipe from generation-bound to training-bound.

Optimizer CPU offload: 18.3 GB for 18%. At this configuration, `-fsdp.offload optimizer` reduced actor peak GPU memory from 64.7 GB to 46.4 GB, the difference between fitting and not fitting the 8-GPU training partition, while `policy_train` time increased from 213 s to 251 s ($+18\%$).

4.3. Head-to-Head: Parity with a Megatron-Based Stack

This experiment tests Principle 3: whether the lean design sustains throughput comparable to a state-of-the-art Megatron-based stack, MOLT (AutoModel + vLLM) against slime (Megatron-Core + SGLang (Zheng et al., 2023)) on Qwen3-30B-A3B. The protocol (Tab. 2) pins every setting the two stacks can share and allows each backend its recommended parallel layout. Both stacks run fully asynchronously, with rollout and training disaggregated onto 8 + 8 GPUs: MOLT through its streaming asynchronous loop, slime through its one-step-asynchronous mode (`train_async.py`); neither side runs synchronously. Slime runs without `-use-kl-loss`, so neither stack loads a reference model and the per-step algorithmic work is identical. Each stack uses its own recommended training layout for this model and context length: slime its native 30B recipe (TP4+SP, CP1, EP8), MOLT its native FSDP2 data-parallel layout (TP1, EP8, DP8), context parallelism targets much longer contexts and is counterproductive at 16K. Micro-batch size is fixed to one with packing and dynamic batching disabled; both stacks also ship packing or token-based batching paths, but these change the parallel layout and are out of scope for a matched-backend comparison. The residual asymmetries are enumerated in the fairness notes below.

Result. The answer is affirmative: the two stacks are statistically comparable at 119.4 ± 2.3 versus 109.5 ± 10.3 s per optimizer step (mean $\pm$ s.d. over three runs). The slime cross-run spread (102–121 s) overlaps MOLT's band, so we claim no superiority in either direction; the mean difference of roughly 9% is within cross-run variability. Both stacks overlap generation with training on disaggregated GPUs, so the end-to-end step time is

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

Table 3: **Throughput parity under the matched protocol** (Tab. 2). Step: mean $\pm$ standard deviation of steady-state wall time per optimizer step over three independent runs per configuration (per-run means over optimizer steps 2–10; prompt order is identical across runs, sampled responses vary). Neither stack loads a reference model. Tok/GPU/s: generated tokens per GPU-second on the common workload basis of approximately 880K tokens per step, i.e., 880K / mean step time / 16 GPUs.

the meaningful unit of comparison, and we report step-level statistics only. At longer output lengths the residual difference should shrink dramatically: as trajectories grow toward 32K–128K reasoning and agent regimes, the optimizer step becomes dominated by generation, which both stacks delegate to a serving engine, while the share attributable to the training backend, the axis this comparison is about, shrinks toward irrelevance; the controlled 16K setting is, if anything, the regime least favorable to washing out backend differences. Training-side layout remains a first-order factor: forcing a context-parallel degree tuned for 32K contexts onto this 16K workload inflates Molt's step time by roughly 30%.

Fairness notes. Beyond the pinned protocol, the residual asymmetries are the training layout and rollout engine (Tab. 2) and the asynchrony styles described above (streaming versus one-step overlap). Neither stack performs a reference forward, and both run fully asynchronously. Molt's queue admits at most one policy-version lag, with per-token importance correction enabled (Liu et al., 2025a). Independently, the benchmark checkpoint exposes an upstream distributed-MoE forward mismatch that makes every row throughput-only: on this routing-sensitive 128-expert checkpoint, actor log-probabilities differ from an independent reference forward by approximately one nat, and the $[0.99, 1.01]$ sequence gate (Liu et al., 2025a) rejects the batch, so the reported step times measure throughput without an effective policy update. The 35B workload shows no such gap and the gate filters no sequences; convergence-parity validation awaits the upstream correction.

5. Related Work

With the design and its measured performance established, we position Molt against three families: RL training frameworks, asynchronous rollout systems, and agent-trajectory capture systems. Its contribution relative to each is the combination shown above, a readability-first research surface, a token-first agent contract, and a high-performance PyTorch-native implementation.

