INTELLECT-2: A Reasoning Model Trained Through Globally Decentralized Reinforcement Learning

Prime Intellect Team

Sami Jaghouar **Justus Mattern** **Jack Min Ong** **Jannik Straube**

Manveer Basra **Aaron Pazdera** **Matthew Di Ferrante** **Kushal Thaman**

Felix Gabriel **Fares Obeid** **Kemal Erdem** **Michael Keiblinger**

Johannes Hagemann

Abstract

We introduce INTELLECT-2, the first globally distributed reinforcement learning (RL) training run of a 32 billion parameter model. Unlike traditional centralized training efforts, INTELLECT-2 trains a reasoning language model using fully asynchronous RL across a dynamic, heterogeneous swarm of permissionless compute contributors.

To enable a training run with this unique infrastructure, we built various components from scratch: we introduce PRIME-RL, our training framework purpose-built for distributed asynchronous reinforcement learning, based on top of novel components such as TOPLOC, which verifies rollouts from untrusted inference workers, and SHARDCAST, which efficiently broadcasts policy weights from training nodes to inference workers.

Beyond infrastructure components, we propose modifications to the standard GRPO training recipe and data filtering techniques that were crucial to achieve training stability and ensure that our model successfully learned its training objective, thus improving upon QwQ-32B, the state of the art reasoning model in the 32B parameter range.

We open-source INTELLECT-2 along with all of our code and data, hoping to encourage and enable more open research in the field of decentralized training.

Contents

1 Introduction

Test-time compute scaling with reinforcement learning has emerged as a new scaling axis for large language models (LLMs), enabling improvements by allowing models to spend more time reasoning. The success of RL approaches used in models like DeepSeek-R1 [9], QwQ [40], and OpenAI's o1 [30] in learning reasoning capabilities highlight the power of test-time compute scaling.

However, reinforcement learning training is typically centralized, requiring large clusters of co-located GPUs and fast interconnect speeds. With INTELLECT-2, we showcase a paradigm shift: reinforcement learning is inherently more asynchronous and well suited for decentralized, globally distributed compute.

In this paper, we present the first large-scale experiment to collaboratively train a 32-billion-parameter language model using reinforcement learning across a permissionless, globally distributed network of contributors.

We open-source the INTELLECT-2 model, tasks and verifier environments at `huggingface.co/PrimeIntellect/INTELLECT-2` and the PRIME-RL framework for globally distributed RL training at `github.com/PrimeIntellect-ai/prime-rl`.

The remainder of this report is organized as follows: Section 2 provides a detailed overview of the decentralized training infrastructure including PRIME-RL, TOPLOC, SHARDCAST and the compute orchestration protocol. Section 3 describes the RL training recipe used to train INTELLECT-2. Section 4 presents the experiments we ran along with training performance metrics and model evaluation results. Section 5 discusses possible implications of decentralized training in the test-time compute paradigm. Finally, Section 6 concludes the report and outlines directions for future work.

2 Training Infrastructure

We introduce the following key open-source infrastructure components for training INTELLECT-2:

- **PRIME-RL**: A fully asynchronous reinforcement learning framework designed for decentralized training. The decoupling of rollout generation, model training, and weight broadcasting, enables training across heterogeneous, unreliable networks.
- **SHARDCAST:** A library for distributing large files via a HTTP-based tree-topology network that efficiently propagates updated model weights to the decentralized inference workers.
- **TOPLOC [29]:** A locality-sensitive hashing scheme for efficient verifiable inference. It detects tampering or precision changes in model inference and works reliably across non-deterministic GPU hardware.
- **Protocol Testnet:** Provides the infrastructure to aggregate and coordinate global compute resources.

As shown in Figure 1, the INTELLECT-2 infrastructure, built using these components, is structured around three primary roles: Inference rollout workers that generate reasoning traces using the current policy; TOPLOC validators that verify the integrity of these rollouts; and GRPO training workers that aggregate verified data, update the policy using the GRPO algorithm, and distribute new weights via SHARDCAST.

This decentralized RL training setup offers several key advantages:

- **No communication overhead:** By leveraging asynchronous reinforcement learning, the broadcast of new policy weights is fully overlapped with ongoing inference and training—eliminating the communication bottleneck.
- **Support for heterogeneous nodes:** Contributors can generate rollouts at their own pace using various hardware; there is no requirement for uniform speed across nodes.
- **Low resource requirements:** Inference workers, which constitute the majority of compute in this setup, can run on consumer-grade GPUs.
- **Efficient validation:** TOPLOC performs validation significantly faster than generation.

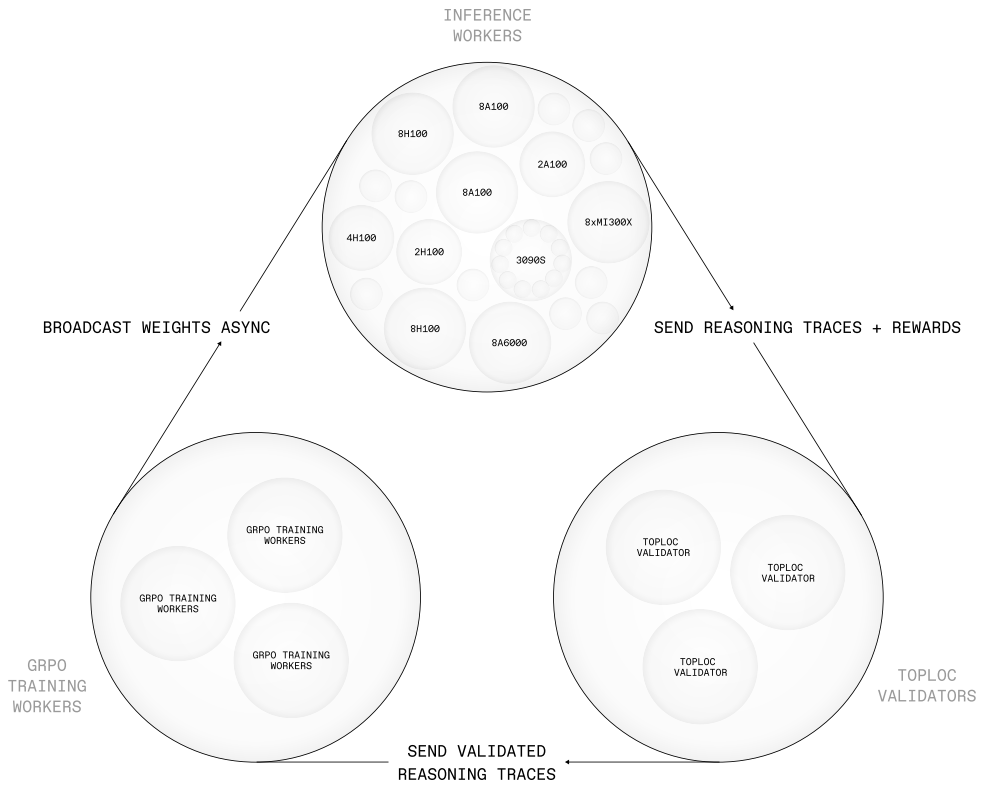


Figure 1: System Overview of INTELLECT-2 Distributed RL Training Infrastructure.

2.1 PRIME-RL: A Framework for Distributed Asynchronous Reinforcement Learning

We developed a new framework, PRIME-RL[1], to support training and inference workloads for reinforcement learning. Unlike existing frameworks such as verl [36] and TRL [41], which execute training and inference sequentially within the same process, PRIME-RL natively enables asynchronous execution of training and inference. This decoupling allows model updates to be computed on trusted centralized nodes, while rollouts are independently generated on trustless decentralized nodes.

PRIME-RL's architecture completely separates training and inference components into distinct executable files that communicate only when exchanging data and checkpoints. This clean separation eliminates the need for centralized orchestrators like Ray [27], and our two-step asynchronous design effectively hides latency that would typically be associated with data transfers, creating an efficient distributed reinforcement learning pipeline.

2.1.1 Training

To reduce GPU memory requirements during training, we shard the model weights, gradients, and optimizer states across GPUs using PyTorch FSDP2 [45], following a strategy similar to ZeRO-3 [31]. We load training data from remote storage, and rollout data is exchanged between inference workers and the trainer using Parquet files.

The asynchronous nature of rollout generation is transparent to the trainer, as we compute log-probabilities using the policy at the start of the optimization step rather than the policy that produced the original trajectories. This design choice aligns with the implementations in verl [36] which we used as a reference.

In its current form, PRIME-RL implements GRPO training, along with auxiliary KL and entropy losses.

[1] https://github.com/PrimeIntellect-ai/prime-rl

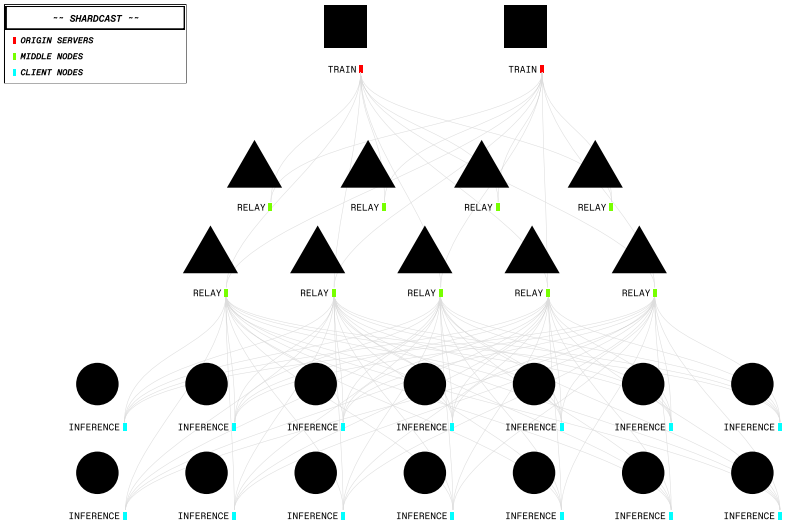


Figure 2: Overview of the Shardcast policy weight distribution network.

