# INTELLECT-3: Technical Report

**Prime Intellect Team**

Mika Senghaas Fares Obeid Sami Jaghouar William Brown
Jack Min Ong Daniel Auras* Matej Sirovatka Jannik Straube
Andrew Baker Sebastian Müller Justus Mattern Manveer Basra
Aiman Ismail Dominik Scherm Cooper Miller Ameen Patel
Simon Kirsten Mario Sieg Christian Reetz Kemal Erdem
Vincent Weisser Johannes Hagemann†

## Abstract

We present INTELLECT-3, a 106B-parameter Mixture-of-Experts model (12B active) trained with large-scale reinforcement learning on our end-to-end RL infrastructure stack. INTELLECT-3 achieves state of the art performance for its size across math, code, science and reasoning benchmarks, outperforming many larger frontier models. We open-source the model together with the full infrastructure stack used to create it, including RL frameworks, complete recipe, and a wide collection of environments, built with the `verifiers` library, for training and evaluation from our Environments Hub community platform.
Built for this effort, we introduce `prime-rl`, an open framework for large-scale asynchronous reinforcement learning, which scales seamlessly from a single node to thousands of GPUs, and is tailored for agentic RL with first-class support for multi-turn interactions and tool use. Using this stack, we run both SFT and RL training on top of the GLM-4.5-Air-Base model, scaling RL training up to 512 H200s with high training efficiency.

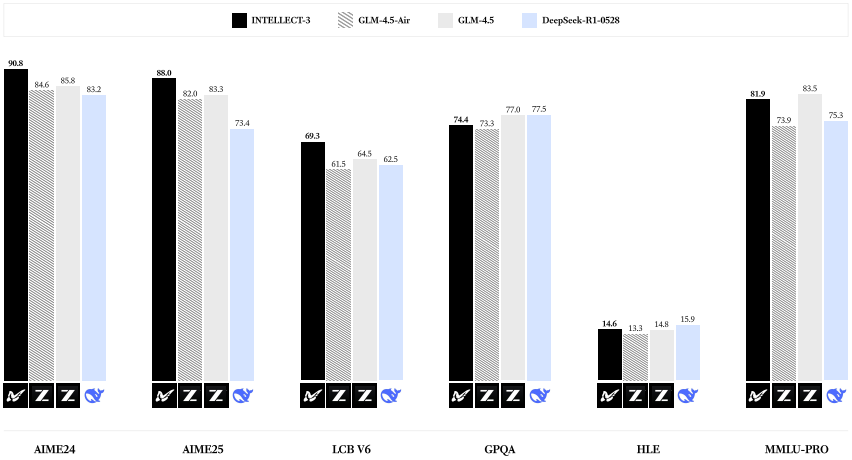



Figure 1: INTELLECT-3 Evaluation Results.[1]

*Partially while at `ellamind`

†Prime Intellect, Inc. Correspondence to: `johannes@primeintellect.ai`

## Contents



[1] All the models above were evaluated using our public, reproducible Environments Hub implementations. To ensure the best results, we use APIs provided directly by the model creators whenever available to avoid any performance loss from quantization or other inference optimizations.

# 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 [34], Grok 4 [50], and DeepSeek R1 [8] 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.

While the open-source ecosystem has been successful in producing strong open-weight models trained with RL, the open-source infrastructure for the end-to-end RL pipeline, including training frameworks, RL environments and evaluations, and stable training recipes, still has notable shortcomings compared to proprietary pipelines inside frontier AI labs. For example, existing open-source frameworks are often complex, monolithic, and designed without modularity in mind [43]. This can make extensibility difficult, inhibit broad adoption, slow down individual research projects, and lead to a fragmentation of ecosystem artifacts.

In this report we present INTELLECT-3, a state-of-the-art model in its weight class built on top of the GLM-4.5-Air base model [44]. Alongside the model, we share the complete training recipe, covering everything from supervised fine-tuning on the initial base checkpoint to large-scale reinforcement learning. We also introduce our training framework `prime-rl`, which is easy to use and hackable yet performant and scalable enough to support state-of-the-art RL post-training. We highlight the following features:

1. First-class support for OpenAI-compatible async inference, `verifiers` environments [5], 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
3. Multi-node deployment with FSDP2 training and vLLM inference backend
4. Naturally asynchronous training for high-throughput including continuous batching and in-flight weight updates [37]
5. Modular and extensible by nature, enabling high research velocity

INTELLECT-3 fully trained end-to-end with `prime-rl` and our open-source infrastructure components, outperforms existing open-source models in its size range across the board and even surpasses frontier open models over $6\times$ larger on reasoning and agentic benchmarks: It achieves scores of 90.8% and 88.0% on AIME 2024 and 2025 respectively, outperforming DeepSeek's frontier models and matching the performance of Z.ai's latest next-generation model GLM-4.6 which has over $3\times$ the number of parameters. On coding benchmarks, INTELLECT-3 achieves 69.3% on LiveCodeBench v6, outperforming Z.ai's GLM-4.5-Air post-train by 8%.

We open-source INTELLECT-3[2], our RL training framework `prime-rl`[3], and all environments [4] used for synthetic data generation, training, and evaluation.

The remainder of this report is organized as follows: Section 2 provides a detailed overview of the end-to-end reinforcement learning infrastructure, including `prime-rl`, `verifiers`, the Environments Hub, sandbox code execution and compute orchestration. Section 3 describes the concrete INTELLECT-3 training run, covering the RL environments used for training and the results of both the SFT and RL stages. Section 4 presents the model's evaluation results on reasoning and agentic benchmarks. Finally, Section 5 concludes the report and outlines directions for future work.

# 2 Training Infrastructure

We introduce the following key training infrastructure components for the end-to-end training of INTELLECT-3

- **prime-rl:** An asynchronous RL framework powering large-scale SFT and RL training of Mixture-of-Experts models.

[2] `huggingface.co/PrimeIntellect/INTELLECT-3`

[3] `github.com/PrimeIntellect-ai/prime-rl`

[4] `hub.primeintellect.ai`

- **Verifiers and the Environments Hub:** A unified environment interface and ecosystem for agentic RL.
- **Prime Sandboxes:** High-throughput, secure code execution for agentic coding environments.

### 2.1 prime-rl: A Framework for Asynchronous Reinforcement Learning at Scale

INTELLECT-3 was trained end-to-end with `prime-rl`, our production-scale post-training framework. `prime-rl` provides native integration with `verifiers` environments, which power our entire post-training stack from synthetic data generation, supervised fine-tuning, reinforcement learning, to evaluations. Through its tight connection to the Environments Hub, the entire training stack can seamlessly access a rapidly expanding ecosystem of training and eval environments.

#### 2.1.1 Architecture

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

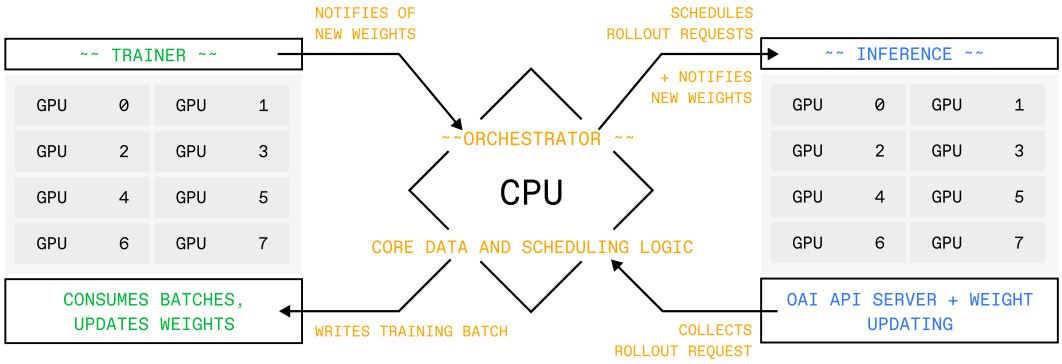



Figure 2: **Architecture.** A RL training run involves the coordination of a trainer, orchestrator and an inference service. The FSDP trainer and vLLM inference run disaggregated, and can be individually deployed across multiple nodes.

**Orchestrator.** The orchestrator is a lightweight CPU process that handles the core data flow 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, allowing any environment on the Environments Hub to plug into the training loop.

**Trainer.** The trainer is responsible for producing an updated policy model given rollouts and advantages. We use FSDP 2 [3] 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` [22] and relies on native PyTorch features to implement advanced parallelism techniques, such as tensor, context, and expert parallelism, and leverages grouped matrix multiplication kernels for efficient MoE training.

**Inference.** The inference pool consists of standard OpenAI-compatible servers with a vLLM [20] backend. The API specification is extended with custom endpoints to enable updating the server with the latest policy: `/update_weights` is used to update the policy, and `/reload_weights` is used to reset the weights to the base model in between experiments. 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 [56], Tokasaurus [18]).

### 2.1.2 Asynchronous Off-Policy Training

In `prime-rl`, the trainer and inference run disaggregated, i.e. on a disjoint set of GPUs, to overlap rollout generation and training.