RL training frameworks. HybridFlow/verl (Sheng et al., 2025b) introduced a hybrid programming model that coordinates distributed RL roles while supporting multiple trainers and rollout engines. OpenRLHF (Hu et al., 2024) established the Ray+vLLM+ZeRO decomposition from which Molt began as a fork. NeMo-RL provides broad DTensor and Megatron-based post-training support, while TRL targets a lighter PyTorch fine-tuning regime. Slime pairs a deliberately thin orchestration layer with Megatron-Core and SGLang (Zheng et al., 2023). The field splits along a scale–complexity tradeoff. At one end, verl, NeMo-RL, and slime reach ultra-large-scale training through broad multi-backend surfaces or a Megatron-Core commitment: capable but heavy. At the other, OpenRLHF and TRL stay light and approachable but do not target agentic RL at frontier scale. Molt refuses the tradeoff, a Swiss-army-knife point in the design space: one lean, PyTorch-native codebase, several times smaller than the Megatron-based stacks (Tab. 1), that nonetheless spans a quick single-node experiment to a 700B MoE on standard engines, at throughput on par in a matched comparison (Sec. 4.3).

Asynchronous and disaggregated RL systems. AReaL (Fu et al., 2025) studies fully asynchronous, staleness-aware RL. StreamRL (Zhong et al., 2025) separates pipeline bubbles from long-tail "skewness bubbles" and addresses them with stream generation and skew-aware scheduling. Laminar (Sheng et al., 2025a) decouples trajectory generation and weight distribution through relay workers, while DORA (Hu et al., 2026) maintains

multiple policy versions to preserve trajectory consistency under streaming rollout. RolloutPipe (Chen et al., 2026) instead pipelines complete prompt groups under fixed weights to retain on-policy semantics. Relax (Zhang et al., 2026b) exposes service-level decoupling and a staleness parameter for omni-modal training. For agentic workloads, RollArt (Gao et al., 2025) maps prefill, decode, environments, and rewards to heterogeneous resources and schedules at trajectory granularity. MOLT shares disaggregation and completion-driven streaming but does not claim these scheduler contributions: it uses a bounded persistent prompt-group pool, generation-time log-probabilities, and per-token correction because those mechanisms fit in the readable core. Multi-version serving, skew predictors, elastic environment services, and optimal resource planning remain outside its scope.

Training–inference consistency. Agentic RL correctness depends on more than policy staleness. R3 (Ma et al., 2025) identifies divergent expert choices between rollout and training as a source of MoE instability and replays the rollout routes during optimization. MOLT implements this mechanism through vLLM's native route capture and AutoModel `RouterReplay`, and complements it with token-level behavior probabilities and fail-fast checks for unsupported feature combinations (Sec. 3.5).

Agent trajectories and harness integration. Agent Lightning (Luo et al., 2025) decouples existing agent execution from RL training through an agent data interface and hierarchical credit assignment. POLAR (Xu et al., 2026), which supersedes PRORL AGENT (Zhang et al., 2026a), treats an arbitrary harness as a black box: a provider-compatible proxy records model traffic and reconstructs token-faithful trajectories. A recent systems perspective identifies three missing layers for self-evolving deployed agents: a standardized step-granular trajectory protocol, enterprise data conversion, and a unified control plane for policy updates (Yan et al., 2026). MOLT addresses the first layer inside a research trainer: its loopback server makes stock SDK calls token-exact and segments trajectories under prefix-rewriting compaction. It does not provide the enterprise data-conversion layer or control plane, and can consume harness-side trajectories as a complementary source.

RL objectives. MOLT intentionally proposes no new objective. Its estimators follow REINFORCE++ (Hu et al., 2025), GRPO (Shao et al., 2024), DAPO (Yu et al., 2025), and GSPO (Zheng et al., 2025). The systems contribution is to keep the quantities these objectives assume, aligned tokens, behavior log-probabilities, complete prompt groups, and consistent sparse routing, explicit in one data path.

6. Future Work

MOLT already runs the full asynchronous loop, rollout, weight refit, and optimizer step, end to end on a 700B MoE at expert parallelism 256, on the same lean loop it runs at 4B. The configuration surface expresses DeepSeek-V3-class models on components validated at frontier scale upstream, vLLM serves the largest open-weight MoE, and AutoModel ships EP-sharded DeepSeek-V3-class recipes, so what remains toward and beyond the 3 trillion-parameter on NVIDIA GB300 mark is end-to-end convergence measurement, not redesign. Together with NeMo AutoModel, we will keep pushing the model size MOLT trains end to end.

