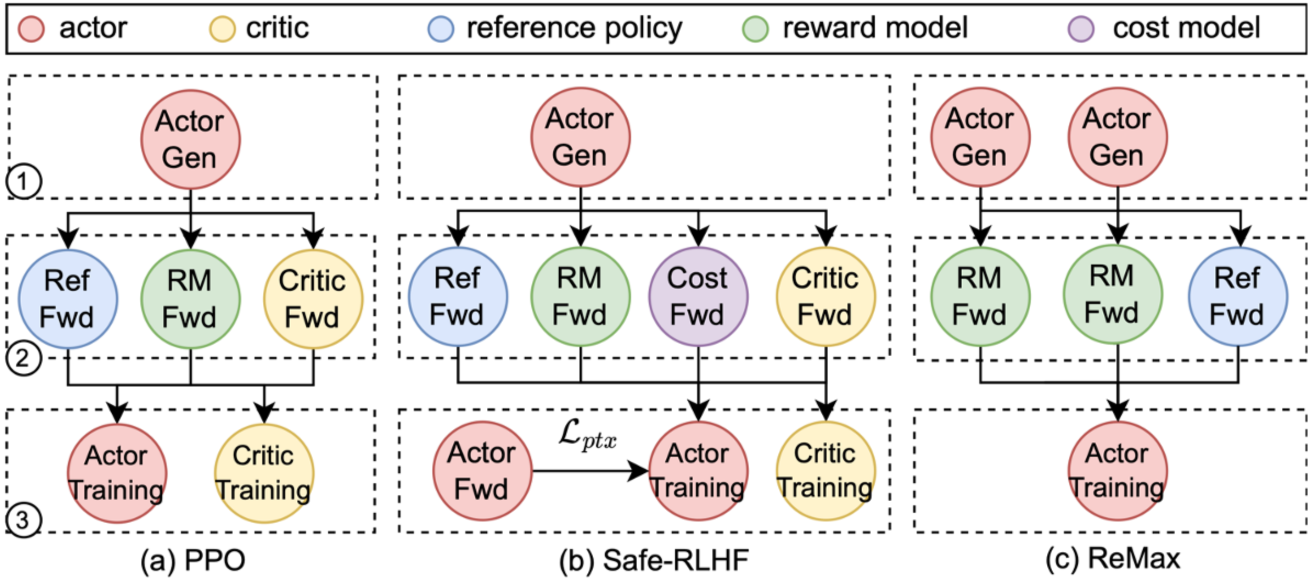
verl:
Flexible and Efficient RL for LLMs