At each training step, all artifacts are identified by the step count $n$. For the trainer, this is the gradients $g_n$ and model weights $\theta_n$, and for the inference service, rollouts $(x_n, y_n)$. At step 0, the inference service uses $\theta_0$ (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 $\theta_1$ 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. For example, in one-step off-policy training, the inference service continues generating $(x_1, y_1)$ from $\theta_0$, while the trainer is producing $\theta_1$ in parallel. An example of such overlapped, asynchronous computation is shown in Figure 3.

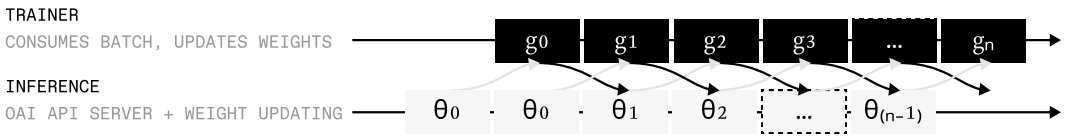



Figure 3: **Asynchronous Off-Policy Training.** We show the execution graph of one-step off-policy training in an idealized setting where the trainer step time equals the inference step time. At step $n$, the inference engine uses a policy no older than $\theta_{\min(0,n-1)}$.

### 2.1.3 Continuous Batching & In-Flight Weight Updates

Generating rollouts which are many tens of thousands of tokens long quickly becomes a major bottleneck in large-scale RL training. Traditional systems schedule $n$ rollout requests and only release a training batch once the slowest rollout has finished generation. However, this method is prone to heavily under-utilize inference compute because less-than-optimal rollouts are being generated concurrently as rollouts in the batch finish. This bottleneck becomes especially visible if there is high variance in the length of the generated rollouts, as is typical for reasoning models in complex agentic environments.

To make training viable, we implement continuous batching with in-flight policy weight updates, as popularized by AReal [11] and PipelineRL [37]. Two main asynchronous task loops on the orchestrator achieve this.

**In-Flight Weight Updates.** The orchestrator continuously polls the trainer to update the inference pool as soon as a new policy becomes available. Once this happens, the inference pool temporarily interrupts generation to receive the updated weights from the trainer. Once the weight update is complete, rollout generation continues with the updated weights. Thus, a single trajectory may be generated by multiple policies. We control the maximum off-policyness by discarding rollouts which are generated by more than `max_off_policy_steps` policies to prevent policy drift.

**Continuous Batching.** The orchestrator maintains a large pool of concurrent, asynchronous rollout requests. Whenever a rollout group completes, its slot is immediately repopulated with a new request. This keeps the pool saturated and ensures that a constant number of rollouts are in flight, thereby sustaining peak inference throughput without waiting for synchronous batch boundaries.

Figure 4 visualizes how rollout trajectories span multiple policies which are updated in-flight as they become available.

We find these optimizations critical to scale to long-context RL training, achieving much higher end-to-end system throughput, while minimizing data off-policyness.

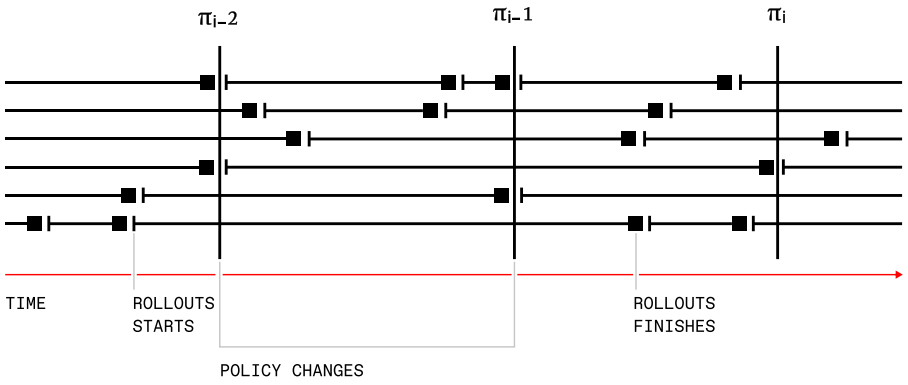



Figure 4: **Continuous Batching & In-Flight Weight Updates.** Continuous batching maintains a constant inference load because finished rollout are immediately replaced with new rollout requests. The policy used to generate rollouts is updated in-flight as soon as it becomes available, allowing rollouts to be generated by multiple policies.

#### 2.1.4 Multi-Client Orchestrator

When scaling inference to hundreds of GPUs, we found that the standard multi-node data-parallel strategy provided by vLLM did not deliver the expected throughput gains. As the number of inference nodes increased, overall performance quickly plateaued.

To overcome this limitation, we implemented a custom data-parallel strategy centered on a multi-client abstraction on the orchestrator. Each inference node is deployed as an entirely independent server, and the orchestrator maintains one client per node. Group rollout requests are distributed across clients using a simple round-robin mechanism, ensuring balanced utilization and eliminating any inter-node synchronization. We found that inference throughput scales linearly with the number of nodes used, as desired.

#### 2.1.5 Online Data Filtering

An effective reinforcement learning setup depends on a well-designed curriculum that exposes the model to tasks of appropriate and progressively increasing difficulty. Beyond pre-selecting data via offline data filtering, we implement advanced online data filtering techniques to continually adapt task difficulty during training.

To support this, problems are sorted into difficulty pools (easy, normal, hard) based on each problem's observed solve rate. By flexibly controlling how many samples are drawn from each pool at every step, we can maintain a balanced curriculum that avoids training with problems that are either too trivial or too challenging. In parallel, an online difficulty filter discards trivial rollouts—such as those that the model always fails or always solves—ensuring that we only train on rollouts with meaningful learning signal. Together, these mechanisms allow the curriculum to evolve as the model's capabilities increase, and continue to provide good learning signal.

#### 2.1.6 Scaling Sequence Length

Over the course of RL training, the sequence length of model generations naturally increases [8]. Training on these increasingly long rollouts while preserving efficiency is nontrivial. In our training setting, we managed to reach up to $48\text{k}$ sequence length with FSDP degree equal to 32, by leveraging aggressive activation check-pointing and Flash Attention 3 (FA3) [41]. However, for some of the more difficult environments, sequence length of at least $64\text{k}$ was required. To solve this, we explored the following two approaches:

**Context Parallelism.** With increasing sequence length, the attention score matrix becomes the dominant component of the training memory footprint. Algorithms such as `FlashAttention` [7] help mitigate this, but are often not sufficient. Context parallelism distributes the attention computation across $N_{\text{cp}}$ GPUs, reducing the memory footprint at the cost of communication overhead. The most common implementation is based on *Ring Attention* [24], which assigns separate `chunks of Q`, `K`,

and V to each GPU and rotates K and V among devices to compute the attention matrix. This algorithm is implemented in PyTorch; however, its implementation for `FlexAttention` [9] was experimental at the time of our training. Although we were able to scale the sequence lengths up to $256\text{k}$ using this approach with $N_{\text{cp}} = 2$, doing so effectively halved our data-parallelism degree and additionally exhibited accuracy degradations, making it unsuitable for our production training setting.

**Activation Offloading.** Our training utilized full activation checkpointing, meaning that only the outputs of each decoder layer and the top-level module activations are stored, while all intermediate activations are recomputed during the backward pass. Ignoring top-level activations, the activation memory for a sequence length of $48\text{k}$, hidden size $4096$, and $46$ decoder layers is

$$\text{Mem}_{\text{act}} = 46 \times (48{,}000 \times 4{,}096) \times 2 \text{ bytes} \approx 18 \text{ GB}.$$

To reduce this footprint, we offload activations to the CPU using an implementation based on `torchtune` [46]. This enabled us to scale the sequence length to $72\text{k}$ with the same hardware configuration as above, without any degradation in MFU. We observed a memory leak in the asynchronous offloading implementation when using CUDA streams, and therefore adopted the synchronous variant. Although this led to a slight decrease in MFU, the impact was negligible at roughly $0.1\%$.

### 2.1.7 Distributed Muon

As shown by [25], reusing the Muon optimizer during post-training yields the best performance when the model was initially pretrained with Muon. Unlike SGD or Adam, which apply element-wise updates, Muon [17] operates at the matrix level: its Newton-Schulz update requires access to the full gradient tensor. Consequently, we cannot directly apply Muon to FSDP-sharded gradients. Gathering full gradients on every rank and redundantly performing the same Muon computation would be prohibitively expensive.

To address this, we explore two strategies for distributing Muon across multiple nodes. Our first approach uses an overlapping round-robin scheme: each rank gathers a subset of FSDP-sharded gradients based on its index, computes the Newton-Schulz update locally, and then scatters the updated gradients back to all FSDP ranks. This approach parallelizes the expensive computation across ranks and can hide the additional communication. However, at large multi-node scales, issuing many overlapping gathers leads to InfiniBand congestion.

We therefore adopt a more efficient method based on all-to-all collectives, which reshuffles gradient shards without relying on many individual gathers. Although this design is less flexible and may require padding tensors before communication, it significantly improves performance and avoids congestion at scale. We use this all-to-all–based algorithm for our main training runs, leveraging the open-source implementation provided in Dion [2].