2.1.2 Inference

To generate the rollouts, we use vLLM [16], loading the model in bfloat16 precision. For each batch, input questions are sampled randomly using a deterministic seed to prevent inference workers from selecting easy samples, as detailed in Section 2.3.3.

To support TOPLOC proof construction, we capture the final hidden states using a hook in the logits processor. Instead of storing the activations for the full sequence, we incrementally store and hash them at every 32-token interval which reduces the memory overhead of the proof construction. To reduce blocking overhead, the proof construction is performed asynchronously on the CPU in parallel with the GPU forward pass. Together, these optimizations limit the overhead of proof generation to only a $\sim 1\%$ reduction in tokens-per-second throughput.

To keep the inference workers in sync with the rest of the training, we host a step counter endpoint which returns the smallest step with insufficient rollouts. Inference workers poll this endpoint and generate rollouts for the step specified. This design allows workers to dynamically join or leave the compute pool without interrupting the training process.

2.1.3 Verifiers

PRIME-RL uses the GENESYS schema introduced in SYNTHETIC-1 [23], making it easy to implement new reward environments. In our initial experiments, we support symbolic verifiers for mathematics and unit test execution for python-based coding competition problems. For this, we adopt existing implementations from [21] and [8].

Note that at this point, LLM-generated code is executed on the inference nodes, where we already apply sandboxing and code sanitization. This approach provides sufficient isolation for the simple algorithmic challenges currently used. For more complex coding tasks (e.g., those requiring filesystem access), further strengthening of isolation mechanisms will be necessary.

2.2 SHARDCAST: Efficient Policy Weight Broadcasts

One of the key challenges in asynchronous distributed reinforcement learning (RL) in a decentralized setting is ensuring that the most recent policy weights are quickly delivered to the inference workers.

For this purpose, we utilize a network of relay servers that distribute the checkpoints from the main training servers to the client inference workers, similar to a content delivery network (CDN). To minimize latency, checkpoint files are sharded and streamed in a pipelined fashion, allowing inference workers to begin downloading shards before the full checkpoint is available on the relay servers. The

relay servers only keep the last five checkpoint versions to avoid running out of disk. This limit is also functionally desirable, as rollouts generated with outdated checkpoints are typically rejected or discarded.

2.2.1 Rate Limiting & Firewall

We use nginx [2] as our HTTP server due to its robustness and widespread industry adoption. To protect the relay servers from malicious inference workers who may make an excessive number of requests, we configure our nginx server with per IP rate limiting. Additionally, we dynamically configure UFW firewall rules on the relay servers to accept traffic only from known, currently active inference nodes in the compute pool. This reduces the surface area for attacks and enables us to quickly blacklist misbehaving nodes when detected.

2.2.2 Maximizing Client Throughput & Load Balancing

If each client were to always select the fastest relay server, it would lead to contention and bandwidth thrashing. To mitigate this, clients instead sample from the set of relay servers based on the expected throughput of requesting from each relay server.

Our implementation begins by having each client request a dummy file from all relay servers to initialize bandwidth and success rate estimates. Clients then select servers in proportion to:

$$\text{expected throughput} \propto \text{success rate} \times \text{bandwidth}$$

These estimates are continuously updated using an exponential moving average (EMA), which smooths transient fluctuations while remaining responsive to actual changes. A healing factor is incorporated into the EMA to encourage periodic exploration of underutilized servers, ensuring the system adapts effectively to changing conditions.

Even in scenarios without contention, this probabilistic sampling strategy outperforms greedily choosing the currently fastest relay because it is able to utilize multiple connections to different relay servers, which will have a higher total bandwidth than any single connection to a relay.

2.2.3 Assembled Model Weights Integrity Checks

Inference nodes face the risk of being penalized if they submit rollouts generated using incorrect model weights. To avoid this, it is essential that only verified, correct weights are used for inference.

To ensure model weight integrity, each inference worker computes the SHA-256 checksum of the assembled checkpoint after downloading and reconstructing it from the shards. This checksum is then compared against the reference checksum produced by the training nodes, which is broadcasted to all relay servers along with the checkpoint metadata.

If the computed checksum does not match the expected value, the inference node discards the corrupted checkpoint and proceeds to attempt download of the next available checkpoint. We avoid retrying the same checkpoint, as it is unlikely the re-download would complete before the checkpoint becomes stale or irrelevant.

While a mechanism for verifying the integrity of downloaded shards would enable trustless peer-to-peer (P2P) weight transfers, we chose not to deploy such a system for INTELLECT-2. The primary reason is the added complexity and security risks associated with exposing inference workers to each another. In a P2P setup, inference worker IP addresses would become visible to peers, requiring additional hardening to prevent malicious behavior or denial-of-service attacks within the pool. Given these concerns, we opted to centralize weight distribution through trusted relay servers instead.

2.3 TOPLOC: Enabling Trustless Inference

Because we rely on trustless compute nodes for inference, there is no inherent guarantee that inference nodes perform the inference faithfully. To ensure verifiable compliance, our validators use several checks: computation, sampling and data sanity check which we will describe in depth in the following sections.

[2] https://nginx.org/

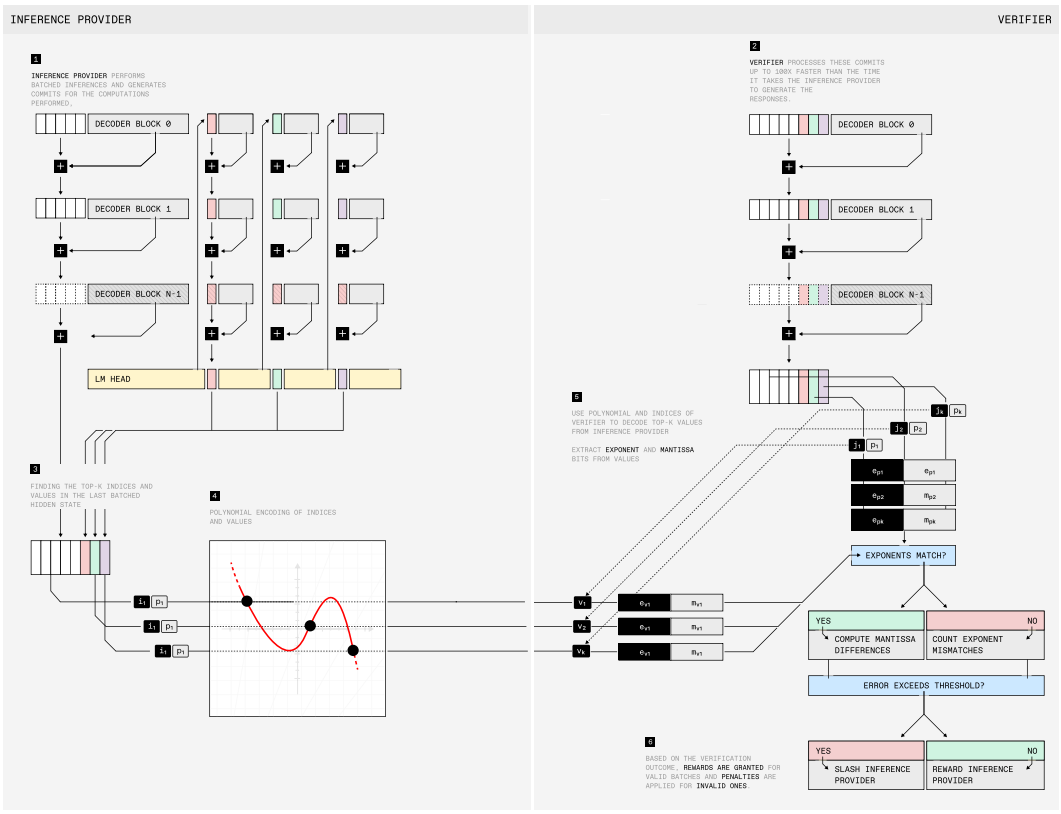


Figure 3: An illustration of TOPLOC. The Inference Provider performs batched inferences and generates commits for the computations performed, while the Verifier audits these commits up to $100\times$ faster than the time it takes the inference provider to generate the responses. Based on the verification outcome, rewards are granted for valid batches and penalties are applied for invalid ones. Further speedup can be obtained for the Verifier by not checking every batch but instead sampling randomly. Since the Inference Provider does not know which generations will be checked by the Verifier, they are incentivized to be honest on all generations to collect the reward and avoid receiving the penalty.

2.3.1 Computation checks

Proof of correct model computation As shown in Figure 3, to confirm that inference was performed using the correct model weights, each inference worker generates a TOPLOC proof [29] for every generated sequence. These proofs serve as cryptographic commitments to the final hidden states produced during decoding. A trusted validator node subsequently reconstructs these activations using prefill and compares them to the submitted commitments to confirm consistency. The proofs generated with TOPLOC are robust against GPU non-determinism, different tensor parallel configurations and are able to reliably detect when quantized or malicious versions of the model are used.