Yaowei Zheng

Committer@verl

1 Background

1.1 RL as Dataflow Graph

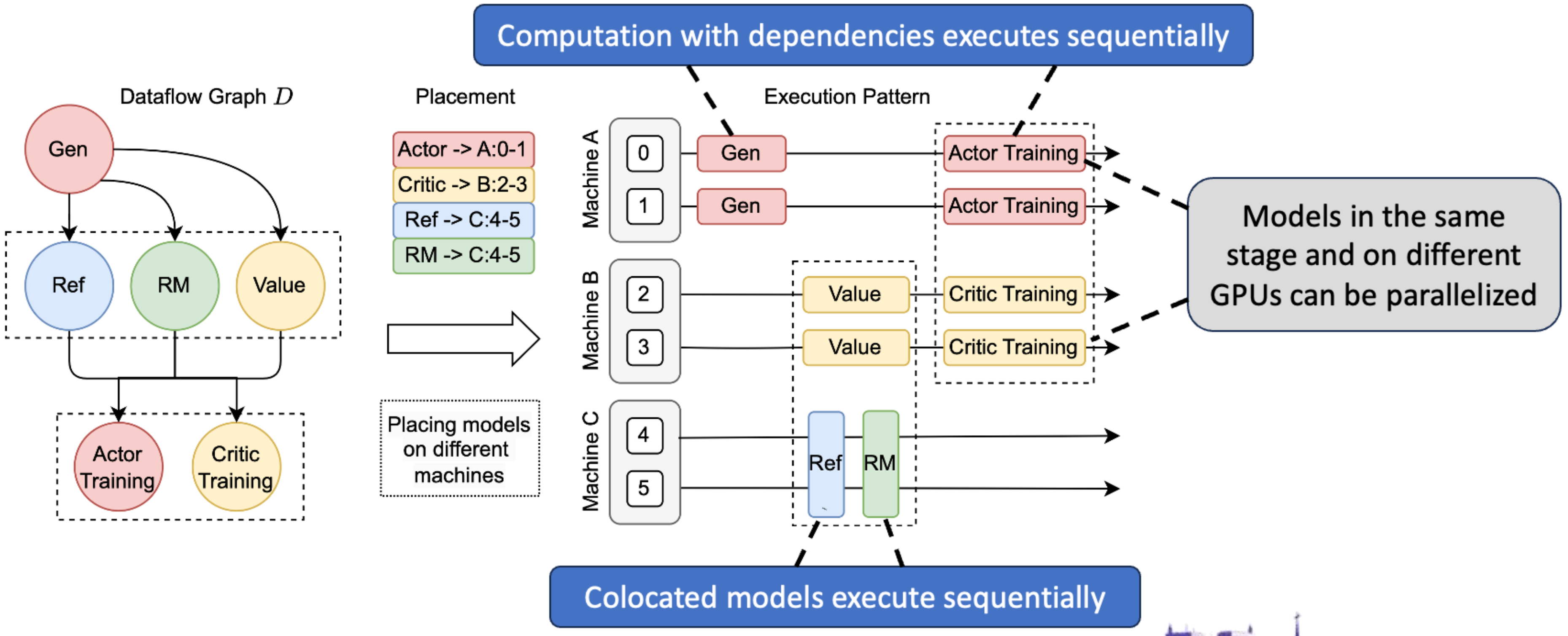


Reinforcement Learning (RL) for LLM Post-Training can typically be modeled as a **dataflow graph**, consisting of:

1. **multiple models**: actor, critic, reference, reward model, etc.
2. **multiple stages**: generating, preparing experiences, training
3. **multiple workloads**: generation, inference, training

1.2 Implementing Dataflow Graph as Execution Pattern

In practice, we should implement the dataflow graph as execution pattern on GPU cluster.



2 verl: Flexible and Efficient RL for LLMs

2.1 Paradigm: Hybrid-Controller

verl introduces a hybrid-controller paradigm, consisting of

- a single-controller (e.g. RayPPOTrainer) that concentrates the training control logic in a single process
- multiple multi-controllers (e.g. ActorRolloutWorker) that conduct the distributed computation in a complex but efficient way

2.2 Flexibility: Single-Controller

Thanks to the programming model of single-controller, verl allows implementing different RL algorithms by **only modifying a few lines**, usually only in the fit function.

Listing 1: PPO example code.

```
for prompts in dataloader:
    # Stage 1: Sampling Trajectories
    batch = actor.generate_sequences(prompts)
    # Stage 2: Preparing Experiences
    batch = reward.compute_reward(batch)
    batch = reference.compute_log_prob(batch)
    batch = critic.compute_values(batch)
    batch = compute_advantage(batch, "gae")
    # Stage 3: Training
    critic.update_critic(batch)
    actor.update_actor(batch)
```