### 2.1.8 Efficient Mixture-of-Experts Support

We use the Mixture-of-Experts (MoE) [42] layer implementation from `torchtitan` [22] to leverage the efficient grouped matrix multiplication kernels for expert execution which comes with support for expert parallelism (EP). We found that enabling EP led to worse training throughput and thus did not enable it for our training. The reduced throughput can be attributed to the relatively large sequence length and hidden dimension on each GPU for our training runs.

We train with relatively large sequence length and hidden dimension and are thus in the regime depicted in Figure 5 where we already saturate the grouped gemm kernel without the need for decreasing the number of experts per GPU with expert parallel. Expert parallel in this regime will lead to increased overhead from the scatter and gather without decreasing the time spent performing the grouped gemm. Training that uses lower sequence length, hidden dimension or utilizes parallelizations that decrease the amount of work per GPU like context parallel and tensor parallel might see an improvement from using expert parallelism.

To maintain compatibility with HF, as for example required by our vLLM inference engine, we transform the state dict on-the-fly during the broadcasting from the `torchtitan`-based MoE layer implementation used by the trainer to the HF-based MoE layer implementation used by the inference engine.

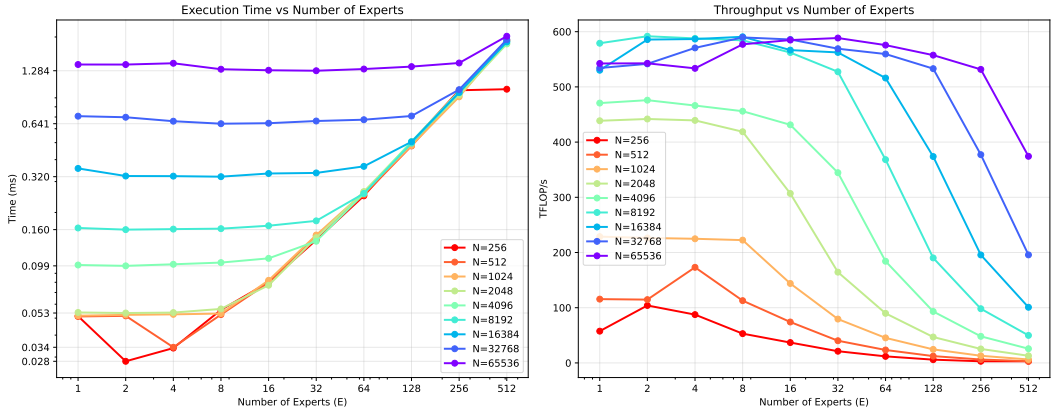



Figure 5: Execution time and TFLOPS of `torch._grouped_mm` with hidden dim 4096 and MoE dim 1408 on H200 SXM at different sequence lengths and number of experts. We assume that the input is perfectly balanced between the experts and thus an increase in experts leads to an inversely proportional decrease in the number of tokens and work per expert, eventually causing lower TFLOPS as the work per expert is no longer able to saturate the kernel. At sequence lengths (N) $32,768$ and $65,536$, the TFLOPS remains in the saturated regime up to 128 experts. We thus do not gain significant throughput from using expert parallel given our training parameters.

To monitor expert load distribution, we compute and log the maximum violation load-balancing metric $\text{MaxViolation} = \frac{\max_i \text{Load}_i - \overline{\text{Load}_i}}{\overline{\text{Load}_i}}$ as described in [47]. This metric quantifies the degree to which expert load imbalance slows down the MoE layer relative to an ideally balanced configuration. Since imbalance directly affects both training and inference throughput, it provides a useful diagnostic of system efficiency.

### 2.2 Verifiers: Environments for LLM Reinforcement Learning

We train INTELLECT-3 using environments built with our `verifiers` library, which provides a set of modular and extensible components for concisely expressing complex environment logic while maintaining highly scalable performance. 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, allowing environments to be developed, tested, and versioned independently of the trainer.

#### 2.2.1 Environment Design

Environments built with `verifiers` are installable Python modules, and consist of:

- a `dataset` where each row corresponds to a task example, including the input prompt and any necessary metadata for execution scoring (e.g. ground-truth answer, test cases);
- a `rollout` method which takes as input a dataset row and an OpenAI-compatible inference client, executes all steps of action until a termination condition is reached, and collects all information required for training (e.g. token ids, logprobs);
- a `Rubric` object which includes one or more reward functions (operating either on a per-rollout or per-group basis), and logs final scores and metrics upon completion of rollouts;
- a `load_environment` method implemented in the environment module which instantiates the environment, handling any necessary preprocessing and resource provisioning.

**Rollout Orchestration.** Rollouts are executed asynchronously via `asyncio`, allowing thousands of concurrent rollouts to proceed in parallel. Inference requests, tool calls, and reward functions are dispatched and awaited independently of other in-flight rollouts. Parallelism occurs at multiple granularities: we replicate across inference workers, API clients, and environment processes via a

central orchestrator. We apply fine-grained semaphore-based throttling to ensure inference workers are kept busy while minimizing KV cache eviction.

**Rubrics and Reward Functions.** The `Rubric` abstraction manages reward computation with support for multiple weighted reward functions. Each reward function receives the prompt, completion, ground-truth answer, and rollout state, and returns a scalar score. Scores from multiple reward functions are combined via configurable weights to produce a final reward signal.

For more complex scoring scenarios, rubrics can be composed to aggregate multiple scoring strategies (e.g. combining a format-checking rubric with an LLM judge rubric). The scoring interface can also be overridden to implement inter-group comparisons such as voting, ranking, or relative scoring across samples from the same problem.

**Evaluations.** We additionally use `verifiers` environments directly for evaluation, both offline (via a standalone CLI which runs remote-hosted evaluations via the Environments Hub) and online during training. The same `rollout` and `Rubric` entrypoints can be used for either training or evaluation, ensuring consistency across deployment settings.

**Extensibility via Class Inheritance.** The environment hierarchy provides progressive specialization for common use cases. The base class handles dataset management, prompt formatting, and the core generate/score pipeline. Multi-turn environments extend this with a rollout loop that alternates between model responses and environment responses until a termination condition is met. Single-turn environments provide a minimal specialization for tasks requiring only a single model response. Tool-calling environments further specialize multi-turn behavior with native OpenAI-format tool calling: tool definitions are automatically converted to the API schema, tool calls in model responses are parsed and executed, and results are appended as tool messages. Custom environments inherit from the appropriate base class and override methods to implement task-specific logic such as termination conditions and environment response generation.

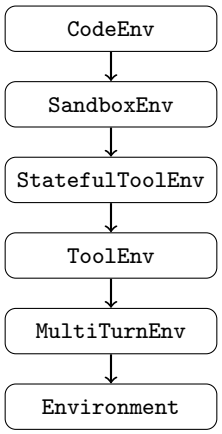



Figure 6: **Inheritance Hierarchy.** Class hierarchy for `CodeEnv`, used in the `primeintellect/i3-code` environment. Each level adds functionality: `Environment` provides the core abstraction; `MultiTurnEnv` adds iterative rollout logic; `ToolEnv` adds OpenAI-format tool calling; `StatefulToolEnv` enables injecting tool arguments that depend on rollout state (e.g. resource IDs); `SandboxEnv` manages containerized execution environments; and `CodeEnv` runs test cases against the code generated by the LLM.

### 2.2.2 Integration with `prime-rl`

`prime-rl` has first-class support for `verifiers` environments, which are installed as standalone Python modules via the Environments Hub[5]. Environments can be developed and tested in isolation against local or API models, then pushed to the Environments Hub and used immediately in training without any code modification in `prime-rl`. The orchestrator loads environments by their Python

[5] `https://hub.primeintellect.ai`

module identifiers, invokes them with batches of inputs and inference clients, and receives finished rollout states, including rewards, token IDs, logprobs (directly from vLLM), and attention masks.

**Multi-Environment RL Training.** The `EnvGroup` pattern in `verifiers` allows the combination of multiple environments into a single class object with concatenated datasets, where an injected task ID column is used to route rollout and scoring logic across appropriate sub-environments. We leverage this functionality in `prime-rl` to support simultaneous training of INTELLECT-3 across many environments without needing any explicit multi-environment-aware code within the orchestrator after the `EnvGroup` is instantiated.

#### 2.2.3 Environments Hub

Many RL frameworks treat environments as subfolders within a central training repository, tightly coupling environment logic to training infrastructure. This complicates versioning, makes it difficult to run controlled ablations across environment variants, and creates friction for external contributors who must navigate the full training codebase to add or modify tasks or run offline evaluations.

The Environments Hub addresses these issues by providing an open registry where environments built with `verifiers` are packaged as standalone Python modules with pinnable dependencies and a standardized entry point. Environments can be versioned, shared, tested, and iterated on outside of the context of training code, enabling researchers to contribute new tasks independently. Combined with our open-source `prime-rl` trainer, sandboxes for secure code execution, and distributed compute infrastructure, the Environments Hub forms part of a full infrastructure stack for open RL research, lowering the barrier for anyone to train, evaluate, and fine-tune models on custom tasks.