2.3.2 Sampling checks

Termination check There are two valid termination criteria for generated sequences: reaching the model's maximum context length or producing an end-of-sequence (EOS) token. Since longer sequences incur greater computational cost, inference providers may be incentivized to terminate sequences prematurely. To guard against this, we check that either the sequence reaches the maximum model length or ends with an EOS token. In the case that the sequence terminated on the generation of an EOS token, we make sure that the EOS token's probability exceeds 0.1 to prevent manipulation through unlikely EOS generations.

Token sampling check Proper sampling from logits should yield a distribution resembling an exponential with a mode at 1. If a smaller model is used to generate tokens and only the larger model is used for prefill (to pass TOPLOC checks), the resulting distribution becomes bimodal, with modes near 1 and 0. We inspect the logit distribution to detect such inconsistencies.

2.3.3 Sanity checks

Fixed data sampling Allowing inference workers to choose their own samples could lead to cherry-picking easy or previously completed examples. To prevent this, each node deterministically selects samples based on a seed computed as:

$$\text{seed} = \text{node address} \cdot \text{step} + \text{number of submissions for this step}$$

We then verify that the correct samples were used by reproducing the sampling process from the seed.

Value bounds check All reported scalar values—such as rewards and advantages—must fall within predefined bounds to ensure they are reasonable and consistent with expected outcomes.

Parquet formatting check We also make sure that the parquet has the correct schema and is in a format that is loadable by our training dataloader. This makes sure that we do not accept any files that would throw exceptions in the trainer.

2.4 The Prime Intellect Protocol

The Prime Intellect protocol coordinates permissionless nodes through a modular, decentralized orchestration layer. It gives model trainers the ability to check the health of all nodes, view logs, and distribute new tasks analogous to a decentralized SLURM[3]. The entire codebase is open source and available on GitHub[4].

2.4.1 System Architecture

As shown in Figure 4, the system is composed of multiple components that are all implemented in Rust. All components, except for the worker nodes and decentralized ledger, are hosted in a Kubernetes cluster. All API endpoints are protected by Cloudflare.

Decentralized Ledger A decentralized ledger is used to store information about the current training run, ownership of the training run, as well as worker contributions. These workloads are organized into "compute pools" that all belong to a broader "compute domain." Each compute domain represents a specific category of AI task—such as pre-training, synthetic data generation, distributed reinforcement learning, and more.

The ledger maintains detailed information about each pool, including ownership details and worker contributions. Each contributor, as well as the compute pool owner, has a cryptographic address used for signing transactions and proving ownership, which secures API interactions and ensures proper attribution of compute resources.

Worker Software The worker software's core task is to communicate heartbeats and metrics to the central orchestrator and to configure and manage the local Docker environment for task execution. Additional features include exports of logs and restart capabilities for running containers, a Unix socket-based connection between the Docker container itself and the worker. The latter can be used to trigger actions on the worker software, such as file uploads from the Docker volumes.

Discovery Service The discovery service is a simple API that allows nodes to upload worker metadata information. It stores this data in a Redis database and allows other authorized components, such as the orchestrator to retrieve information about nodes that have signed up. This ensures worker IPs are only visible to the orchestrator, reducing the risk of denial-of-service attacks.

[3]Simple Linux for Resource Management: `https://slurm.schedmd.com/`

[4]Prime Intellect Protocol: `https://github.com/primeIntellect-ai/protocol`

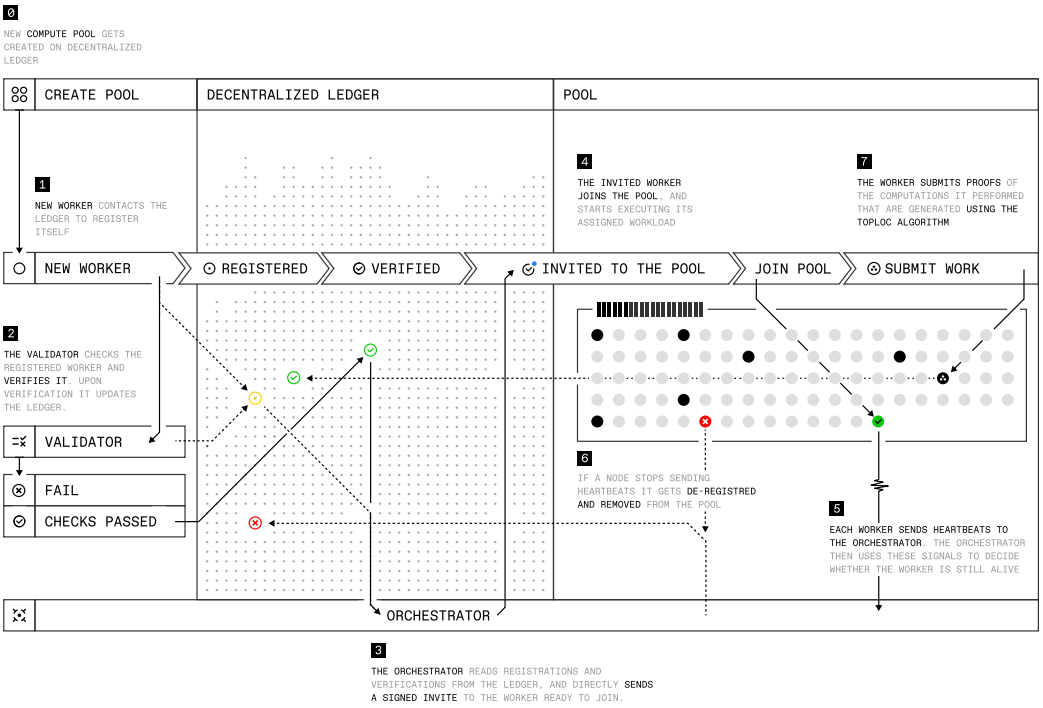


Figure 4: Overview of Protocol Testnet Infrastructure.

Orchestrator The orchestrator's core tasks include the distribution of tasks and observing the lifecycle of the decentralized worker nodes based on their heartbeat. It gives the model trainer the ability to interact with the infrastructure via web API. It thereby exposes information about all nodes that are currently alive, an API to create and schedule new tasks, and insights into the current metrics and logs of each node. Additionally, since containers on each node might fail, each node's workload can be restarted.

2.4.2 Operational Flows

Node Registration & Discovery The worker software is installed and started by compute contributors on their machines. It automatically detects the system components (GPU, available RAM and storage) and checks these for compatibility. Additional software and connection uplink checks are performed and inform the user in the logs about any misconfigurations or system issues that make the node incompatible with the training run.

Once the system hardware and software are confirmed, the node automatically uploads its metadata, including hardware information and IP, to a discovery service. In parallel, the node also sends a registration call to the decentralized ledger. After successful registration, the worker starts a webserver and waits for an invitation - this security measure ensures the worker doesn't need to know the orchestrator's endpoint in advance, protecting the orchestrator from potential denial-of-service attacks.

The orchestrator periodically checks the discovery service for newly created nodes and sends an invite to the worker's HTTP server to start contributing. This invite contains a cryptographic signature combining the node's address as well as the current compute pool's ID and domain. The invite is validated on the decentralized ledger and makes the worker an active compute contributor. After sending the invite, the orchestrator stores the node information in the local Redis storage and waits for incoming heartbeats.

Node Health & Heartbeats Each node maintains a continuous heartbeat loop to maintain communication with the orchestrator. These heartbeats act as simple signals sent from the node back to the orchestrator, allowing it to track whether nodes are still active. The orchestrator stores these heartbeats

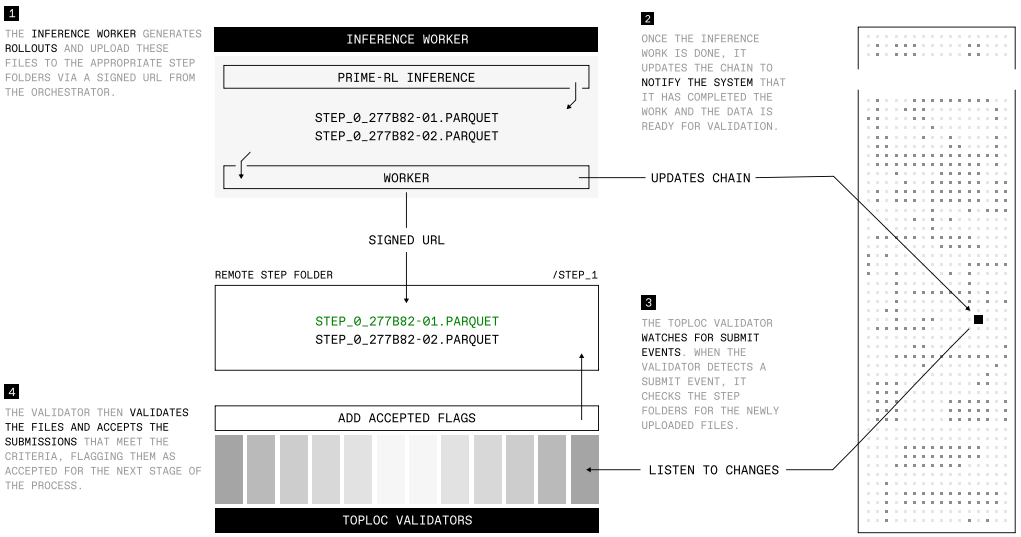


Figure 5: Overview of TOPLOC Validator Setup.

in Redis with an expiration time, so it can automatically detect when a node stops responding. A separate status update loop regularly checks the health of each node by counting missed heartbeats. If too many are missed, the node is marked as dead, and its worker is removed from the decentralized ledger. If the node comes back online, it tries to re-register and update the discovery service so it can be invited back into the pool.