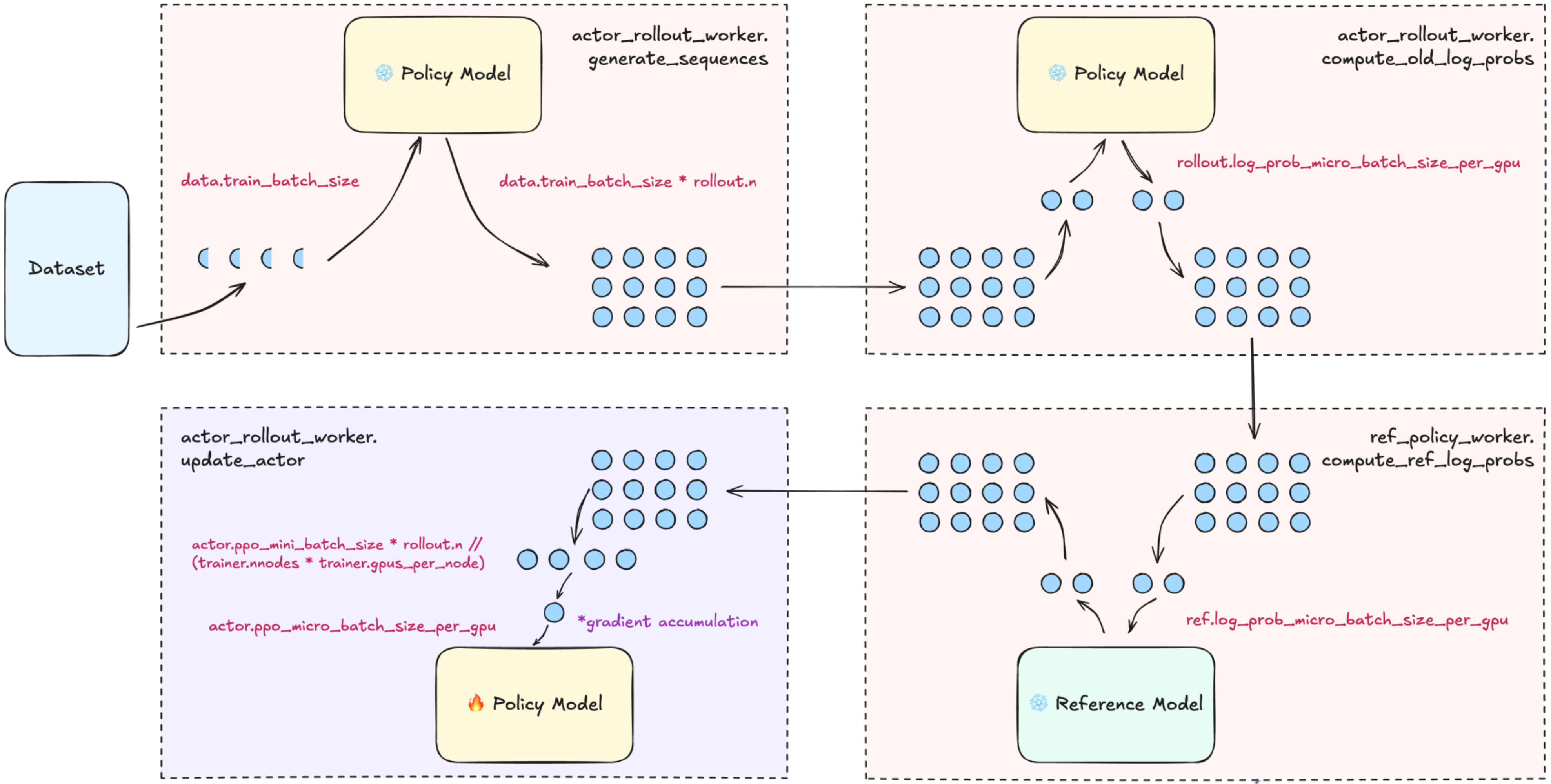
Listing 2: GRPO example code.

```
for prompts in dataloader:
    # Stage 1: Sampling Trajectories
    batch = actor.generate_sequences(prompts)
    # Stage 2: Preparing Experiences
    batch = reward.compute_reward(batch)
    batch = reference.compute_log_prob(batch)
    batch = compute_advantage(batch, "grpo")
    # Stage 3: Training
    critic.update_critic(batch)
    actor.update_actor(batch)
```

With such flexibility, verl has supported diverse RL algorithms

including PPO, GRPO, RLOO, ReMax, REINFORCE++, PRIME, DAPO, DrGRPO, etc.