#### 2.2.4 Evaluations

Evaluation and training are tightly coupled through the use of `verifiers`, streamlining the process of evaluating against a wide range of common benchmarks. 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 evaluation requests with training requests using the same inference pool as the trainer, effectively hiding evaluation overhead while providing real-time feedback of training performance.

### 2.3 Prime Sandboxes: Code Execution for RL Training

Executing untrusted code for thousands of concurrent rollouts requires a container orchestration layer capable of sub-second provisioning and millisecond-level execution latency. While Kubernetes provides the primitives for container management, standard architectural patterns are insufficient for the throughput required by high-velocity training.

#### 2.3.1 The Limits of Naive Orchestration

A standard, naive implementation of a remote execution sandbox typically relies on the Kubernetes API Server. In this design, the training loop utilizes standard client libraries to spawn ephemeral pods and executes commands via `kubectl exec`. This operation relies on upgrading an HTTP request to a WebSocket connection, which is then proxied through the Kubelet to the container runtime. Empirical testing revealed that this approach is fundamentally unscalable. While the code execution itself should take milliseconds, the orchestration overhead creates latencies measured in seconds. Because every execution command involves an authenticated API request that is logged and persisted in `etcd`, the control plane becomes the primary bottleneck. Etcd is Kubernetes' distributed key-value store that maintains cluster state; every API operation must be serialized through write locks, creating a fundamental throughput ceiling. At a scale of thousands of concurrent sandboxes, we measured execution latency spiking to 2.5 seconds per command due to API server saturation and `etcd` write-lock contention.

#### 2.3.2 Prime Sandboxes Architecture

To overcome these control plane limitations, we developed *Prime Sandboxes*, which is a cloud based, Sandbox execution infrastructure which covers the requirements for agentic RL at scale. We re-

engineered the communication path to bypass the Kubernetes API for the critical execution loop. We introduced a high-performance Rust-based Gateway that accepts execution requests via a lightweight HTTP API. Instead of routing through the Kubernetes control plane, this Gateway communicates directly with the sandbox pods via Kubernetes Headless Services. A Headless Service allows the Gateway to resolve the direct IP addresses of individual pods via DNS, bypassing the overhead of `kube-proxy` load balancing.

However, relying on DNS for thousands of ephemeral pods creates a secondary scalability challenge. Standard CoreDNS configurations can become overwhelmed by the high churn rate of A-records associated with headless services, leading to resolution throttling. To ensure the Gateway can resolve pod IPs in milliseconds, we deployed a custom, high-throughput CoreDNS architecture optimized for high-velocity record updates.

The sandbox pod itself utilizes a Sidecar pattern. A privileged sidecar container functions as an execution agent. Upon receiving a request from the Gateway, the sidecar utilizes `nsenter` to inject commands directly into the target's namespace. This architecture achieves the speed of a local process spawn while maintaining full container isolation.

#### 2.3.3 Asynchronous Lifecycle Management

Reliably detecting when a sandbox is ready for execution is a critical latency path. The naive approach involves either actively polling the Kubernetes API to check for the `Running` state or relying on a standard Controller to watch the event stream. While sufficient for low-volume workloads, this method fails under high concurrency. Polling thousands of pods creates immense read pressure on the API server. Furthermore, standard controllers—specifically those built with Kopf (Kubernetes Operator Pythonic Framework)—typically process reconciliation queues sequentially. In a "thundering herd" scenario where thousands of pods initialize simultaneously, the resulting notification backlog causes the system to report a sandbox as "Ready" seconds after it has actually booted, leaving valuable compute resources idle.

To eliminate this latency, we inverted the status reporting flow. We utilize Kopf strictly for asynchronous maintenance tasks, such as error handling and resource finalization. For readiness signaling, we bypass the Kubernetes control plane entirely: the sandbox sidecar transmits a direct webhook to the training backend the moment it becomes operational. This "push-based" architecture ensures the training loop is notified within milliseconds of boot completion, achieving a consistent start-up time of under 10 seconds regardless of cluster load.

#### 2.3.4 Image Distribution and Infrastructure Density

Distributing container images to thousands of sandboxes simultaneously presents a distinct scalability barrier. A naive implementation that pulls images from public repositories (e.g., Docker Hub) hits fatal bottlenecks immediately: upstream registries enforce strict rate limits that block high-concurrency requests, and the sheer volume of data transfer saturates node network bandwidth, delaying boot times by minutes. To overcome these physical constraints, we architected a two-tiered image distribution strategy:

- **Custom Registry and Image Streaming:** For dynamic environments requiring unique dependencies, we host a private, high-throughput container registry. Crucially, we utilize Container Image Streaming (Lazy Pulling). Instead of blocking startup until the entire image manifest is downloaded, the runtime fetches only the data chunks necessary for the entrypoint process. The remaining layers are streamed in the background, allowing the sandbox to become operational seconds before the full image is physically present on the node.
- **Warm Pools:** For environments relying on static runtime images (e.g., standard Python distributions), we maintain a "warm pool" of pre-provisioned pods. This allows the training loop to acquire a sandbox instantly without incurring any image pull latency or initialization overhead.

Underlying this distribution layer is a custom Cluster Autoscaler and bin-packing scheduler designed for extreme density. We target a packing factor of 256 sandboxes per node and utilize the *Burstable* QoS class. Burstable QoS allows pods to request a baseline of CPU resources but burst above that

limit when available, which is ideal for RL workloads where sandboxes alternate between brief execution spikes and idle waiting periods. This allows the cluster to vastly oversubscribe CPU resources during the idle periods inherent in RL training steps, maximizing hardware efficiency without compromising peak execution throughput.

#### 2.3.5 Security and Capabilities

We utilize gVisor (runsc) as the container runtime, providing a user-space kernel that isolates the host from potential exploits within the untrusted code as well as configurable network policies to limit the communication allowed from within a sandbox. Beyond simple code execution, the architecture supports complex network requirements, allowing sandboxes to expose arbitrary TCP/UDP ports for HTTP traffic. Furthermore, the system is designed to support hardware acceleration, enabling the mounting of GPUs for environments dependent on custom GPU kernels.

### 2.4 Compute Orchestration: Frontier GPU Infrastructure

We deployed 512 NVIDIA H200 GPUs across 64 interconnected nodes. The primary engineering challenge lies in maintaining determinism and synchronization across a distributed system prone to hardware failures.

**Provisioning and Fabric.** To eliminate configuration drift, we enforce a strict Infrastructure as Code paradigm using idempotent Ansible playbooks that handle dynamic hardware discovery and automated firewall generation. Distributed training performance is bound by the tail latency of the *AllReduce* collective, so we utilize a 400Gbps NDR InfiniBand fabric (NVIDIA ConnectX-7) and validate throughput before every run, targeting $\geq$160 GB/s. When performance degrades, an automated binary search isolates straggler nodes with faulty transceivers.

**Orchestration.** We use Slurm with Cgroup v2 integration to guarantee resource reclamation—upon job termination, the kernel freezes and eliminates the entire cgroup hierarchy, preventing zombie processes from holding GPU memory. This provides container-like isolation without filesystem overhead.

**Storage.** A tiered architecture balances I/O performance: Lustre handles high-throughput operations (training trajectories, multi-terabyte checkpoints), while NVMe-backed NFS serves metadata-heavy user environments and enables keyless SSH across the fleet.

**Observability.** We monitor GPU telemetry via DCGM aggregated into Prometheus, with active alerting on Xid errors and thermal slowdown events. This allows proactive node draining before a failing component corrupts training progress.

## 3 INTELLECT-3 Training

We train INTELLECT-3 in two main stages: a supervised fine-tuning stage and a large-scale RL stage. We use GLM-4.5-Air base as our base model. Both stages, including multiple ablations, were carried out on a 512 H200 cluster over the course of two months. For the entire stack, from the RL environments to the evaluation of the model, we utilize open-source environments contributed to our Environments Hub.

### 3.1 Environments Mix

We train on a diverse and challenging mix of environments designed to enhance the reasoning and agentic capabilities of our model.

#### 3.1.1 Math

For our model to excel in mathematical problem solving, we carefully design our math environment [6] for long chain-of-thought reasoning. It consists of 21.2K challenging math prob-

[6] `environments/primeintellect/i3-math`

lems, curated from Skywork-OR1 [12], Acereason-Math [6], DAPO [53], and ORZ-Hard [15]. To verify the model's responses we parse out the final answer and compare it against the ground truth using `math-verify` [21]. In practice, we have found a non-negligible fraction of false negatives with mere rule-based verification. For this reason, we additionally employ `opencompass/CompassVerifier-7B` [27] as an LLM-judge to double-check all answers which are marked as wrong by the rule-based verifier. Finally, we difficulty annotate the entire dataset by computing the average solve rate of `Qwen/Qwen3-4B-Thinking-2507` [45] over eight generations per problem. We use these annotations to filter our samples which are too easy at various stages of our post-training pipeline.