Task Scheduling & Execution Tasks are created by the orchestrator through a POST API and scheduled asynchronously across all healthy nodes. Rather than pushing tasks, the orchestrator distributes them in response to heartbeat requests from the nodes, allowing for a reactive and fault-tolerant pull-based model. Once a task is received, the worker communicates with the Docker daemon to translate the task specification into a running container — this includes setting up volumes, managing the container lifecycle, and applying task-specific settings such as environment variables and custom start commands. One key insight during development was the introduction of a shared volume, used to store persistent data like model weights. Without this, restarting a task would trigger redundant downloads, slowing execution and increasing resource usage.

Inference Validation As shown in Figure 5, the inference workers will generate the rollouts and upload them to the remote folder using a signed URL. An on-chain event is then generated, which triggers the validators to begin validation for the new file. Based on the result of the checks described in Section 2.3, the file is either accepted or rejected. The accepted files are then read by the training node data loaders. Rejected files cause the node which created to them to be slashed and evicted from the compute pool.

2.4.3 Design Trade-offs & Limitations

The orchestrator and discovery service are currently centralized, which simplifies coordination but creates potential single points of failure and limits horizontal scalability. This design also introduces trust assumptions that may not be suitable for more distributed or permissionless environments. To address this, we plan to move toward a fully peer-to-peer architecture using a distributed hash table (DHT), which would eliminate the need for central coordination and enable more resilient, decentralized node discovery.

Another key limitation lies in how the worker is deployed. At present, it only runs on bare-metal machines or virtual machines with direct access to the Docker daemon. This excludes environments like Kubernetes, where Docker access is abstracted or unavailable. To address this, we are developing a version of the worker that can run as a container itself, making it compatible with container orchestration platforms.

3 Training Recipe

The goal of INTELLECT-2 is to train a model with reasoning capabilities, specifically in the domains of mathematics and coding. Additionally, we aim to enable control over the model's thinking budget by allowing users to specify the desired number of thinking tokens as part of the task prompt. As our base model, we use QwQ-32B [40] and largely follow Deepseek-R1's [9] approach of GRPO-based training with verifiable rewards.

In this section, we describe our RL recipe and the ablation experiments that led to it in detail, ranging from our training data and reward function implementations to modifications to the original GRPO objective to improve training stability.

3.1 Training Data & Rewards

We train INTELLECT-2 using a dual objective: we incorporate both task rewards encouraging the model to improve its reasoning on mathematics and coding tasks, as well as length rewards in order to teach the model to adhere to a thinking budget provided in the prompt.

3.1.1 Task Rewards

Following existing state-of-the-art open reasoning models [9, 40] , we curate a training dataset consisting of mathematics and coding tasks that can be verified through symbolic verification / string matching and unit test execution. To do so, we choose high quality problems from NuminaMath-1.5 [18] and Deepscaler [24] for math problems and coding tasks previously curated for SYNTHETIC-1 [22], which were also used in prior work such as DeepCoder [20]. Our full dataset consists of 285k tasks, including 26k python-based algorithmic coding challenges and 259k mathematics problems. The dataset can be found on Huggingface [5].

Both our mathematics and code reward functions implement binary rewards, with a reward of 1 being assigned for correct responses and 0 for incorrect responses. While this is an obvious choice for mathematics tasks, we explicitly don't assign partial rewards for passing some, but not all unit tests of coding problems to discourage reward hacking (e.g. through memorizing public test cases).

3.1.2 Length Rewards

Beyond rewards for solving tasks correctly, we incorporate length rewards to enable users to specify the thinking budget of INTELLECT-2 as part of the task prompt; hereby, we largely follow the methodology of L1 [1].

Concretely, for every problem in a training batch, we sample a target length l_{target} and include it in our prompt via the template "Think for l_{target} tokens before giving a response." - subsequently, a length penalty representing the difference between the actual response length and the target length, multiplied by a weighting factor α, is combined with the task reward. Letting y denote our model output for a given prompt, l_y its length in tokens, and r_{task} the task reward function, the total reward can be computed as:

$$r_{\text{total}}(y, l_{\text{target}}) = r_{\text{task}}(y) - \alpha * | \, l_{\text{target}} - l_y \, |$$

Different from the setting of L1, where l_{target} is sampled uniformly from a continuous range, we sample from a small discrete set of target lengths (e.g., 2000, 4000, 6000 tokens) to simplify the objective and make it easier for our model to learn. To validate this approach, we reproduced L1 using target lengths of 500, 1000, 2000 and 3000 with a maximum sequence length of 4000.

3.2 Asynchronous Reinforcement Learning

As discussed in Section 2, we use asynchronous reinforcement learning to use both dedicated inference and training nodes to minimize GPU idle time. This approach has proven to be effective in prior work [13, 28] and has also been adopted for large LLM training runs such as Tülu 3 [17] and Llama 4 [38].

[5] https://huggingface.co/datasets/PrimeIntellect/Intellect-2-RL-Dataset

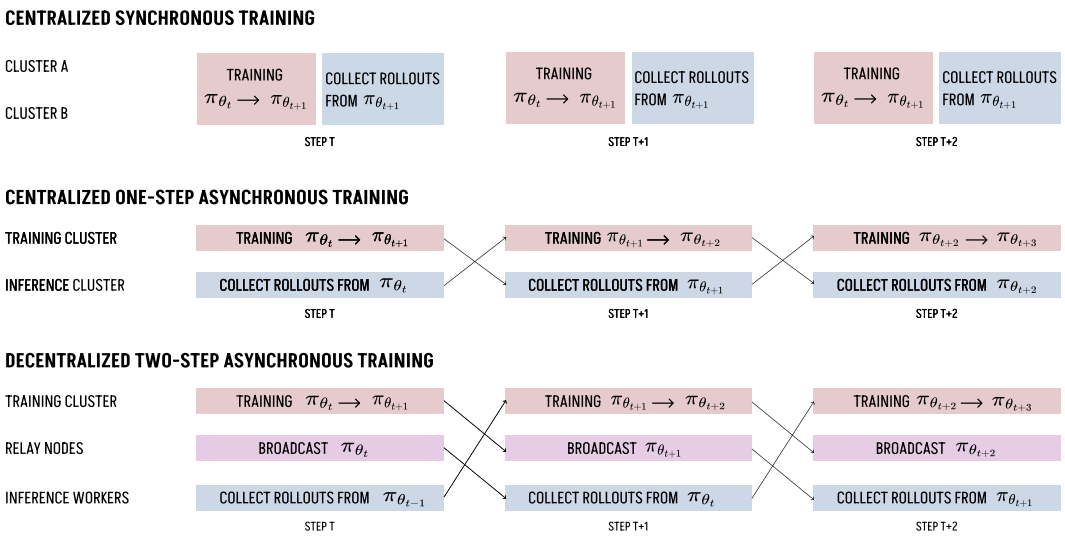


Figure 6: A comparison of synchronous, centralized one-step asynchronous and decentralized two-step asynchronous reinforcement learning: **Synchronous RL** leverages the same compute resources for training and inference and sequentially switches between performing only inference and training. Therefore, training is fully on-policy. **Centralized One-Step Asynchronous RL** has dedicated compute resources for training and inference and performs training and inference at the same time. Therefore, rollouts are collected from the policy of the last RL step, making training off-policy by one step. **Decentralized Two-Step Asynchronous RL** works similarly to centralized asynchronous RL, but inference workers don't have access to the up-to-date policy weights immediately after a training step due to the time-consuming weight broadcast. Therefore, rollouts are collected from policy weights from two or more RL steps prior.

In a centralized asynchronous RL training setup with fast connection speeds, the same policy weights that are updated on the dedicated training nodes are simultaneously used to obtain rollouts for training during the next RL step. In a decentralized setup, the updated policy weights are not available immediately to the inference workers, as the weight broadcast costs time, which is why we perform rollouts using weights not from the previous step, but from two or more steps prior, depending on the duration of the weight broadcast. A graphical overview of these differences along with a comparison to synchronous RL can be found in Figure 6.

Prior to starting our INTELLECT-2 training run, we ran ablation experiments to validate that asynchronous RL training does not hurt the performance of our model. To do so, we replicated the results of Deepscaler's synchronous RL training run of DeepSeek-R1-Distill-Qwen-1.5B using a context length of 2048 tokens and compared it with asynchronous RL based on PRIME-RLwith varying levels of asynchrony. The results of these runs can be found in Figure 7.

As seen in the graph, even with asynchrony levels of up to four, our model's reward trajectory matches the trajectory of the synchronous baseline, indicating that training on slightly off-policy data does not hurt the performance of RL training.

3.3 Offline & Online Data Filtering

During our ablation experiments, we found that filtering our dataset for difficulty had a significant impact on training performance. We employ both offline filtering before starting our training run and online filtering to selectively choose training samples from our rollouts.

