PRIME-RL: Async & Decentralized RL Training at Scale

Mika Senghaas[1] **Fares Obeid**[1] **Sami Jaghouar**[1] **William Brown**[1]

Jack Min Ong[1] **Andrew Baker**[1] **Justus Mattern**[1] **Daniel Auras**[12]

Jannik Straube[1] **Manveer Basra**[1] **Aiman Ismail**[1] **Johannes Hagemann**[1]

Abstract

We present prime-rl, an open-source framework for large-scale reinforcement learning (RL). `prime-rl` is designed to scale seamlessly from a single node to thousands of GPUs, making it suitable for tinkering, research, and production-scale training. Tailored for agentic RL, it offers first-class support for multi-turn interactions and tool use through its asynchronous architecture. Environments are constructed using the `verifiers` library and integrated with the Environments Hub, enabling environment development to remain fully decoupled from the training infrastructure. To demonstrate its capabilities, we train `DeepSeek-R1-Distill-Qwen-32B` on chain-of-thought (CoT) math reasoning using 24 NVIDIA H200 GPUs. We measure up to 30K tokens per second in aggregated throughput and reach a peak MFU of 38.46%.

1 Introduction

Scaling compute for training large language models (LLMs) with reinforcement learning with verifiable rewards (RLVR) has emerged as the dominant paradigm for improving model performance in post-training. Models such as OpenAI o3 [13], Grok 4 [22], and DeepSeek R1 [3] demonstrate that training models via RL for long-context reasoning and agentic tool use greatly enhances their capabilities, making them more effective both for everyday and specialized tasks.

However, existing open-source frameworks are often complex, monolithic, and designed without modularity in mind [16]. This can make extensibility difficult, inhibit broad adoption, slow down individual research projects, and lead to a fragmentation of ecosystem artifacts. In addition, no framework is designed for the unique requirements set of decentralized RL, including support for heterogeneous, dynamically-scaling or permissionless compute.

In this report, we present `prime-rl`, a framework for large-scale reinforcement learning, which powers our internal post-training pipelines and public permissionless runs. `prime-rl` is easy-to-use and hackable, yet performant and scalable enough to facilitate state-of-the-art RL post-training. We highlight the following features:

1. First-class support for OpenAI-compatible async inference, `verifiers` environments [2], and a public Environments Hub to standardize agentic RL training and evaluation
2. Support for end-to-end post-training, including SFT and multi-turn agentic RL

[1]Prime Intellect, Inc. Correspondence to: `johannes@primeintellect.ai`
[2]Partially while at `ellamind`

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

3. Multi-node deployment with FSDP2 training and vLLM inference backend
4. Naturally asynchronous training for high-throughput performance in decentralized settings
5. Modular and extensible by nature, enabling high research velocity

2 Design & Architecture

2.1 Architecture

Three main abstractions facilitate RL training: the *orchestrator*, the *trainer*, and the *inference* service.

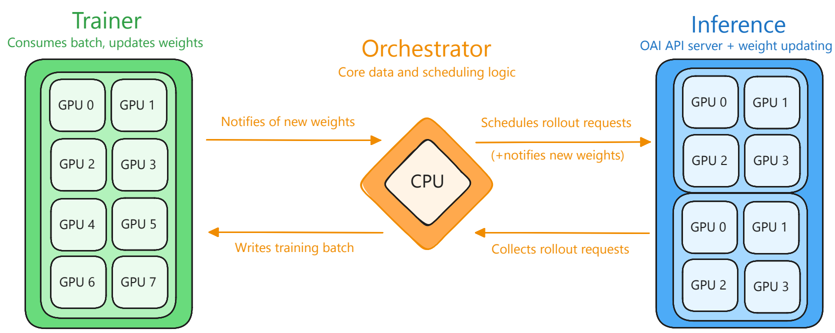


Figure 1: **Architecture.** A RL training run involves the coordination of a trainer, orchestrator and an inference service. The FSDP trainer and vLLM inference run disaggregated, and can be individually deployed across multiple nodes.