#### 3.1.2 Code

Our code environment[7] tasks the model with single-turn programming challenges in Python. The environment takes inspiration from DeepCoder [31] and makes heavy use of our `SYNTHETIC-2` dataset [39]. Solutions are verified by executing up to 15 test cases per problem inside *Prime Sandboxes*. During training, asynchronous and isolated execution per solution is facilitated by over 4000 concurrent sandboxes. On any sandbox failure, we mask out the corresponding model completion. For difficulty filtering, we annotate the 8.6K examples with the average solve rate of `Qwen/Qwen3-4B-Instruct-2507` [45] over eight generations per problem.

#### 3.1.3 Science

We use the science environment[8] to improve our model's capabilities in domains such as physics, chemistry, and biology. Similarly to our math environment, we use both math-verify and an LLM-judge to verify the answers. The dataset we use consists of 29.3K challenging problems spanning diverse domains curated and filtered from MegaScience [10]. Again, we annotate the dataset for difficulty by computing the average solve rate of `Qwen/Qwen3-4B-Instruct-2507` [45] over 16 generations per problem, which we use to filter out too easy samples at various stages of the post-training pipeline.

#### 3.1.4 Logic

Our logic environment[9] includes a diverse set of 29 logical tasks, puzzles and games, such as evaluating boolean expressions, solving cross-word puzzles or Sudoku, or playing Minesweeper. Both the 11.6K problems and verifiers were adapted from SynLogic [26]. Similarly to our other environments, we compute the solve rate of `Qwen/Qwen3-4B-Instruct-2507` [45] over 16 trials to gauge the difficulty of the problems, in order to selectively filter out samples during post-training.

#### 3.1.5 Deep Research

Our web search environment[10] [29] provides the model with a search tool that uses Serper [40] to return an enumerated list of search results, a click tool to retrieve the markdown-formatted textual content of webpage chosen by its index in the previous search results, an open tool that returns a website's content in the same way as the click tool, given a URL, and a finish tool which the model uses to provide its final answer. The environment tasks the model with answering questions from the dataset using the given tools, and rewards it with 1 for a correct answer and 0 for an incorrect one. Optionally, a redundancy penalty can be applied to the queries of subsequent searches, which we set to zero. We leverage a corpus of training examples from z-AI's DeepDive dataset, which consists of complex, multi-step questions extracted from open knowledge graphs with the help of LLMs. It contains 1K samples for SFT trajectory generation and 2.2K samples for RL.

To validate the implementation of our environment, we trained `Qwen/Qwen3-4B-Instruct-2507` [45] using SFT on the public DeepDive traces for 26 steps at batch size 34, for a total of 884 samples, followed by 122 steps of RL at a group size of 16 and a total batch size of 512. Figure 7 shows the mean reward per step over the course of RL

[7] `environments/primeintellect/i3-code`
[8] `environments/primeintellect/i3-science`
[9] `environments/primeintellect/i3-logic`
[10] `environments/primeintellect/deepdive`

training. The success of a 4 billion parameter model to learn Deep Research via RL on DeepDive demonstrates the correctness and usability of our DeepDive environment implementation.

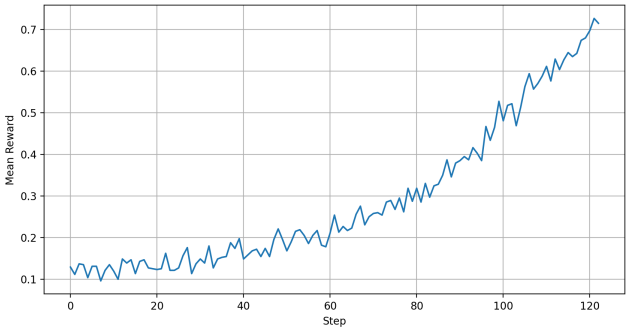



Figure 7: Mean reward of `Qwen/Qwen3-4B-Instruct-2507` [45] over RL training steps on DeepDive after a short SFT phase.

#### 3.1.6 Software Engineering

We developed two Software Engineering (SWE) environments [11][12]. They implement two modified agent scaffolds, R2E-Gym [16, 30] and mini-swe-agent-plus [48]. Further, they include three sandbox harnesses supporting common formats for SWE datasets, images and test suites like R2E-Gym [16], SWE-smith [52] and Multi-SWE-bench [54]. For the R2E-Gym scaffold, we swapped out the `finish()` for the `submit()` tool as the `result` parameter serves no use internally and is being called verbosely by the model. We adapt the mini-swe-agent-plus scaffold for native reasoning and tool use by changing its prompt and replacing code block parsing with tool calling. Inside a sandbox the agent can navigate the repository of a given Github project and is tasked with fixing an issue. The scaffold equips the model with tools for executing Bash commands and editing files. The maximum number of turns the agent can take is capped at 200. After submitting the solution, the test suite of the repository runs to determine whether the correct tests change their status from failing to passing. As with our code environment, when a given sandbox fails we mask out the model's completion and cancel the generation. To render ephemeral and nearly instant rollouts possible we deploy our Custom Registry with *Prime Sandboxes* hosting over 20,000 images containing pre-installed Github repositories [13].

### 3.2 Supervised Fine-Tuning

Prior to RL, we conduct two distinct supervised fine-tuning stages: a large-scale general chat and reasoning SFT stage, designed to improve the conversational and reasoning capabilities of our model, and an agentic SFT stage, designed to improve our model's ability to efficiently use tools in long-running agentic contexts. Both stages are designed to provide a strong prior for the subsequent RL stage.

Prior to RL, we run two complementary supervised fine-tuning (SFT) phases. First, we conduct a large-scale general chat-and-reasoning SFT stage that strengthens the model's conversational abilities and core reasoning skills. Second, we run an agentic SFT stage focused on improving the model's ability to use tools effectively and operate within long-horizon, agentic workflows. Together, these stages establish a strong prior and a stable behavioral foundation for the subsequent RL phase.

**General Reasoning SFT.** For our first SFT stage, we construct a large-scale dataset spanning diverse domains and leverage many high-quality, permissive open-source datasets. Our two main sources are the math, code, science, and tool splits from NVIDIA's `Nemotron-Post-Training-Dataset-v1` [33, 4] and the chat and instruction following splits from AM's `AM-DeepSeek-R1-0528-Distilled` [1] dataset. Both contain synthetically generated reasoning traces from `DeepSeek-R1-0528`. During training we respect the natural ratios of the datasets.

[11] `environments/primeintellect/deepswe`
[12] `environments/primeintellect/mini-swe-agent-plus`
[13] `Prime Sandboxes`

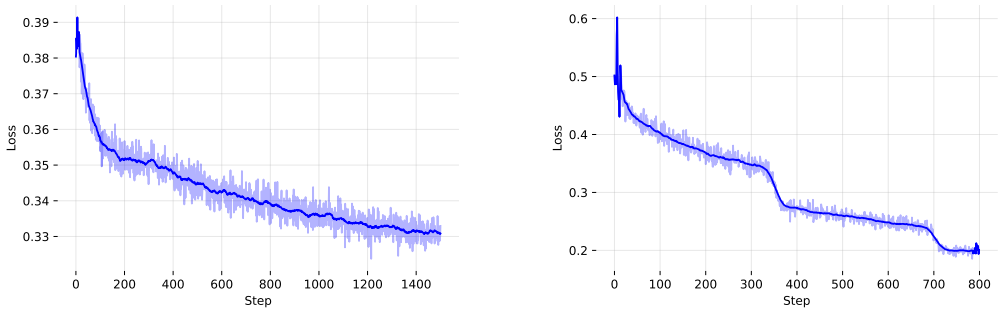
Table 1: **SFT Data Sources**

| **Dataset** | Num. Examples | Num. Tokens | Stage 1 | Stage 2 |
|---|---|---|---|---|
| OpenReasoning-Math | 2M | 78.1B | ✓ | ✓ |
| OpenReasoning-Code | 1.9M | 94.3B | ✓ | ✓ |
| OpenReasoning-Science | 310K | 32B | ✓ | ✓ |
| OpenReasoning-Tool | 800K | 3.8B | ✓ | ✓ |
| AM General Chat | 952K | 8.4B | ✓ | ✓ |
| AM Instruction Following | 54K | 400M | ✓ | ✓ |
| SWE Swiss | 10.3K | 700M | | ✓ |
| Toucan Tool | 116K | 700M | | ✓ |
| Environments Mix | 38.4K | 1.9B | | ✓ |

We train a full epoch with ∼33M tokens per step at context length 65K. We use the Muon optimizer with weight decay 0.01 and learning rate 5e-5, warmed up linearly from 1e-8 over 300 steps. We rely on FSDP with a world size of 64 to efficiently shard the model and DP replicate size 8, spanning training across the full cluster of 512 GPUs

**Agentic SFT.** Following the general chat-and-reasoning SFT phase, we conduct a second supervised fine-tuning stage targeted at agentic behavior, tool use, and long-horizon control. Whereas the first SFT phase focuses on conversational competence and long chain-of-thought reasoning in STEM domains, the agentic SFT phase is smaller, and curated to endow the model with robust capabilities for calling external tools, maintaining coherent state over long-running tasks, and operating effectively in extended sequences. To this end, we combine multiple open-source agentic datasets such as SWE-Swiss [13], and Toucan Tool [51], as well as synthetically-generated datasets created from other environments on the Environments Hub using `DeepSeek-R1-0528`. All datasets were processed to ensure consistent tool call formatting, filtered for English content, and standardized so that they are compatible with our trainer.