3.3.1 Offline Data Filtering

When trying to train DeepSeek-R1-Distill-Qwen-7B using the Deepscaler mathematics dataset [21], we found that it was highly important to filter out problems that were too easy or too difficult from our training set. As shown in Figure 8, when training on the original dataset, our rewards barely improved. After filtering out problems in which the base model's pass@8 rate was above 50%, and below 12.5%,

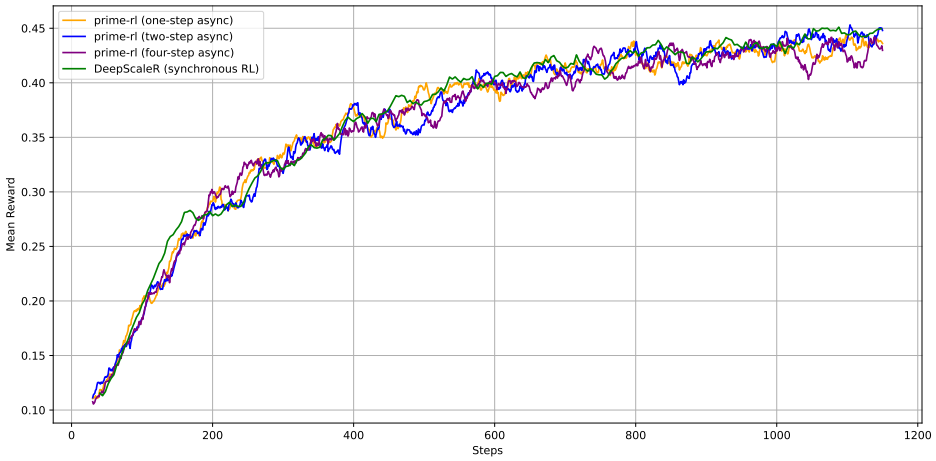
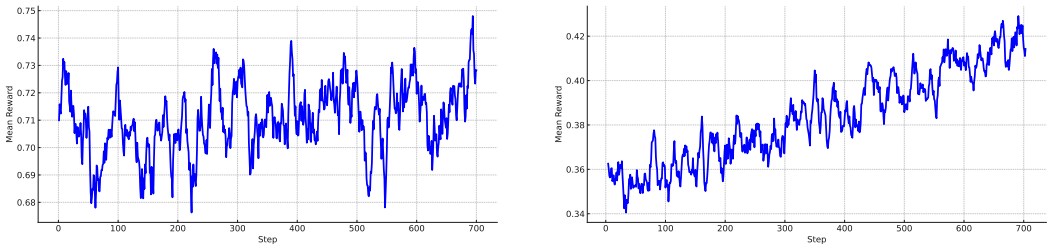


Figure 7: Comparison of synchronous DeepScaler [24] training vs asynchronous PRIME-RL under varying asynchrony levels. Even with increased delay (up to four steps), PRIME-RL matches the performance of synchronous baselines.



(a) 7B GRPO Training without Offline Filtering

(b) 7B GRPO Training with Offline Filtering

Figure 8: Reward trajectories when training DeepSeek-R1-Distill-Qwen-7B using GRPO on the Deepscaler Math dataset with and without online filtering. 8a shows the reward trajectory when training without filtering, leading to stagnant rewards. 8b shows the results of training on a filtered dataset only containing samples on which the base model had a pass@8 value between 1 and 4.

rewards improved significantly. As a result, we also used DeepSeek-R1-Distill-Qwen-7B to prefilter our training dataset for INTELLECT-2.

3.3.2 Online Data Filtering

Training algorithms such as GRPO [35] and RLOO [15] rely on group-based relative rewards to compute advantages. If all completions for a single problem receive the same rewards (for binary rewards either 0 or 1), this means that the advantages for all of these samples are zero and no training signal is given beyond auxiliary losses such as a KL or entropy loss. To mitigate this, we employ online filtering and continue sampling responses from our inference workers until we have a full batch of samples with non-zero advantages before performing a training step. Conveniently, this increases the amount of inference that has to be performed per training step, allowing us to onboard and leverage a higher amount of decentralized inference nodes.

3.4 Two-Sided GRPO Clipping for Increased Training Stability

During training, we faced loss and resulting gradient norm spikes that caused instabilities leading to model collapse, particularly as our models got larger. Upon inspection, we found that a major cause of instabilities was one-sided token probability ratio clipping employed in GRPO and PPO-like [34] training objectives.

Recall the original GRPO objective: for each prompt or question q, GRPO samples a group of outputs $\{o_1, o_2, ..., o_G\}$ from the old policy $\pi_{\theta_{\text{old}}}$ and computes advantages based on their relative rewards based in this group. The optimization objective without the KL penalty is given by:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\ \{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(O|q)}$$

$$\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left[\min\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}\hat{A}_{i,t},\ \text{clip}\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}, 1-\varepsilon,\ 1+\varepsilon\right)\hat{A}_{i,t}\right)\right]$$

where $\hat{A}_{i,t}$ denotes the advantage within each sampled group and ε is a parameter used to clip the token probability ratio to avoid excessively large loss values and gradient updates. Note that in the case of negative advantage values, this clipping will not be applied due to the $\min$ operation, as large updates that move the policy away from bad rollouts are encouraged [34]. However, this can cause huge loss values and gradient updates as a result in case $\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}$ takes on large values.

To mitigate this, we introduce an additional hyperparameter δ that adds an upper bound to the token probability ratio in the case of negative advantages:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\ \{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(O|q)}$$

$$\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left[\min\left(\min\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}, \delta\right)\hat{A}_{i,t},\ \text{clip}\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}, 1-\varepsilon,\ 1+\varepsilon\right)\hat{A}_{i,t}\right)\right]$$

The value δ should be higher than $1+\varepsilon$ to still enable large updates that move away from bad rollouts, but avoid huge token probability ratios of a hundred or much higher. With this change, training stabilized significantly, as it has also been reported in concurrent work [25].

3.5 Mitigating Training Instability at Scale

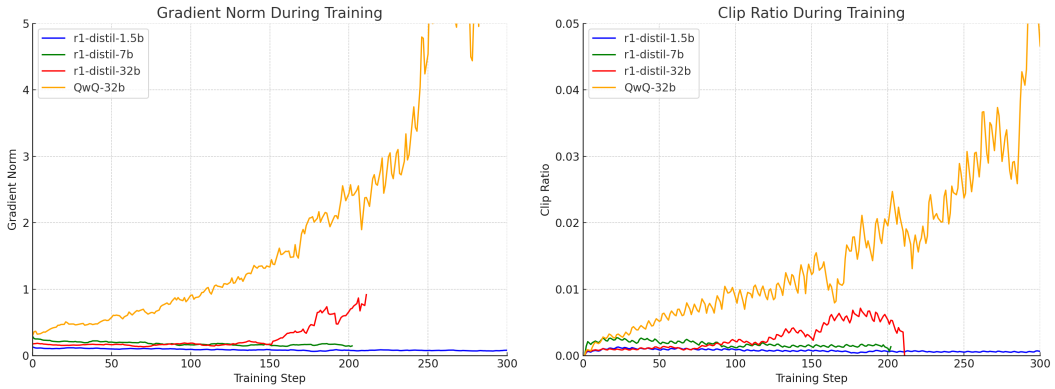
While the two-sided GRPO clipping mechanism described above significantly reduced large loss and gradient spikes, we observed additional types of training instabilities when using larger models. These instabilities share similarities with those encountered in large-scale pretraining [43, 5, 26, 6].

Escalating Gradient Norms As training progressed, we observed a gradual but persistent increase in gradient norms, even in the absence of immediate spikes. This phenomenon appears to be correlated with model size as shown in 9a, becoming more pronounced in our larger architectures. Similar to findings in [43], we found that gradient norm growth often precedes more severe instability events, serving as an early warning signal for potential training collapse.

We found that employing aggressive gradient clipping (with thresholds as low as 0.05-0.1) effectively mitigates stability issues without significantly impeding convergence, providing a favorable trade-off between stability and training efficiency. While this approach does not completely eliminate instability issues, it substantially delays the growing gradient phase and postpones potential stability crashes, extending the viable training period for our models. Aggressive gradient clipping has also been applied successfully in concurrent work training QwQ-32B [2].

Token Probability Clip Ratio Escalation. In addition to gradient norm and entropy instabilities, we observed a steady increase in the token probability clip ratio during training, as shown in Figure 9b. This increase directly correlates with the rising gradient norm, as the clip ratio effectively tracks the difference in logits between consecutive optimizer steps.

Entropy Loss pattern. During training, we identified a distinctive pattern for the entropy loss, as shown in Figure 10. After decreasing initially, the entropy loss begins to trend upward again. This entropy resurgence typically precedes catastrophic training failures causing a full collapse of the model. Increasing the weighting factor of the KL penalty was able to delay this collapse but also caused slower learning, and hence wasn't an effective mitigation strategy.



(a) Gradient norms during training across training steps (b) Token probability clipping ratio across training steps

Figure 9: Escalating gradient norms (Figure 9a) and clipping ratios (Figure 9b) across model scales, trained on the MATH dataset [12]. Smaller models remain stable, while both 32B models show rising instability, with QwQ-32B diverging earlier than R1-Distill-32b.