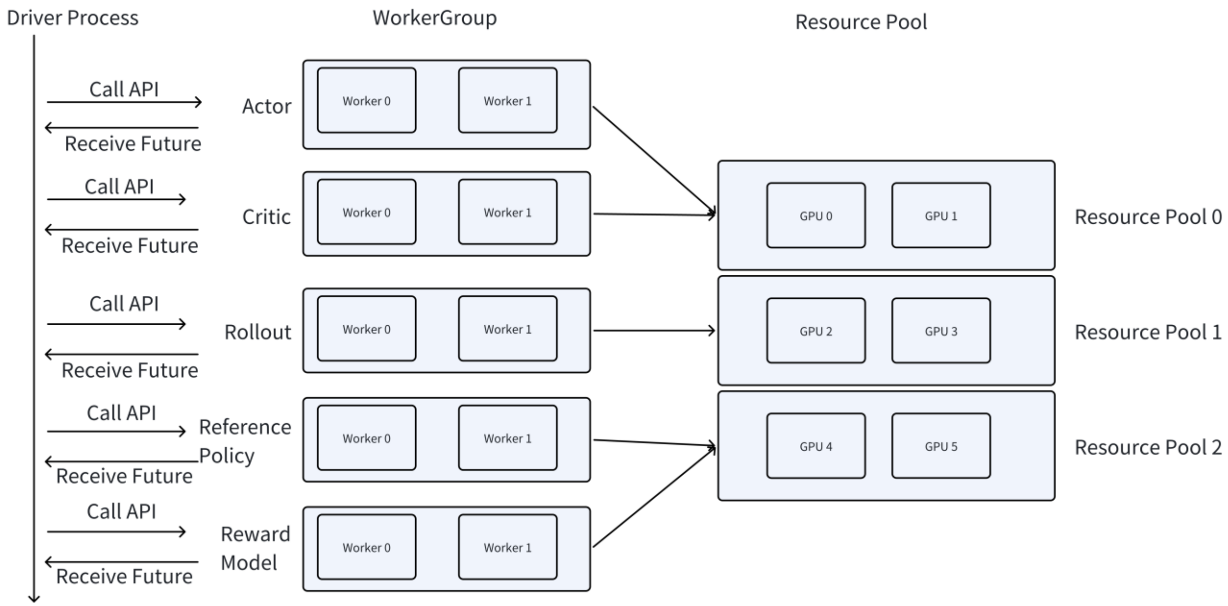
2.2 Flexibility: Single-Controller



2.3 Efficiency: Hybrid Engine

GOSIM

The optimal execution pattern for different workloads, e.g., training, generation, are usually different. Instead of splitting the devices to deploy different engines separately for different workloads, causing many bubbles, verl implements a hybrid engine that can switch between the different procudures on the same cluster, fully utilizing all the GPUs.



2.3 Efficiency: Hybrid Engine

GOSIM

Thanks to the hybrid engine, verl allows flexibly switching between parallelism strategies to optimize the performance.

Training:

- PyTorch FSDP
- DeepSpeed Ulysses
- Megatron 3D Parallelism
  - Tensor Parallelism
  - Pipeline Parallelism
  - Sequence Parallelism

Generation:

- vLLM Tensor Parallelism
- SGLang Tensor Parallelism
- Expert Parallelism (Comming Soon)

2.3 Efficiency: Hybrid Engine

Data Parallelism (DP) is the most commonly used parallelism strategy.

However, DP performance might be damaged by load imbalance, which is especially severe in long-context training.

verl implements the following feature to improve load balance:

1. balance_batch: make the token numbers of the samples dispatched to each DP rank as balanced as possible by reordering the samples in each batch.

However, in gradient accumulation, it's not enough to only balance the total number of tokens for each rank in a batch, since DP syncs in the unit of micro batch.

So here comes the second feature:

2. use_dynamic_bsz: deviding the batch into micro batches in such a way that the token numbers of the micro batches are as balanced as possible.