Orchestrator. The orchestrator is a lightweight CPU process that handles the core data and scheduling logic, serving as an intermediary between the trainer and inference service with bidirectional relays. In one direction, it collects rollouts from the inference server, assembles them into packed batches, and dispatches them to the trainer; in the other direction, it relays updated model weights from the trainer to the inference service. The orchestrator utilizes `verifiers` environments to abstract multi-turn rollout generation and scoring, leveraging async OpenAI-compatible inference clients.

Trainer. The trainer is responsible for producing an updated policy model given rollouts and advantages. We use FSDP 2 [1] as the backend with compatibility for any HuggingFace (HF) model. FSDP shards model parameters, gradients, and optimizer states, allowing training large models with data parallelism and minimal GPU memory footprint. The trainer is inspired by `torchtitan` [10] and relies on native PyTorch features to implement advanced parallelism techniques, such as tensor, context or expert parallelism.

Inference Service. The inference service in its simplest form is a standard OpenAI-compatible server with a vLLM [8] backend. The API specification is extended with three custom endpoints to enable updating the server with the latest policy: `/init_broadcaster` is used to initialize a NCCL process group if the NCCL weight broadcast backend is enabled, `/update_weights` is used to update the policy, and `/reload_weights` is used to reset the weights to the base model in between experiments. Otherwise, we rely on vLLM's optimized kernels, parallelism strategies, and scheduling for fast rollout generation. Given the disaggregated nature of the service architecture, it can be directly extended to include multiple engines with a shared request pool, allowing operation across multiple clusters and straightforward integration of alternative inference engines (e.g. SGLang [28], Tokasaurus [7].

2.2 Data Flow

Here we describe the core data flow within a single training step for `prime-rl`. At the beginning of a step, the orchestrator checks whether the inference service policy model should be updated with the latest training checkpoint. If so, it sends a request `/update_weights` to trigger replacing vLLM tensors in-place throughout the inference service. The orchestrator then samples prompts from the data buffer, an abstraction used to define dynamic data sampling strategies, e.g. online difficulty filtering [27] or difficulty pools [23]. Sampled prompts are sent to the `verifiers` environment, which asynchronously schedules rollout generation and scoring. The `verifiers` environment returns rollout results, including completions, vLLM logprobs, masks, and rewards, according to the spec of the environment. Completed rollouts are then added to the data buffer; orchestrator scheduling of rollouts continues until a sufficiently large batch is ready to be consumed by the trainer, e.g. as determined by an online difficulty filtering strategy. The orchestrator then shards the batch of rollouts across DP ranks, collates them into training-ready tensors, and dispatches them to each trainer. Each FSDP rank consumes the local training batch and processes micro-batches while accumulating a synchronized gradient. Upon completion of a global batch, the updated policy model is written as a weight checkpoint to disk, from where it can be loaded by the inference service in future steps. For asynchronous training, this entire process occurs in multiple parallel channels, staggered by appropriate offsets (see Section 3.4 for more details). Checkpoints persist on disk only as long as necessary for the asynchronicity level.

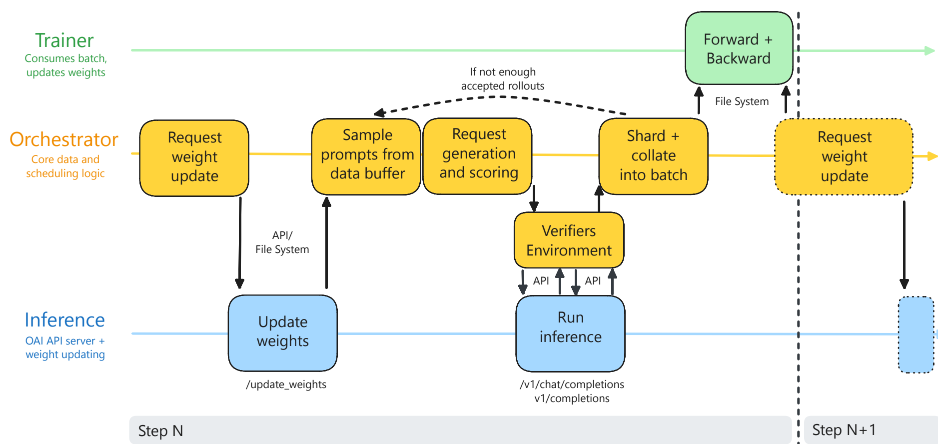


Figure 2: **Data Flow.** The data flow between the trainer, orchestrator and inference module during a single training step. We show actions for each module, and their sequential dependencies from left-to-right. For simplicity, we show a fully synchronous flow and hide non-essential logic.