This stage also serves a complementary purpose: pushing the model toward longer effective context lengths, ensuring stability and competence beyond 65K context window. We leverage context parallelism (CP) to effectively scale to 98K context length training, ensuring that our model learns to maintain consistency in long-running agentic tasks. We train for two epochs, resuming from our final stage 1 checkpoint. Again, we use the Muon optimizer starting with a learning rate of 5e-8, and decay it linearly over the full 800 steps of training.




(a) Stage 1 (b) Stage 2

Figure 8: **Supervised Fine-Tuning.** We show the loss curves of our general reasoning SFT stage (left) and the agentic SFT stage (right) over the course of training. Both runs show smooth optimization without any loss spikes.

**Chat Template.** The INTELLECT-3 chat template is inspired by Qwen3- and GLM-family of models. The template uses familiar control tokens—such as `<|system|>`, `<|user|>`, and `<|assistant|>`—to mark roles in multi-turn conversations, and `<|im_start|>` and `<|im_end|>` to delimit conversational turns. Tool calls follow XML-style tagging format.

We introduce the following modifications: First, our model always reasons, with no user-exposed reasoning-effort controls. This is implicitly baked into the model by dominantly training on reasoning-only SFT traces, and explicitly by setting appending a `<|think|>` token via the chat template. To ensure proper usage of our model, use the `qwen3_coder` tool call parser, and the `deepseek_r1` reasoning parser. To retain thinking across turns, the chat template automatically parses `reasoning_content` field to ensure reasoning chains are consistently represented without requiring manual formatting.

### 3.3 Reinforcement Learning

For the RL portion of the training, we used a batch size of 256 prompts with 16 rollouts per prompt and a maximum context length of 65536. We utilize online difficulty filtering, as well as an easy difficulty pool to remove any prompt with a pass rate of 1 from being sampled again, as these prompts would contribute no learning signal. Our `max_off_policy_steps` is set to 8 to ensure we remove any excessively off-policy rollout. We use Muon with a learning rate of $1e-6$. During training, the data mix is carefully tuned to balance performance across domains.

We use 60 nodes in total for our RL training, each with 8 H200s. We allocate nodes between training and inference at about a 1:3 ratio, with 16 nodes for training and 44 nodes for inference to get the best throughput. We observe a step time of $\sim 1500s$ per step when training at $65,536$ sequence length with in-flight weight updating. Without in-flight weight updating, we observe an increase in step time of more than $2\times$ as the inference is significantly less efficient.

**Training Algorithm.** We adopt masked token-level importance sampling [55]. For a batch of $N$ rollouts, we use the formulation below.

$$\mathcal{J}_{\text{IcePop}}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\{y_i\}_{i=1}^{N}\sim\pi_{\text{infer}}}\left[\frac{1}{\sum_{i=1}^{N}|y_i|}\sum_{i=1}^{N}\sum_{t=1}^{|y_i|}\left[\mathcal{M}\Big(\frac{\pi_{\text{train}}(y_{i,t}\mid x, y_{i,<t};\theta)}{\pi_{\text{infer}}(y_{i,t}\mid x, y_{i,<t};\theta_{\text{old}})};\alpha,\beta\Big)\ \widehat{A}_{i,t}\right]\right] \tag{1}$$

$$\mathcal{M}(k) = \begin{cases} k & \text{if } \ k\in[\alpha,\beta] \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $\pi_{\text{infer}}$ refers to the policy that generated the rollout, $\pi_{\text{train}}$ refers to the current trainer policy, $\mathcal{M}(\cdot;\alpha,\beta)$ is the masking function from Eq. 2. The token-level advantage is estimated as $\hat{A}_{i,t} = S_i - \text{mean}(\{S_i\}_i^G)$ [28] where $S_i$ is the reward given to rollout $i$, and $G$ being the number of rollouts for a given prompt and we default to $\alpha = 0.5$ and $\beta = 5$ [55].

We found double-sided masking critical to combat the trainer-inference mismatch: Even when $\pi_{\text{infer}}$ and $\pi_{\text{train}}$ and share the same parameters $\theta$, they can produce significantly different token probabilities, leading to unexpected distribution shifts that can cause runs to crash multiple days into the experiments, if not explicitly addressed. This is similar to CISPO [32] (later further validated in [19], however, we use masking instead of clipping to avoid noisy updates that come with excessive importance ratios. We also apply masking to any rollouts if any of its tokens importance ratio falls under a certain threshold (we use 1e-5 for our training).

**Online Evaluation** In Figure 9 we plot the model performance at 15 step intervals on AIME25, AIME24, LiveCodeBench, HLE and GPQA. The reasoning benchmark scores generally trend up and do not seem to have reached a plateau. This leads us to believe that allowing the model to continue training would yield continued improvements in the benchmark scores.

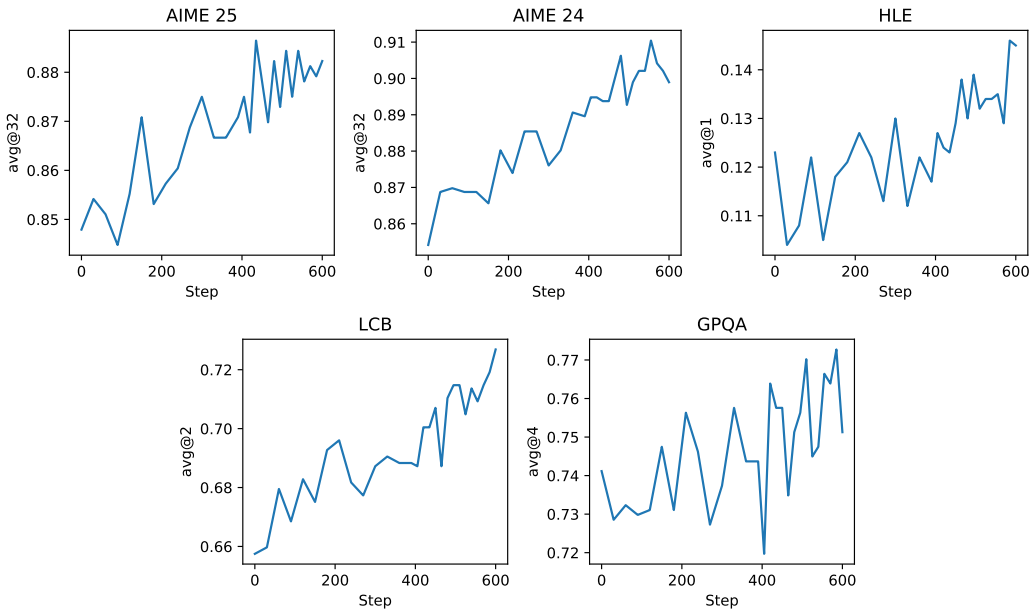



Figure 9: **Reinforcement Learning.** Reasoning benchmark scores as training progresses. The benchmarks scores generally trend up and do not appear to have reached a plateau.

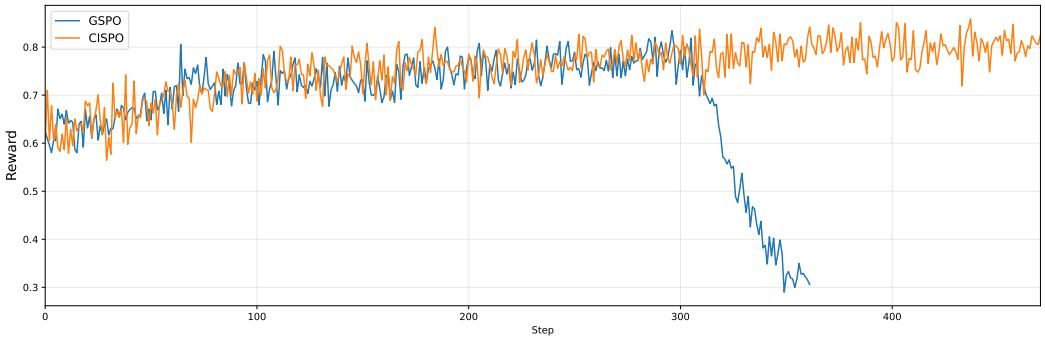



Figure 10: **Training Stability.** Early ablations of GSPO against CISPO (our algorithm at the time). We use async-8 as a testbed for algorithms to test if they would handle high levels of off-policyness. We observe strange reward (and all other metrics) collapse with GSPO also reported in [19] [38].