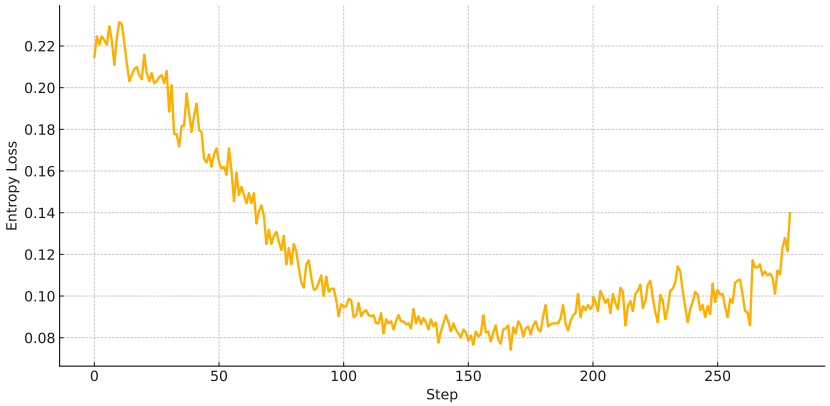


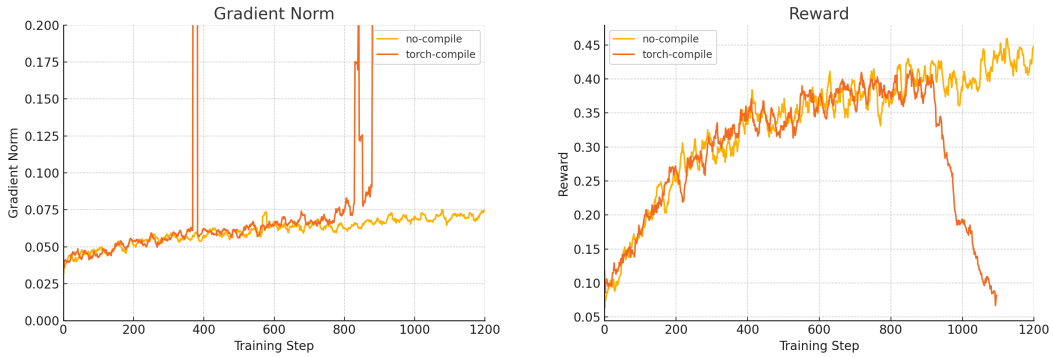
Figure 10: As training progresses, our policy's entropy loss initially decreases but later starts increasing past ≈ 150 steps. Soon after seeing entropy increases, we observed our model collapsing across all of our ablation runs.

QwQ is less stable to train than DeepSeek-R1-Distill-Qwen-32B. We noticed that our training on top of QwQ exhibited worse stability compared to DeepSeek-R1-Distill-Qwen-32B, despite both being based on the same pre-trained model (Qwen 2.5). We hypothesize that this difference stems from QwQ having already undergone a phase of reinforcement learning with verifiable rewards. This prior RL training appears to make the model more susceptible to subsequent optimization instabilities, suggesting that models may become progressively more difficult to fine-tune stably after multiple rounds of reward optimization.

Instabilities caused by torch.compile. We observed that using `torch.compile` led to a catastrophic collapse during the later stages of our training, regardless of our model size (see Figure 11). Although the issue likely stemmed from a single faulty kernel generated by `torch.compile`, we ultimately decided to disable it across the entire codebase for the run to ensure training stability. This decision came with a trade-off of slightly increased memory usage.

4 Experiments

Over the duration of two weeks, we ran multiple training runs using our setup consisting of a trusted training cluster and validator nodes and trustless, community-contributed heterogenous inference workers. In this section, we report the experiments we ran along with corresponding results.



(a) Gradient Norm

(b) Reward

Figure 11: Training dynamics with and without `torch.compile` when training DeepSeek-R1-Distill-Qwen-1.5B on the Deepscaler mathematics dataset. The *torch-compile* setup leads to early instability and reward collapse, whereas the *no-compile* baseline remains stable across 1200 steps.

4.1 Experimental Setup

Using QwQ-32B as our base model, we trained using GRPO with modifications described in section 3 and used clipping thresholds $\varepsilon = 0.2$, $\delta = 4$ and an entropy loss coefficient of 1e-4. We set the KL divergence loss coefficient to 0.001, set α to 0.0003 to balance task and length rewards, and apply gradient norm clipping at 0.1. Additionally, we implement a token-level policy loss calculation rather than a sample-level loss calculation as proposed in DAPO [44] and Dr. GRPO [19]. The training used a learning rate of 3e-7 with 25 warmup steps; during every rollout step, we generated 4096 samples consisting of 16 responses to 256 prompts, and performed 8 optimizer steps using a batch size of 512. We used two-step asynchrony during training to enhance throughput while ensuring that we did not go too far off-policy.

We used the Huggingface implementation of Qwen with Flash Attention 2. Instead of obtaining token log probabilities from our inference workers, we computed them seperately using our policy weights on the training cluster, as vLLM log probabilities turned out to not be numerically stable. The model was sharded using FSDP2 with activation recomputation enabled. We configured a maximum sequence length of 32K to accommodate longer inputs required by our application.

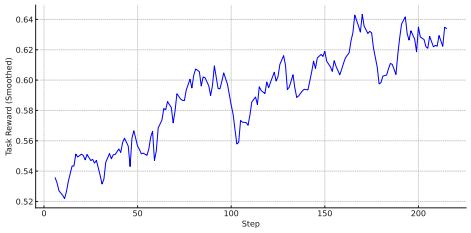
Sequence Packing To maximize computational efficiency with our 32K sequence length, we implemented sequence packing to address the significant variance in sample lengths. This approach prevents wasting compute on padding tokens, which would be particularly inefficient given our distribution of sequence lengths during inference rollout. Unlike pretraining scenarios, where arbitrarily cutting samples is acceptable due to the local nature of the next token prediction loss function, reinforcement learning requires preserving complete samples since RL fundamentally learns at the sample level rather than locally. While RL fundamentally requires preserving complete samples, GRPO's token-level loss formulation allowed us to implement cross-sample packing by adapting the attention mask and collating samples into the sequence dimension. This optimization proved essential for scaling beyond 20K+ sequence lengths and significantly reduced our training time while maintaining the integrity of the cross entropy calculations across packed sequences.

4.2 Results

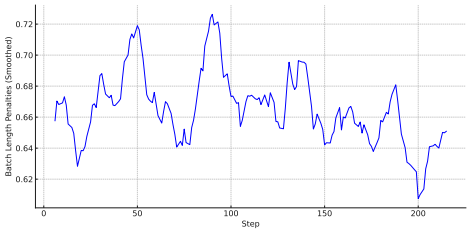
We report results from two main experiments: TARGET-SHORT, an experimental run with target lengths $\{1000, 2000, 3000, 4000\}$ to train an efficient reasoning model, and, TARGET-LONG, our main run with longer target lengths of $\{2000, 4000, 6000, 8000, 10000\}$.

Compute Utilization During the two main experiments, we successfully overlapped communication with computation through asynchronous reinforcement learning.

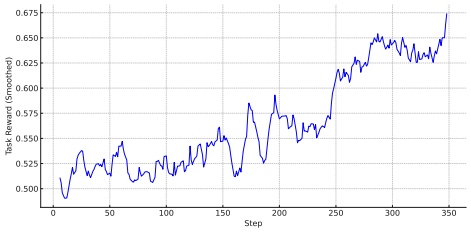
In both experimental settings, the SHARDCAST broadcast to all nodes averaged 14 minutes, corresponding to a bandwidth throughput of approximately 590 Mb/s (62 GB of weights transmitted over



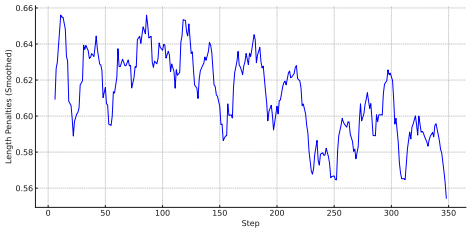
(a) Task Rewards for TARGET-SHORT



(b) Length penalties for TARGET-SHORT



(c) Task Rewards for TARGET-LONG



(d) Length penalties for TARGET-LONG

Figure 12: Trajectories (smoothed by 10-step moving average) of task rewards and length penalties for TARGET-SHORT, our training run with target lengths $\{1000, 2000, 3000, 4000\}$ and TARGET-LONG, our training run with target lengths $\{2000, 4000, 6000, 8000, 10000\}$. During both runs, our task rewards, indicating the ability to solve mathematics and coding problems, rose significantly. In contrast to our ablation experiments with smaller models, length penalties decreased significantly slower; while a clear downward trend is recognizable, the training runs were too short for the length penalties to converge and equip the model with the ability to precisely adhere to a thinking budget.

14 minutes). Nodes with superior connectivity to the SHARDCAST relay servers received checkpoints earlier, allowing them to begin data generation ahead of others. Furthermore, nodes with more computational resources, such as full H100 nodes, generated batches more quickly, resulting in earlier validation by the TOPLOC validators.

In the TARGET-SHORT setup, the first data file was submitted approximately 10 minutes after the broadcast completed. Thanks to the prefill verification mechanism in TOPLOC and the random sampling strategy that verifies only subsets of submitted data, the inference verification was highly efficient—typically completing within 1 minute. Consequently, sufficient verified samples to form a batch were available roughly 22 minutes after the broadcast in the TARGET-SHORT scenario. In contrast, the TARGET-LONG scenario required approximately 29 minutes to accumulate enough verified samples to form a batch.

The ratio of training to inference FLOPs in both experiments averaged $4.5\times$, with significantly more compute spent on the decentralized inference workers than on the training side.

In TARGET-SHORT, training nodes took approximately 22 minutes to execute a full rollout step. This duration resulted in minimal idling of training GPUs, as the data generated by inference nodes for the next step was already available. Conversely, in TARGET-LONG, training nodes completed their rollout steps in about 21 minutes. The asynchronous setup effectively synchronized with the broadcast, inference generation, and verification phases, ensuring nearly perfect computational overlap and minimizing GPU idling time.

These results highlight the inherent advantage of decentralized reinforcement learning, especially when scaling inference computations—such as generating longer reasoning chains—to achieve an optimal inference-to-training compute ratio. Additional analysis on this scaling benefit is discussed in section 5.