3 Features

3.1 Verifiers & Environments Hub Integration

`prime-rl` has first-class support for RL environments developed with the `verifiers` library and installed as standalone Python modules via the Environments Hub[1]. This decouples the development of RL environments from training abstractions, allowing for quicker development and portability, e.g. the same environment can be used with `prime-rl`, `verifiers`, `trl` [19], or any other trainer which adopts support for `verifiers` environments.

An environment is a lightweight abstraction that encapsulates multi-turn rollout logic with native tool calling support (search, code execution) or other external system interactions (games, user simulators), along with dataset preprocessing and reward computation. Conceptually, `verifiers` environments for RL training play the same role as datasets for SFT or pre-training; disentangling environments from training infrastructure yields desirable compositionality and interoperability, as multiple environments can be straightforwardly grouped into a single "mixture" environment, and support both online and offline rollout generation (e.g. for use as evaluations). Environments manage submission of inference requests and state information over the lifetime of a rollout, and return completed rollouts to the orchestrator. A reward manager abstraction ("Rubric" in `verifiers`) lives within the environment and provides configurable control and resources to support a broad range of reward computation strategies, such as compound reward functions, global state references, LLM judges, caching of expensive computations, and customizable parallelism strategies.

[1]URL omitted for double-blind review

Environments are built in isolation of training logic and can easily be tested against local or API models. Once ready for training, they are pushed to the Environments Hub and immediately available to the `prime-rl` trainer as installable Python modules. By adopting the `verifiers` spec, training and evaluating inside of an environment works directly upon installation without any code modification in `prime-rl`.

3.2 End-to-End Post-Training

Modern post-training typically combines supervised fine-tuning (SFT) and reinforcement learning (RL) [17, 24]. To support this, our framework provides a unified interface for both methods. The SFT and RL trainers share core modeling components, so any model usable for RL can also serve as an SFT warm-up, and vice versa. This tight integration streamlines the overall post-training workflow.

3.3 Rayless Multi-Node Training

A key requirement of `prime-rl` is seamless scalability. It should be frictionless to take a research idea developed on a single node to production-scale training running on a decentralized cluster with hundreds of nodes. This motivates the key design choice to use FSDP as the training and vLLM as the inference backend, and fully decouple those components. Crucially, it removes the need for custom hardware orchestration logic as both have built-in support for multi-node deployments.

3.4 Decentralized Permissionless Training

Using `prime-rl` for globally distributed training with permissionless inference workers does not require any changes to the trainer. The only difference is that the orchestrator does not communicate with an inference server directly but will instead request rollout completions from an intermediate scheduler component. The scheduler is responsible for load balancing the incoming rollout requests into the permissionless inference pool and guarantee that rollout responses have been verified by TOPLOC [12]. To this end, each worker in the inference pool will serve an extended OpenAI-compatible spec with additional routes for generating and validating TOPLOC proofs. Each request is first routed to an inference worker for generation. The generation response, including the associated TOPLOC proof, is then routed to another worker in the pool for validation.

3.5 Asynchronous Off-Policy Training

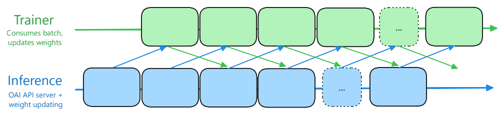
In decentralized settings asynchronous execution of trainer and inference becomes a necessity to avoid long blocking time. For this reason, `prime-rl`'s trainer and inference run disaggregated, i.e. on different sets of (possibly non-colocated) GPUs.

At each step, all artifacts are identified by the step count n. For the trainer, this is the gradients g_n and model weights θ_n, and for the inference service, rollouts (x_n, y_n). At step 0, the inference service uses θ_0 (base model) to produce (x_0, y_0). The trainer subsequently uses (x_0, y_0) to compute g_0 to finally update the model as $\theta_1 \leftarrow \theta_0 - g_0$.