7. Conclusion

MOLT is designed for the research loop: environments remain ordinary programs, the complete RL path remains small enough to inspect, and model scale does not require leaving PyTorch. Human readability is the primary code-quality criterion, and AI coding assistants must be able to trace the same explicit path from request tokens through rollout, reward, weight version, and loss. Stable upstream components then supply serving and distributed execution without expanding the framework-owned surface.

The measurements are consistent with these objectives: engine optimizations and optimizer offload arrive as configuration through the composed path, and system performance is statistically comparable to a state-of-the-art Megatron-based stack under a matched protocol. Readable infrastructure pays a second dividend: when something anywhere in the stack is wrong, the failure is visible, localizable, and difficult to ignore. Quality and

usability studies are natural next steps; MOLT provides a concrete, high-performance substrate for them. More broadly, MOLT is infrastructure designed from the start for the era in which research happens with AI coding assistants in the loop: a codebase sized to be read whole, one visible loop, parts shaped like the algorithm, a form we expect more research infrastructure to take.

Availability. MOLT is open source under the Apache-2.0 license. The repository ships the complete framework, the reference agents and one-command recipes behind every reported measurement, and prebuilt containers with the full training and serving stack (Sec. A.2).

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. *arXiv preprint arXiv:2402.14740*, 2024. 7

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016. 5

Rongjian Chen, Jianmin Hu, Kejiang Ye, and Minxian Xu. RolloutPipe: Overlapping pipelined rollout and training in disaggregated on-policy LLM reinforcement learning. *arXiv preprint arXiv:2606.26997*, 2026. 11

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. 7

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. 10

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. 11

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. 2

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. 10

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. 7, 11

Tianhao Hu, Xiangcheng Liu, Youshao Xiao, Yang Zheng, Xuan Huang, Jinrui Ding, et al. DORA: A scalable asynchronous reinforcement learning system for language model training. *arXiv preprint arXiv:2604.26256*, 2026. 10

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. 2, 4

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. 4, 5, 7, 10

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. 7

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. 2, 5, 11

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. 7, 11

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. 2, 4

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. 7

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 7

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. 2, 7, 11

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. 10

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)*, 2025b. 10

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Jing Hua Toh, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024. 2, 6

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. 2, 5, 11

Ran Yan, Wei Fu, Jiale Li, Shusheng Xu, Zhiyu Mei, Jiaxuan Gao, et al. Next-generation agentic reinforcement learning systems enable self-evolving agents. *arXiv preprint arXiv:2607.01120*, 2026. 2, 11

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, 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. 7, 11

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. 11

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. 11

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. 4

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025. 11

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. *arXiv preprint arXiv:2312.07104*, 2023. 9, 10

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. 10

A. Appendix

A.1. Scaling Knobs

Tab. 4 lists MOLT's principal scaling controls; one shared launch path spans a dense 4B model and a configured 1T-class MoE.

Side	Flag	Purpose
Actor	`-fsdp.tp_size`	tensor parallelism (AutoModel-native)
	`-fsdp.ep_size`	expert parallelism; 256 expresses DeepSeek-V3-class MoE
	`-fsdp.cp_size`	context parallelism for 32K+ sequences
	`-fsdp.offload optimizer`	Adam states in host memory; checkpoint-compatible
vLLM rollout	`-vllm.tensor_parallel_size`	per-engine tensor parallelism
	`-vllm.enable_expert_parallel`	per-engine expert parallelism (EP = TP × DP)
	`-vllm.data_parallel_size`	rollout data parallelism; raises EP past TP (DeepSeek-V3-style)
	`-vllm.max_num_batched_tokens`	scheduler token budget
	`-vllm.mtp_num_speculative_tokens`	speculative decoding; rollout-only; 0 disables
MoE stability	`-train.routing_replay`	rollout routing replay; default in the MoE recipes
	`-actor.freeze_moe_router`	router freeze; coarser alternative to replay

Table 4: **MOLT scaling knobs.** Actor-side parallelism composes with FSDP2 sharding; the vLLM side mirrors it per engine.

A.2. Reproduction Note

The single-framework measurements come from the shipped Qwen3.6-35B-A3B geo3k RL recipe (2 nodes × 8 H100, 8 training + 8 rollout GPUs; 32K context; CP8 / EP8 / TP1; 4 prompts × 4 samples; temperature 1.0, `seq-mask-tis`, R3 on). The head-to-head benchmark follows Tab. 2, which pins the framework commits. Each reported configuration ships as a single-command launch recipe, with container specifications, in the repository.