## 4 Evaluations

We evaluate INTELLECT-3 on a wide range of reasoning benchmarks, including AIME 2024[14], AIME 2025[15], LiveCodeBench v6[16], GPQA Diamond[17], HLE[18], MMLU-Pro[19],

To ensure a fair comparison, we run evaluations in the exact same settings against API services. Precise details on the evaluation setup are given in Appendix A.

Table 2 summarizes the results across all benchmarks. INTELLECT-3 outperforms the best matching comparison model, GLM-4.5 Air, which is Z.ai's post-trained version of the GLM-4.5 Air base model,

[14] primeintellect/aime2024
[15] primeintellect/aime2025
[16] primeintellect/livecodebench
[17] primeintellect/gpqa
[18] primeintellect/hle
[19] primeintellect/mmlu-pro

Table 2: **Evaluations.** We report benchmark scores on a wide range of reasoning benchmarks, and compare against models of similar or larger size. All implementations are open-source and reproducible via the Environments Hub.

| Benchmark | AIME24 | AIME25 | LCB v6 | GPQA | HLE | MMLU-Pro |
|---|---|---|---|---|---|---|
| INTELLECT-3 | 90.8 | 88.0 | 69.3 | 74.4 | 14.6 | 81.9 |
| GLM-4.5-Air | 84.6 | 82.0 | 61.5 | 73.3 | 13.3 | 73.9 |
| GLM-4.5 | 85.8 | 83.3 | 64.5 | 77.0 | 14.8 | 83.5* |
| GLM-4.6 | 92.0 | 90.3 | 73.0 | 78.8 | 13.3* | 83.1 |
| DeepSeek R1 0528 | 83.2 | 73.4 | 62.5 | 77.5 | 15.9 | 75.3 |
| DeepSeek v3.2 | 88.1 | 84.7 | 71.6 | 81.4 | 17.9 | 84.6 |
| GPT-OSS 120B | 75.8 | 77.7 | 69.9 | 70.0 | 10.6 | 67.1 |

across every tested benchmark. Even the $3\times$ larger GLM-4.5 is outperformed on many benchmarks by INTELLECT-3, including AIME 2024, AIME 2025 and LiveCodeBench v6.

At the end of RL training, rewards were still increasing with no sign of plateauing in benchmark performance. We will continue training INTELLECT-3 with a focus on agentic environments, to further improve the model's performance on complex agentic tasks.

## 5 Conclusion & Future Work

In this report, we present INTELLECT-3, a 100B+ parameters post-train on top of the GLM-4.5-Air base model. INTELLECT-3 achieves strong performance on math, code, science and broader reasoning benchmarks, and is competitive with or ahead of significantly larger frontier models. The model is trained on a diverse mixture of open environments from the Environments Hub, including math, code, science, logic, deep research and software engineering tasks, which together target long context tool using agentic behavior.

To support this training run, we introduced a frontier infrastructure stack for reinforcement learning at scale. `prime-rl` provides a production grade asynchronous RL framework with disaggregated trainer and inference, continuous batching, in flight weight updates and efficient support for Mixture-of-Experts models. The `verifiers` library and the Environments Hub standardize how environments and evaluations are expressed, turning them into reusable, versioned artifacts that can be shared, mixed and reproduced across projects. *Prime Sandboxes* and make it possible to execute untrusted code at very high throughput on thousands of concurrent rollouts, and to sustain long multi-week training runs on a 512 H200 cluster.

By open sourcing INTELLECT-3, the environments, and the complete training framework, we aim to narrow the gap between proprietary RL pipelines and what independent researchers, small labs and companies can build. The same code that produced INTELLECT-3 is available for single node experiments, mid scale research runs, and for production-scale training. Our hope is that this stack becomes a common foundation for the next generation of open reasoning and agentic models.

There are several directions for future work.

- **Scaling Agentic RL.** By the end of our current RL run, reward and evaluation curves had still not flattened, and the training remained extremely stable. This suggests that INTELLECT-3 is very much still in the high return regime of additional RL compute. With more agentic environments such as DeepDive and Software Engineering in the mix, we expect substantial further gains from simply continuing to train inside these RL environments, particularly for complex agentic use cases.
- **Richer RL environments.** Over the last few months, more than 500 RL environments have been released on the Environments Hub, and we've continued scaling this through our RL Residency and Bounty programs. These environments cover autonomous AI research, computer use, theorem proving, browser automation, and many domain-specific tasks such as law, finance, and tax. INTELLECT-3 used only a small slice of what's already available.

* Reported by AA Index

A major next step is scaling RL across a much broader set of high-quality community-contributed environments, covering more tools, modalities, and real-world workloads.

- **Long Horizon Agents.** A key next step on our research roadmap is making long horizon behavior RLable by letting the model manage its own context. We've been exploring simple tools for cutting context, prompting itself in isolated sub branches, and maintaining an external memory across turns. These keep the scaffold minimal and let the model learn end-to-end context handling through RL, with future work scaling training on environments that reward effective long-horizon reasoning. In line with recent evidence of "context rot" in long-context models, where the effective reasoning window is much smaller than the advertised context size and performance degrades on long-range reasoning tasks despite successful retrieval of relevant spans [23], we treat the context window as a scarce resource to be actively managed rather than a passive, ever growing transcript.

## References


[1] a-m team. Am-deepseek-r1-0528-distilled, June 2025.

[2] Kwangjun Ahn, Byron Xu, Natalie Abreu, and John Langford. Dion: Distributed orthonormalized updates. *arXiv preprint: 2504.05295*, 2025.

[3] 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.

[4] Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, Ido Shahaf, Oren Tropp, Ehud Karpas, Ran Zilberstein, Jiaqi Zeng, Soumye Singhal, Alexander Bukharin, Yian Zhang, Tugrul Konuk, Gerald Shen, Ameya Sunil Mahabaleshwarkar, Bilal Kartal, Yoshi Suhara, Olivier Delalleau, Zijia Chen, Zhilin Wang, David Mosallanezhad, Adi Renduchintala, Haifeng Qian, Dima Rekesh, Fei Jia, Somshubra Majumdar, Vahid Noroozi, Wasi Uddin Ahmad, Sean Narenthiran, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Siddhartha Jain, Igor Gitman, Ivan Moshkov, Wei Du, Shubham Toshniwal, George Armstrong, Branislav Kisacanin, Matvei Novikov, Daria Gitman, Evelina Bakhturina, Jane Polak Scowcroft, John Kamalu, Dan Su, Kezhi Kong, Markus Kliegl, Rabeeh Karimi, Ying Lin, Sanjeev Satheesh, Jupinder Parmar, Pritam Gundecha, Brandon Norick, Joseph Jennings, Shrimai Prabhumoye, Syeda Nahida Akter, Mostofa Patwary, Abhinav Khattar, Deepak Narayanan, Roger Waleffe, Jimmy Zhang, Bor-Yiing Su, Guyue Huang, Terry Kong, Parth Chadha, Sahil Jain, Christine Harvey, Elad Segal, Jining Huang, Sergey Kashirsky, Robert McQueen, Izzy Putterman, George Lam, Arun Venkatesan, Sherry Wu, Vinh Nguyen, Manoj Kilaru, Andrew Wang, Anna Warno, Abhilash Somasamudramath, Sandip Bhaskar, Maka Dong, Nave Assaf, Shahar Mor, Omer Ullman Argov, Scot Junkin, Oleksandr Romanenko, Pedro Larroy, Monika Katariya, Marco Rovinelli, Viji Balas, Nicholas Edelman, Anahita Bhiwandiwalla, Muthu Subramaniam, Smita Ithape, Karthik Ramamoorthy, Yuting Wu, Suguna Varshini Velury, Omri Almog, Joyjit Daw, Denys Fridman, Erick Galinkin, Michael Evans, Katherine Luna, Leon Derczynski, Nikki Pope, Eileen Long, Seth Schneider, Guillermo Siman, Tomasz Grzegorzek, Pablo Ribalta, Monika Katariya, Joey Conway, Trisha Saar, Ann Guan, Krzysztof Pawelec, Shyamala Prayaga, Oleksii Kuchaiev, Boris Ginsburg, Oluwatobi Olabiyi, Kari Briski, Jonathan Cohen, Bryan Catanzaro, Jonah Alben, Yonatan Geifman, Eric Chung, and Chris Alexiuk. Llama-nemotron: Efficient reasoning models, 2025.

[5] William Brown. Verifiers: Environments for LLM Reinforcement Learning. `https://github.com/primeintellect-ai/verifiers`, 2025.

[6] Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning, 2025.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[8] 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.

[9] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels, 2024.

[10] Run-Ze Fan, Zengzhi Wang, and Pengfei Liu. Megascience: Pushing the frontiers of post-training datasets for science reasoning, 2025.

[11] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning, 2025.

[12] 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.

[13] Zhenyu He, Qingping Yang, Wei Sheng, Xiaojian Zhong, Kechi Zhang, Chenxin An, Wenlei Shi, Tianle Cai, Di He, Jiaze Chen, and Jingjing Xu. Swe-swiss: A multi-task fine-tuning and rl recipe for high-performance issue resolution.

[14] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021.

[15] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.

[16] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

[17] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024.