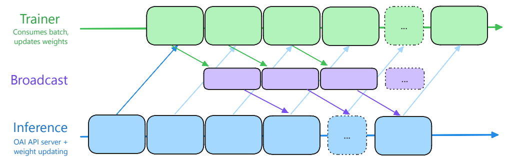
In synchronous on-policy training, the inference engine stalls after producing (x_0, y_0) because it requires θ_1 to produce the next rollouts (x_1, y_1). To prevent this, we allow off-policy training, which means that the inference service can asynchronously generate rollouts from an old policy model up to some `async_level`. For example, if `async_level=1`, the inference service continues generating (x_1, y_1) from θ_0, while the trainer is producing θ_1 in parallel. An example of such overlapped, asynchronous computation is shown in Fig 3(a). More generally, for any `async_level`, the inference service produces rollouts from $\theta_{min(0,n-\texttt{async\_level})}$. In decentralized settings we often need `async_level` ≥ 2 to fully hide the communication bottleneck from broadcasting the updated model weights, as illustrated in Figure 3(b).

3.6 Mixture-of-Experts Support

We support the Mixture-of-Experts (MoE) [15] layer implementation from `torchtitan` [10]. This implementation leverages a grouped matrix multiplication kernel for expert execution and provides support for expert parallelism (EP). To monitor load distribution, we compute and log the maximum violation load-balancing metric $\mathrm{MaxViolation} = \frac{\max_i \mathrm{Load}_i - \overline{\mathrm{Load}_i}}{\overline{\mathrm{Load}_i}}$ as described in [20].



(a) One-Step Off-Policy



(b) Two-Step Off-Policy

Figure 3: **Asynchronous Off-Policy Training.** We show the execution graph of one-step (left) and two-step (right) off-policy training. At step n, the inference engine uses $\theta_{\min(0, n-\texttt{async\_level})}$. In colocated settings, one-step off-policy allows to fully overlap training and inference. In decentralized settings, weight broadcasting may become a bottleneck, requiring higher levels of asynchrony.

3.7 Evals

Evaluation and training are tightly coupled through the use of `verifiers`, streamlining the process of evaluating against a wide range of common benchmarks, including AIME [5, 14], SWE-Bench [6], TerminalBench [18] or τ-bench [26]. Evaluation can be run both as part of an active training (online) or as a standalone entrypoint (offline). When evaluating online, the orchestrator asynchronously interleaves eval requests with training requests, effectively hiding any overhead, while providing useful real-time feedback of the training performance.

4 Training Algorithm

We adopt a token-level [27] loss variant of the AIPO training objective introduced in Llama-RL [21], and omit the entropy and KL loss terms. At each step, we sample N prompts from our dataset. For each prompt x, we sample a group of rollouts $\{y_i\}_i^G$ and use a verifier to assign scores s_i to each y_i. Then, the optimization objective is given by

$$\mathcal{J}_{\text{AIPO}}(\theta) = \frac{1}{\sum_{j=1}^{N}\sum_{i=1}^{G}|y_i^{(j)}|}\sum_{j=1}^{N}\sum_{i=1}^{G}\sum_{t=1}^{|y_i^{(j)}|}\min\left(\frac{\pi(y_{i,t}^{(j)} \mid x_j, y_{i,<t}^{(j)})}{\mu(y_{i,t}^{(j)} \mid x_j, y_{i,<t}^{(j)})}, \delta\right)\hat{A}_{i,t}^{(j)} \tag{1}$$

where μ refers to the policy that generated the rollout and π refers to the current policy. The token-level advantage is estimated as $\hat{A}_{i,t} = S_i - \text{mean}(\{S_i\}_i^G)$O [11] and we use $\delta = 8$.

We found it to be critical to address the following inference-trainer mismatch: Even when π and μ share the same parameters θ, they can produce significantly different token probabilities, leading to unexpected distribution shifts that can cause runs to crash multiple days into the experiments. In our setup, we found it more stable to not recompute logprobs using our training backend, but instead rely on the logprobs from vLLM directly to estimate $\mu(y_{i,t}^{(j)} \mid x_j, y_{i,<t}^{(j)})$. In this setup, this is directly equivalent to including an importance-sampling correction between the trainer and inference logprobs. For a more thorough investigation we point the reader to [25].