Reward Trajectories Throughout training, we saw significant improvements of our task rewards, indicating that the model improved its performance on our mathematics and coding problems. We also saw a reduction of length penalties, but a much slower one than during our ablation experiments

Model	AIME24	AIME25	LiveCodeBench (v5)	GPQA-Diamond	IFEval
INTELLECT-2	**78.8**	64.9	**67.8**	66.8	81.5
QwQ-32B	76.6	64.8	66.1	66.3	83.4
Qwen-R1-Distill-32B	69.9	58.4	55.1	65.2	72.0
Deepseek-R1	78.6	65.1	64.1	71.6	82.7

Table 1: Performance comparison of models across benchmarks.

with 1.5B and 7B parameter models. As a result, our models did not learn to strictly adhere to the specified thinking budget within the given timeframe of our experiments. Further exploration will be needed to determine what exactly the cause of this is and whether a stronger weighting of the length rewards could have helped learn the length following objective. The full reward trajectories can be found in Figure 12

Benchmark Performance We use evalchemy[6] [11] along with its default settings to test the performance of our model on common reasoning benchmarks. We use the provided default prompts for Deepseek-R1, QwQ-32B and DeepSeek-R1-Distill-Qwen-32B and attach the length control prompt "Think for l_{target} tokens before giving a response" to INTELLECT-2. Since the length penalty was not affected significantly during training, we only evaluate our model with the longest target length of 10,000.

As can be seen in Table 1, we were able to increase the performance of QwQ-32B on mathematics and coding benchmarks, while seeing a slight drop on IFEval, which is likely caused by us solely training on mathematics and coding tasks rather than using more general instruction-following tasks. Overall, as QwQ-32B was already extensively trained with reinforcement learning, it was difficult to obtain huge amounts of generalized improvement on benchmarks beyond our improvements on the training dataset. To see stronger improvements, it is likely that better base models such as the now available Qwen3 [39], or higher quality datasets and RL environments are needed.

5 Discussion: Decentralized Training in the Test-Time-Compute Paradigm

As the compute demands of large language models have increased by several orders of magnitude in recent years, distributed training across data centers has become increasingly relevant. Beyond offering an economically sustainable path for collaborative open-source development, the sheer compute power and energy required to train these models will soon outpace even the largest data centers in the world.

So far, most progress has come from scaling parameters and dataset size—commonly referred to as pretraining scaling. More recently, a complementary axis of progress has emerged: test-time compute scaling, as seen in reasoning-focused models.

While both forms of scaling are compatible with decentralization, we argue that test-time compute scaling is particularly well-suited for decentralized training. It reduces coordination requirements and shifts compute demands toward inference, enabling broader participation from heterogeneous devices.

Asynchronous RL Hides Most Communication Overhead Communication is the primary bottleneck in decentralized training. Techniques such as DiLoCo [10] can reduce pre-training communication overhead by up to two orders of magnitude. However, as model sizes increase, communication—especially blocking communication—once again becomes the limiting factor.

A promising strategy is to overlap communication with computation. Unlike approaches such as ZeRO-offload [33], which delay gradient application and impact convergence, we argue that delaying rollouts in RL yields a better tradeoff. This is because the delay applies at the model level, not the optimization step. Even if the model is slightly off-policy, it can still generate useful reasoning traces that lead to positive rewards, which are valid training signals.

[6] https://github.com/mlfoundations/Evalchemy

Further investigation is needed to evaluate asynchronous RL with delays beyond two steps. Nonetheless, with delays of 4–5 steps, we could effectively hide various blocking stages in the RL pipeline—including weight broadcasting, environment verification, permissionless validation, and relative KL log-probability computation. This strategy improves compute utilization across both training and inference and enables greater hardware heterogeneity. Slower devices can still contribute valuable samples. Additionally, decentralized pipeline-parallel inference can facilitate the use of large models on consumer-grade hardware.

Inference Will Consume the Majority of Compute In INTELLECT-2, the training-to-inference compute ratio was approximately 1:4. We anticipate this ratio will shift even more heavily toward inference as test-time reasoning scales. This trend opens the door to training models with hundreds of billions of parameters on globally distributed heterogeneous compute resources.

A key driver of this shift is dataset filtering. As illustrated in Figure 8, model capabilities improve when training focuses on more challenging samples. However, not all data generated during inference is useful. As models tackle harder tasks with increasingly sparse positive rewards, inference will demand substantially more compute than training. In this setting, generating high-quality rollouts becomes the dominant compute cost. Since only a small subset of these rollouts contains strong learning signals, the majority of compute is allocated to exploration rather than model updates.

This asymmetry in compute demand reshapes the scaling dynamics of decentralized RL and indirectly addresses one of its historical limitations: memory constraints. By shifting most of the workload to inference—where memory requirements are significantly lower than during training—decentralized training becomes feasible at scale across a broader range of hardware.

6 Conclusion & Future Work

In this report, we introduce INTELLECT-2, the first globally distributed reinforcement learning run of a 32-billion-parameter language model. We are open-sourcing the trained model, tasks & verifier environments along with all infrastructure components including our training framework PRIME-RL. We hope that this report and the accompanying open-source components will support the broader research community in exploring decentralized training, and help advance globally distributed reinforcement learning as a foundation for building frontier open-source models.

While INTELLECT-2 is a first step towards open frontier reasoning models trained in a decentralized fashion, several avenues for future work remain open:

Increasing the Ratio of Inference to Training Compute in Reinforcement Learning As inference is infinitely parallelizable and does not require any communication between workers, RL training recipes that spend higher amounts of compute on inference relative to training are ideally suited for decentralized training. Methods such as VinePPO [14] spend additional time on inference to compute Monte Carlo-based value estimates rather than leveraging a value network such as PPO, and are thus an interesting field of study to explore. Additionally, various forms of online data filtering for curriculum learning approaches are purely based on inference, and are thus favorable for decentralized setups, if proven effective.

Tool Calls for Reasoning Models The latest generation of proprietary reasoning models have access to tool calls such as web search or python interpreters as part of their reasoning chain. Initial promising research results in this direction have come from open source research efforts [3, 4, 42], opening the door to scaling these methods further and training larger open-source reasoning models capable of leveraging such tools.

Crowdsourcing RL tasks and environments To teach models new skills, diverse RL environments have to be built. This boils down to a traditional software engineering problem which is highly parallelizable and requires various contributors with specialized areas of domain expertise, making it ideally suited for open-source, community-driven efforts. We invite everyone to contribute RL environments to PRIME-RL and are aiming to make it as easy as possible to crowdsource reinforcement learning environments.

Model Merging and DiLoCo Model merging has emerged as an effective post-training technique in recent work [7, 37, 32]. Whether such methods extend to reasoning tasks remains an open question. However, the ability to merge models trained on distinct reasoning domains would mark a significant step toward scaling asynchronous reinforcement learning across parallel compute resources. In this setup, multiple models could be trained independently and later merged into a single unified model. This could be done at the end of training or continuously during training using techniques like DiLoCo [10], originally developed to reduce communication in data-parallel pretraining. Applying merging in RL would enable scaling decentralized training to one more order of magnitude more compute.

Acknowledgements

We would like to thank all the compute contributors for this training run. This includes Demeter Compute, string, @BioProtocol, @mev_pete, @plaintext_cap, @skre_0, @oldmankotaro, plabs, @ibuyrugs, @0xfr_, @marloXBT, @herb0x_, mo, @toptickcrypto, cannopo, @samsja19, @jackminong and primeprimeint1234.

We would also like to thank Michael Luo for his advice on replicating the DeepScaler results.

References

[1] Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning, 2025.

[2] Carlo Baronio, Pietro Marsella, Ben Pan, and Silas Alberti. Multi-turn training for cuda kernel generation. https://cognition.ai/blog/kevin-32b.

[3] William Brown. Verifiers: Reinforcement learning with llms in verifiable environments. 2025.

[4] Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[6] Jeremy M. Cohen, Behrooz Ghorbani, Shankar Krishnan, Naman Agarwal, Sourabh Medapati, Michal Badura, Daniel Suo, David Cardoze, Zachary Nado, George E. Dahl, and Justin Gilmer. Adaptive gradient methods at the edge of stability, 2024.

[7] Team Cohere, :, Aakanksha, Arash Ahmadian, Marwan Ahmed, Jay Alammar, Milad Alizadeh, Yazeed Alnumay, Sophia Althammer, Arkady Arkhangorodsky, Viraat Aryabumi, Dennis Aumiller, Raphaël Avalos, Zahara Aviv, Sammie Bae, Saurabh Baji, Alexandre Barbet, Max Bartolo, Björn Bebensee, Neeral Beladia, Walter Beller-Morales, Alexandre Bérard, Andrew Berneshawi, Anna Bialas, Phil Blunsom, Matt Bobkin, Adi Bongale, Sam Braun, Maxime Brunet, Samuel Cahyawijaya, David Cairuz, Jon Ander Campos, Cassie Cao, Kris Cao, Roman Castagné, Julián Cendrero, Leila Chan Currie, Yash Chandak, Diane Chang, Giannis Chatziveroglou, Hongyu Chen, Claire Cheng, Alexis Chevalier, Justin T. Chiu, Eugene Cho, Eugene Choi, Eujeong Choi, Tim Chung, Volkan Cirik, Ana Cismaru, Pierre Clavier, Henry