[18] 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.

[19] Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S. Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms, 2025.

[20] 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.

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

[22] 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.

[23] Wei Hern Lim. Evaluating long context (reasoning) ability. `https://nrehiew.github.io/blog/long_context/`, 2025. Accessed 25 Nov 2025.

[24] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[25] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for llm training, 2025.

[26] Junteng Liu, Yuanxiang Fan, Zhuo Jiang, Han Ding, Yongyi Hu, Chi Zhang, Yiqi Shi, Shitong Weng, Aili Chen, Shiqi Chen, Yunan Huang, Mozhi Zhang, Pengyu Zhao, Junjie Yan, and Junxian He. Synlogic: Synthesizing verifiable reasoning data at scale for learning logical reasoning and beyond, 2025.

[27] Shudong Liu, Hongwei Liu, Junnan Liu, Linchen Xiao, Songyang Gao, Chengqi Lyu, Yuzhe Gu, Wenwei Zhang, Derek F. Wong, Songyang Zhang, and Kai Chen. CompassVerifier: A Unified and Robust Verifier for Large Language Models. 2025.

[28] 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.

[29] Rui Lu, Zhenyu Hou, Zihan Wang, Hanchen Zhang, Xiao Liu, Yujiang Li, Shi Feng, Jie Tang, and Yuxiao Dong. Deepdive: Advancing deep search agents with knowledge graphs and multi-turn rl, 2025.

[30] Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl, 2025. Notion Blog.

[31] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level, 2025. Notion Blog.

[32] MiniMax, :, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Enwei Jiao, Haigang Zhou, Haimo Zhang, Han Ding, Haohai Sun, Haoyu Feng, Huaiguang Cai, Haichao Zhu, Jian Sun, Jiaqi Zhuang, Jiaren Cai, Jiayuan Song, Jin Zhu, Jingyang Li, Jinhao Tian, Jinli Liu, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kaiyi Feng, Ke Yang, Kecheng Xiao, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Li, Lin Zheng, Linge Du, Lingyu Yang, Lunbin Zeng, Minghui Yu, Mingliang Tao, Mingyuan Chi, Mozhi Zhang, Mujie Lin, Nan Hu, Nongyu Di, Peng Gao, Pengfei Li, Pengyu Zhao, Qibing Ren, Qidi Xu, Qile Li, Qin Wang, Rong Tian, Ruitao Leng, Shaoxiang Chen, Shaoyu Chen, Shengmin Shi, Shitong

Weng, Shuchang Guan, Shuqi Yu, Sichen Li, Songquan Zhu, Tengfei Li, Tianchi Cai, Tianrun Liang, Weiyu Cheng, Weize Kong, Wenkai Li, Xiancai Chen, Xiangjun Song, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xinzhu Hou, Xuan Lu, Xun Zou, Xuyang Shen, Yan Gong, Yan Ma, Yang Wang, Yiqi Shi, Yiran Zhong, Yonghong Duan, Yongxiang Fu, Yongyi Hu, Yu Gao, Yuanxiang Fan, Yufeng Yang, Yuhao Li, Yulin Hu, Yunan Huang, Yunji Li, Yunzhi Xu, Yuxin Mao, Yuxuan Shi, Yuze Wenren, Zehan Li, Zelin Li, Zhanxu Tian, Zhengmao Zhu, Zhenhua Fan, Zhenzhen Wu, Zhichao Xu, Zhihang Yu, Zhiheng Lyu, Zhuo Jiang, Zibo Gao, Zijia Wu, Zijian Song, and Zijun Sun. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025.

[33] Dhruv Nathawani, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Ameya Sunil Mahabaleshwarkar, , Jian Zhang, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v1, 2025.

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

[35] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b gpt-oss-20b model card, 2025.

[36] Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, and et al. Humanity's last exam, 2025.

[37] Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. Pipelinerl: Faster on-policy reinforcement learning for long sequence generation, 2025.

[38] Penghui Qi, Zichen Liu, Xiangxin Zhou, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Defeating the training-inference mismatch via fp16, 2025.

[39] Mika Senghaas, Justus Mattern, Jannik Straube, Jack Min Ong, Manveer Basra, Andrew Baker, and Johannes Hagemann. SYNTHETIC-2 Release: Four Million Collaboratively Generated Reasoning Traces. `https://www.primeintellect.ai/blog/synthetic-2-release`, July 2025. Accessed: 2025-11-24.

[40] Serper. Serper: Google search api. https://serper.dev, 2025.

[41] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[42] 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.

[43] 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.

[44] 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.

[45] Qwen Team. Qwen3 technical report, 2025.

[46] torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024.

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

[48] Qi Wang, Hongzhi Zhang, Jia Fu, Kai Fu, Yahui Liu, Tinghai Zhang, Chenxi Sun, Gangwei Jiang, Jingyi Tang, Xingguang Ji, Yang Yue, Jingyuan Zhang, Fuzheng Zhang, Kun Gai, and Guorui Zhou. Klear-agentforge: Forging agentic intelligence through posttraining scaling, 2025.

[49] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark, 2024.

[50] xAI. Grok 4. News announcement.

[51] Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments, 2025.

[52] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS 2025 DB Spotlight)*, 2025. arXiv:2504.21798, accepted at NeurIPS 2025 (Spotlight).

[53] 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.

[54] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025.

[55] Xin Zhao, Yongkang Liu, Kuan Xu, Jia Guo, Zihao Wang, Yan Sun, Xinyu Kong, Qianggang Cao, Liang Jiang, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Small leak can sink a great ship–boost rl training on moe with icepop!, Sep 2025.

[56] 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 Reproducing Evaluations

## A.1 Evaluation Environments

**MATH-500.** MATH-500 consists of 500 high-school competition math problems distilled from the original MATH [14] dataset. To verify responses, we parse reasoning content, extract the final answer from the last `\boxed{...}`, and use `math-verify` to compare the answer against the ground truth. We run two generations per problem, for a total of 1000 generations.

**AIME.** We evaluate both AIME 2024 and AIME 2025, which each consist of 30 challenging high-school competition math problems. Similar to the MATH 500 environment, we parse reasoning content, extract the final answer from the last `\boxed{...}`, and use `math-verify` to verify responses. We do not employ a LLM-judge for verification thus our reported numbers are more conservative compared to e.g. Artificial Analysis Index, which uses LLM judges. To obtain robust results, we report Avg@32 (Pass@1 over 32 generations per question).

**GPQA.** GPQA is a Ph.D.-level STEM MCQA benchmark. We use the diamond subset, which includes the 198 hardest questions. We ask the model to put the letter of the final answer in a box, and judge the response by looking for an exact match with the ground truth answer. To obtain robust results we report Avg@4 (Pass@1 over 4 generations per question).

**LiveCodeBench.** LiveCodeBench (LCB) is a single-turn coding evaluation benchmark that collects new problems over time from popular programming contests. We use version v6 and include the 454 latest problems (August 2024 to May 2025) as reported in the official LiveCodeBench leaderboard at the time of writing. We copy the the verification logic from the official GitHub repository but integrate it with our sandboxes to ensure secure and scalable test verification. We report Avg@2 (Pass@1 with 2 rollouts per problem)

**MMLU-Pro.** MMLU-Pro [49] is a challenging subset of 12K general STEM MCQA questions from MMLU. We ask the model to put the letter of the final answer in a box, and grade by checking for an exact match with the ground truth answer. We report Avg@1 over the 12K samples.

**HLE.** Humanity's Last Exam (HLE) [36] consists of 2,500 questions across dozens of subjects, including mathematics, humanities, and the natural sciences. HLE is developed globally by subject-matter experts and consists of multiple-choice and short-answer questions suitable for automated grading. We use the text-only subset, which includes 2,158 examples, problems and do not give the models additional tools. We report the average solve rate across all samples.

## A.2 API models

**GLM 4.5 Air, GLM 4.5 & GLM 4.6** We evaluate `GLM-4.5-Air`, `GLM-4.5` [44] and `GLM 4.6` via OpenRouter and enforce that requests are routed to the official z-AI API. We adopt the sampling parameters setup recommended by z-AI. (e.g. we use temperature 0.6 across all benchmarks.)

**DeepSeek.** We evaluate `DeepSeek R1 0528` via OpenRouter and enforce that requests are routed to the official DeepSeek API. We did not manage to evaluate `DeepSeek v3.2 (Thinking)` via OpenRouter as most providers (including the official DeepSeek provider on OpenRouter) would host the chat version. For this reason, we fall back to using the official DeepSeek API with the `deepseek-reasoner` model slug.

**OpenAI.** We tried evaluating `GPT-OSS 120B (High)` [35] via OpenRouter but from the average response length it was clear that we were evaluating the model with lower than advertised reasoning efforts. We resorted to evaluating `GPT-OSS 120B (High)` via TogetherAI, where we confirmed that the hosted version is indeed using high reasoning effort. Our observed scores differ slightly from those in the OpenAI model card, and reflect an apples-to-apples comparison from running all models in the same evaluation harnesses, where formatting and scoring logic may differ slightly from other implementations.