5 Experiments

5.1 Experiment Setup

To showcase `prime-rl`, we train `DeepSeek-R1-Distill-Qwen-32B` across 24 H200 GPUs on math reasoning tasks with `prime-rl`. We detail the training setup with complete steps to reproduce and report on training dynamics and efficiency.

Environment. We train on `skywork-math`[2], a single-turn environment with challenging math problems sourced from Skywork OR1's training data [4]. For verification we use `verifiers` native think parser and boxed-answer extraction and `math-verify` [9] for final reward assignment. We filter the data prior to training to exclude overly easy and hard problems.

[2] URL omitted for double-blind review

RL Setup. At each step, we sample 128 prompts and generate 16 rollouts per prompt at temperature 0.7 and 16k maximum context length. We compute gradients with respect to the training objective given in Equation **??**. We perform 160 training steps with a constant learning rate of 1×10^{-6}.

Hardware Setup. We deploy the run across 24 NVIDIA H200 GPUs. We use a full node of 8 GPUs for the trainer. All remaining 16 GPUs are used for inference with data parallel size 4 and tensor parallel size 4 across two nodes. Because the trainer and inference were not distributed around the world and had decent direct Ethernet based connection, we only require one-step off-policy to overlap computation and prevent GPU idle time.

5.2 Experiment Results

Training Efficiency. As shown in Figure 4, the system maintained high throughput throughout training. On the trainer, we achieved a throughput of 11.3K (±1K) tokens per second, while inference reached 14.4K (±1.3K) tokens per second, highlighting a mild throughput imbalance. Nonetheless, compute utilization remained high, reaching a peak Model FLOPs Utilization (MFU) of 38.46% on the trainer. The experiment ran for 64 hours in total, consuming 1,536 GPU hours, with each training step averaging 22.9 (±3.4) minutes.

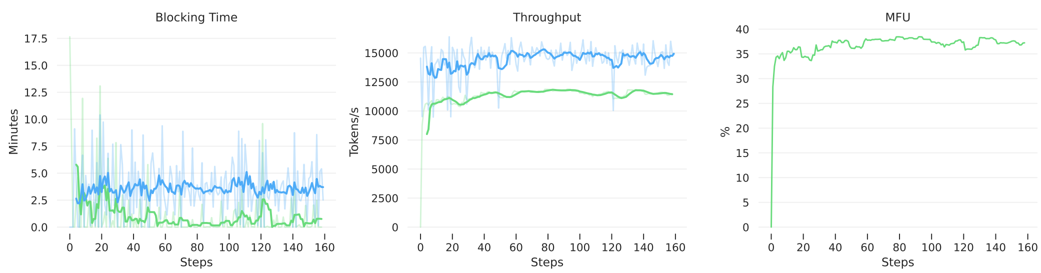


Figure 4: **Training Efficiency.** We sustain high throughput with little idle time and achieve a peak MFU of 38.46% on the trainer in our experiment setup.

Training Dynamics. Throughout training, the run exhibited stable and healthy learning behavior, as visible in Figure 5. The gradient norm remains consistent, without signs of vanishing or exploding gradients. Similarly, the entropy across tokens shows no significant decline, suggesting that the policy retains exploration and does not prematurely converge. Meanwhile, the reward trends upward, indicating progressive improvement in performance. Together, these trends reflect a well-behaved optimization process and steady policy refinement.

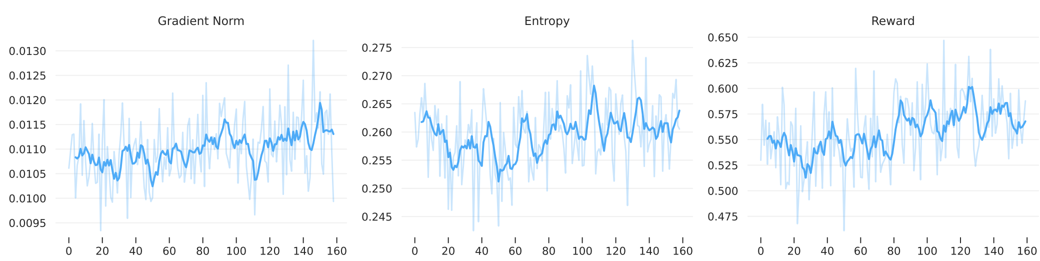


Figure 5: **Training Dynamics.** We report key indicators of training stability, including the gradient norm, entropy and mean reward. The gradient norm and entropy are stable and non-increasing, while the reward is going up, indicating healthy learning behavior.