Conklin, Lucas Crawhall-Stein, Devon Crouse, Andres Felipe Cruz-Salinas, Ben Cyrus, Daniel D'souza, Hugo Dalla-Torre, John Dang, William Darling, Omar Darwiche Domingues, Saurabh Dash, Antoine Debugne, Théo Dehaze, Shaan Desai, Joan Devassy, Rishit Dholakia, Kyle Duffy, Ali Edalati, Ace Eldeib, Abdullah Elkady, Sarah Elsharkawy, Irem Ergün, Beyza Ermis, Marzieh Fadaee, Boyu Fan, Lucas Fayoux, Yannis Flet-Berliac, Nick Frosst, Matthias Gallé, Wojciech Galuba, Utsav Garg, Matthieu Geist, Mohammad Gheshlaghi Azar, Ellen Gilsenan-McMahon, Seraphina Goldfarb-Tarrant, Tomas Goldsack, Aidan Gomez, Victor Machado Gonzaga, Nithya Govindarajan, Manoj Govindassamy, Nathan Grinsztajn, Nikolas Gritsch, Patrick Gu, Shangmin Guo, Kilian Haefeli, Rod Hajjar, Tim Hawes, Jingyi He, Sebastian Hofstätter, Sungjin Hong, Sara Hooker, Tom Hosking, Stephanie Howe, Eric Hu, Renjie Huang, Hemant Jain, Ritika Jain, Nick Jakobi, Madeline Jenkins, JJ Jordan, Dhruti Joshi, Jason Jung, Trushant Kalyanpur, Siddhartha Rao Kamalakara, Julia Kedrzycki, Gokce Keskin, Edward Kim, Joon Kim, Wei-Yin Ko, Tom Kocmi, Michael Kozakov, Wojciech Kryściński, Arnav Kumar Jain, Komal Kumar Teru, Sander Land, Michael Lasby, Olivia Lasche, Justin Lee, Patrick Lewis, Jeffrey Li, Jonathan Li, Hangyu Lin, Acyr Locatelli, Kevin Luong, Raymond Ma, Lukáš Mach, Marina Machado, Joanne Magbitang, Brenda Malacara Lopez, Aryan Mann, Kelly Marchisio, Olivia Markham, Alexandre Matton, Alex McKinney, Dominic McLoughlin, Jozef Mokry, Adrien Morisot, Autumn Moulder, Harry Moynehan, Maximilian Mozes, Vivek Muppalla, Lidiya Murakhovska, Hemangani Nagarajan, Alekhya Nandula, Hisham Nasir, Shauna Nehra, Josh Netto-Rosen, Daniel Ohashi, James Owers-Bardsley, Jason Ozuzu, Dennis Padilla, Gloria Park, Sam Passaglia, Jeremy Pekmez, Laura Penstone, Aleksandra Piktus, Case Ploeg, Andrew Poulton, Youran Qi, Shubha Raghvendra, Miguel Ramos, Ekagra Ranjan, Pierre Richemond, Cécile Robert-Michon, Aurélien Rodriguez, Sudip Roy, Sebastian Ruder, Laura Ruis, Louise Rust, Anubhav Sachan, Alejandro Salamanca, Kailash Karthik Saravanakumar, Isha Satyakam, Alice Schoenauer Sebag, Priyanka Sen, Sholeh Sepehri, Preethi Seshadri, Ye Shen, Tom Sherborne, Sylvie Shang Shi, Sanal Shivaprasad, Vladyslav Shmyhlo, Anirudh Shrinivason, Inna Shteinbuk, Amir Shukayev, Mathieu Simard, Ella Snyder, Ava Spataru, Victoria Spooner, Trisha Starostina, Florian Strub, Yixuan Su, Jimin Sun, Dwarak Talupuru, Eugene Tarassov, Elena Tommasone, Jennifer Tracey, Billy Trend, Evren Tumer, Ahmet Üstün, Bharat Venkitesh, David Venuto, Pat Verga, Maxime Voisin, Alex Wang, Donglu Wang, Shijian Wang, Edmond Wen, Naomi White, Jesse Willman, Marysia Winkels, Chen Xia, Jessica Xie, Minjie Xu, Bowen Yang, Tan Yi-Chern, Ivan Zhang, Zhenyu Zhao, and Zhoujie Zhao. Command a: An enterprise-ready large language model, 2025.

[8] Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. Process reinforcement through implicit rewards, 2025.

[9] 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.

[10] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models, 2024.

[11] Etash Guha, Negin Raoof, Jean Mercat, Ryan Marten, Eric Frankel, Sedrick Keh, Sachin Grover, George Smyrnis, Trung Vu, Jon Saad-Falcon, Caroline Choi, Kushal Arora, Mike Merrill, Yichuan Deng, Ashima Suvarna, Hritik Bansal, Marianna Nezhurina, Yejin Choi, Reinhard Heckel, Seewong Oh, Tatsunori Hashimoto, Jenia Jitsev, Vaishaal Shankar, Alex Dimakis, Mahesh Sathiamoorthy, and Ludwig Schmidt. Evalchemy, November 2024.

[12] 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. *NeurIPS*, 2021.

[13] Shengyi Huang, Jiayi Weng, Rujikorn Charakorn, Min Lin, Zhongwen Xu, and Santiago Ontañón. Cleanba: A reproducible and efficient distributed reinforcement learning platform, 2023.

[14] Amirhossein Kazemnejad, Milad Aghajohari, Eva Portelance, Alessandro Sordoni, Siva Reddy, Aaron Courville, and Nicolas Le Roux. Vineppo: Unlocking rl potential for llm reasoning through refined credit assignment, 2024.

[15] Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 REINFORCE samples, get a baseline for free!, 2019.

[16] 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.

[17] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025.

[18] Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. `https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf`, 2024.

[19] 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.

[20] 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. `https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level_-1cf81902c14680b3bee5eb349a512a51`, 2025. Notion Blog.

[21] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL_-19681902c1468005bed8ca303013a4e2, 2025. Notion Blog.

[22] Justus Mattern, Manveer Basra, Sami Jaghouar, Matthew Di Ferrante, Felix Gabriel, and Johannes Hagemann. SYNTHETIC-1 Release: Two Million Collaboratively Generated Reasoning Traces from Deepseek-R1. https://www.primeintellect.ai/blog/synthetic-1-release, February 2025. Accessed: 2025-05-06.

[23] Justus Mattern, Sami Jaghouar, Manveer Basra, Jannik Straube, Matthew Di Ferrante, Felix Gabriel, Jack Min Ong, Vincent Weisser, and Johannes Hagemann. Synthetic-1: Two million collaboratively generated reasoning traces from deepseek-r1, 2025.

[24] Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–65, 2023.

[25] MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, Enwei Jiao, Gengxin Li, Guojun Zhang, Haohai Sun, Houze Dong, Jiadai Zhu, Jiaqi Zhuang, Jiayuan Song, Jin Zhu, Jingtao Han, Jingyang Li, Junbin Xie, Junhao Xu, Junjie Yan, Kaishun Zhang, Kecheng Xiao, Kexi Kang, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Zheng, Linbo Chai, Long Xing, Meizhi Ju, Mingyuan Chi, Mozhi Zhang, Peikai Huang, Pengcheng Niu, Pengfei Li, Pengyu Zhao, Qi Yang, Qidi Xu, Qiexiang Wang, Qin Wang, Qiuhui Li, Ruitao Leng, Shengmin Shi, Shuqi Yu, Sichen Li, Songquan Zhu, Tao Huang, Tianrun Liang, Weigao Sun, Weixuan Sun, Weiyu Cheng, Wenkai Li, Xiangjun Song, Xiao Su, Xiaodong Han, Xinjie Zhang, Xinzhu Hou, Xu Min, Xun Zou, Xuyang Shen, Yan Gong, Yingjie Zhu, Yipeng Zhou, Yiran Zhong, Yongyi Hu, Yuanxiang Fan, Yue Yu, Yufeng Yang, Yuhao Li, Yunan Huang, Yunji Li, Yunpeng Huang, Yunzhi Xu, Yuxin Mao, Zehan Li, Zekang Li, Zewei Tao, Zewen Ying, Zhaoyang Cong, Zhen Qin, Zhenhua Fan, Zhihang Yu, Zhuo Jiang, and Zijia Wu. Minimax-01: Scaling foundation models with lightning attention, 2025.

[26] Igor Molybog, Peter Albert, Moya Chen, Zachary DeVito, David Esiobu, Naman Goyal, Punit Singh Koura, Sharan Narang, Andrew Poulton, Ruan Silva, Binh Tang, Diana Liskovich, Puxin Xu, Yuchen Zhang, Melanie Kambadur, Stephen Roller, and Susan Zhang. A theory on adam instability in large-scale machine learning, 2023.

[27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2018.

[28] Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous rlhf: Faster and more efficient off-policy rl for language models, 2025.

[29] Jack Min Ong, Matthew Di Ferrante, Aaron Pazdera, Ryan Garner, Sami Jaghouar, Manveer Basra, and Johannes Hagemann. Toploc: A locality sensitive hashing scheme for trustless verifiable inference, 2025.

[30] OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan,

David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[32] Alexandre Ramé, Johan Ferret, Nino Vieillard, Robert Dadashi, Léonard Hussenot, Pierre-Louis Cedoz, Pier Giuseppe Sessa, Sertan Girgin, Arthur Douillard, and Olivier Bachem. Warp: On the benefits of weight averaged rewarded policies, 2024.

[33] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *CoRR*, abs/2101.06840, 2021.

[34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[35] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

[36] 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.

[37] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy,

Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report, 2025.

[38] Llama 4 Team. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation, April 2025.

[39] Qwen Team. Qwen3, April 2025.

[40] Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

[41] 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.

[42] Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Monica Lam, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning, 2025.

[43] Mitchell Wortsman, Peter J. Liu, Lechao Xiao, Katie Everett, Alex Alemi, Ben Adlam, John D. Co-Reyes, Izzeddin Gur, Abhishek Kumar, Roman Novak, Jeffrey Pennington, Jascha Sohl-dickstein, Kelvin Xu, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Small-scale proxies for large-scale transformer training instabilities, 2023.

[44] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, 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, Weinan Dai, 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.

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