6 Summary

In this report, we introduced `prime-rl`, our open-source framework for large-scale reinforcement learning (RL). It is a feature-packed and battle-tested RL trainer designed for the age of agentic RL training and decentralized compute. Its architecture disaggregates training and inference, enabling efficient asynchronous execution. Key features include native integration with `verifiers` environments and the Environments Hub, end-to-end post-training (SFT + RL), and native MoE support. We demonstrate the efficacy and robustness of the framework by training a 32B model on math reasoning tasks, sustaining high throughput and stable optimization.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[2] William Brown. Verifiers: Reinforcement Learning with LLMs in Verifiable Environments. `https://github.com/willccbb/verifiers`, 2025. Commit abcdefg • accessed DD Mon YYYY.

[3] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, 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, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, 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, 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, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, 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, Yanhong Xu, 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.

[4] Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork Open Reasoner 1 Technical Report, 2025.

[5] Hugging Face H4. AIME 2024, 2024. 30 problems from AIME I & II (2024).

[6] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[7] Jordan Juravsky, Ayush Chakravarthy, Ryan Ehrlich, Sabri Eyuboglu, Bradley Brown, Joseph Shetaye, Christopher Ré, and Azalia Mirhoseini. Tokasaurus: An LLM Inference Engine for

High-Throughput Workloads. https://scalingintelligence.stanford.edu/blogs/tokasaurus/, 2025.

[8] 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*, 2023.

[9] Hynek Kydlíček. Math-Verify: Math Verification Library.

[10] 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.

[11] 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, 2025.

[12] Jack Min Ong, Matthew Di Ferrante, Aaron Pazdera, Ryan Garner, Sami Jaghouar, Manveer Basra, Max Ryabinin, and Johannes Hagemann. TOPLOC: A Locality Sensitive Hashing Scheme for Trustless Verifiable Inference, 2025.

[13] OpenAI. OpenAI o3 and o4-mini System Card. System card.

[14] OpenCompass. AIME 2025, 2025. 30 problems from AIME 2025-I & II.

[15] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*, 2017.

[16] 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. *arXiv preprint arXiv: 2409.19256*, 2024.

[17] 5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models, 2025.

[18] The Terminal-Bench Team. Terminal-Bench: A Benchmark for AI Agents in Terminal Environments, Apr 2025.

[19] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformer Reinforcement Learning. `https://github.com/huggingface/trl`, 2020.

[20] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-Loss-Free Load Balancing Strategy for Mixture-of-Experts, 2024.

[21] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A Distributed Asynchronous Reinforcement Learning Framework for Efficient Large-scale LLM Training, 2025.

[22] xAI. Grok 4. News announcement.

[23] LLM-Core Xiaomi, :, Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiji Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinzhe Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. MiMo: Unlocking the Reasoning Potential of Language Model – From Pretraining to Posttraining, 2025.

[24] 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.

[25] Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your Efficient RL Framework Secretly Brings You Off-Policy RL Training, August 2025.

[26] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, 2024.

[27] 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.

[28] 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, 2024.

A Reproducibility

The following commands where used to launch the experiments described in Section 5.

```
# Inference (Node 0)
uv run inference \
  --model.name deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
  --max-model-len 16384 \
  --max-seq-len-to-capture 16384 \
  --data-parallel-size 4 \
  --tensor-parallel-size 4 \
  --data-parallel-size-local 2 \
  --data-parallel-address 192.168.0.113 \
  --data-parallel-rpc-port 13345
```

```
# Inference (Node 1)
uv run inference \
  --worker-extension-cls
     rl_framework.inference.vllm.worker.CheckpointWorker \
  --model.name deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
  --max-model-len 16384 \
  --max-seq-len-to-capture 16384 \
  --data-parallel-size 4 \
  --tensor-parallel-size 4 \
  --data-parallel-size-local 2 \
  --data-parallel-address 192.168.0.113 \
  --data-parallel-rpc-port 13345 \
  --data-parallel-start-rank 2 \
  --headless
```

```
# Orchestrator
uv run orchestrator \
  --client.host 192.168.0.113 \
  --num-train-workers 8 \
  --model.name deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
  --environment.id skywork-math \
  --environment.args '{"solve_rate_field":␣
     "solve_rate_qwen_r1_distill_32b",␣"min_solve_rate":␣0.001,␣
     "max_solve_rate":␣0.999}' \
  --max-steps 500 \
  --log.level debug \
  --ckpt.interval 50 \
  --ckpt.keep 1 \
  --monitor.wandb.project prime-rl \
  --monitor.wandb.name r1-distill-qwen32b-orchestrator \
  --monitor.wandb.log-extras.interval 1 \
  --outputs-dir ~/shared/outputs \
  --batch-size 2048 \
  --micro-batch-size 1 \
  --seq-len 16384 \
  --rollouts-per-example 16 \
  --sampling.temperature 0.7 \
  --async-level 1
```

```
# Trainer
uv run torchrun --nproc-per-node 8
    src/rl_framework/trainer/rl/train.py \
  --model.name deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
  --model.compile \
  --model.ac \
  --optim.lr 1e-6 \
```

```
--max-steps 500 \
--log.level debug \
--ckpt.interval 50 \
--ckpt.keep 1 \
--weights.interval 10 \
--monitor.wandb.project prime-rl \
--monitor.wandb.name r1-distill-qwen32b-trainer \
--monitor.wandb.log-extras None \
--outputs-dir ~/shared/outputs \
--async-level 1
```

B RL Entrypoint

To streamline single-node RL experiments, `prime-rl` offers a simple entrypoint script. The script takes in configuration files and launches subprocesses for the trainer, orchestrator, and optionally the inference service. It manages hardware placement, validates shared configuration fields, and ensures proper cleanup in the event of a failure. For better observability, we also provide a `tmux` layout script that streams logs from each module into dedicated panes, offering a clean and organized view of the training process (Figure 6).

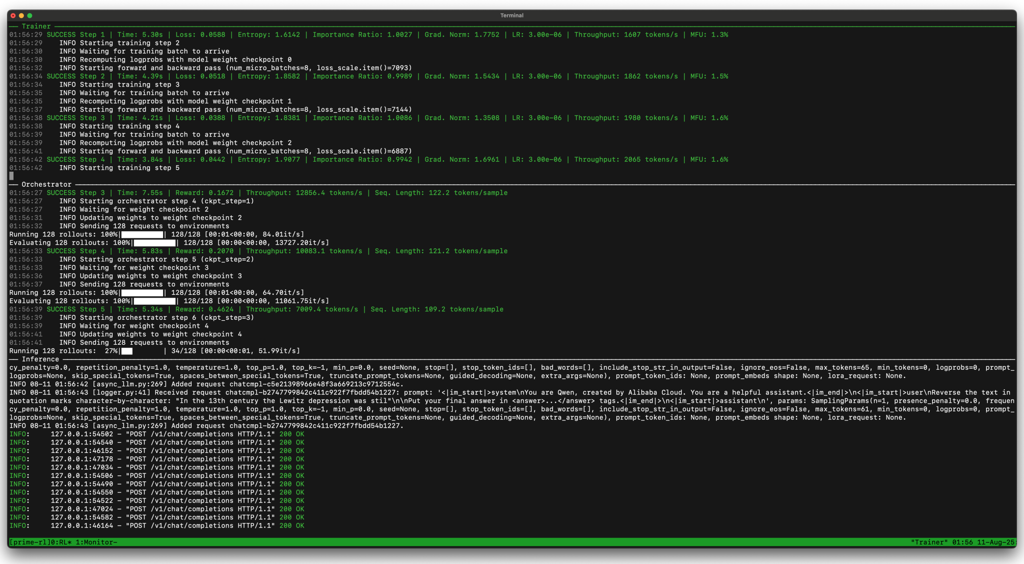


Figure 6: **Layout Script.** The helper script `tmux.sh` streams the logs of the trainer, orchestrator and inference service into three horizontal panes.