2.3 Efficiency: Hybrid Engine

GOSIM

verl also supports other optimization tricks:

1. Padding-free training (use_remove_padding): verl can save computation by removing padding tokens based on Flash Attention 2.
2. Gradient checkpointing (enable_gradient_checkpointing)
3. Torch compile (use_torch_compile)
4. Liger kernel (use_liger)
5. LoRA (lora_rank etc.)
6. …

3 Programming Guide

3.1 Customizing the Dataset

A canonical RL dataset in verl has the following fields:

- **prompt**: a list of messages {"role": "...", "content": "..."}
- **data_source**: used to choose the reward function
- **reward_model**: a dict containing
  - "**ground_truth**"
  - "style" like "**model**" or "**rule**"
- (Optional) **extra_info**: a dict containing extra information

For VLM RL, verl expects fields "**images**" and/or "**videos**"

For examples, please check the examples/data_preprocess.

3.2 Customizing the Reward

verl allows to define custom reward function via the custom_reward_function config:

```yaml
custom_reward_function:
  path: null # path to the `.py` file containing the function definition
  name: compute_score # the function name after `def`
reward_model:
  reward_manager: naive
```

An example CLI config could be:

```
--custom_reward_function.path=./examples/reward_fn/custom_reward_fn.py \
--custom_reward_function.name=compute_score \
--reward_model.reward_manager=naive
```

3.2 Customizing the Reward

The function defined in the python script should accept the parameters passed from **the reward manager __call__ method**. Taking NaiveRewardManager as an example:

```
class NaiveRewardManager:
    def __call__(self, data: DataProto, return_dict: bool=False):
        # Preprocessing for the input data
        score = self.compute_score(
            data_source=data_source,
            solution_str=solution_str,
            ground_truth=ground_truth,
            extra_info=extra_info,
        )
        # Other processing for the final `reward`
```

For more complex features, you can also add a new reward manager like PRIMERewardManager or DAPORewardManager.

3.3 Customizing the Loss Function

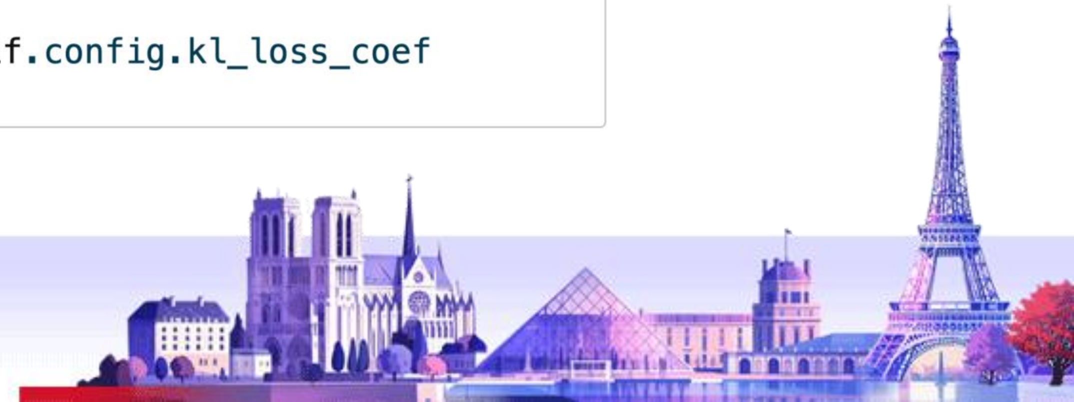
To modify the loss function, the most convenient way is to

1. Search for the .backward() call
2. Modify functions like compute_policy_loss
3. Or add loss terms like entropy_loss

3.3 Customizing the Loss Function

For example, the DataParallelPPOActor.update_policy method defines the loss function as follows:

```python
class DataParallelPPOActor(BasePPOActor):
    def update_policy(self, data: DataProto):
        pg_loss = compute_policy_loss(
            old_log_prob=old_log_prob, log_prob=log_prob,
            advantages=advantages, # ...
        )
        entropy_loss = agg_loss(loss_mat=entropy)
        policy_loss = pg_loss - entropy_loss * entropy_coeff
        kld = kl_penalty(
            logprob=log_prob, ref_logprob=ref_log_prob, # ...
        )
        kl_loss = agg_loss(loss_mat=kld)
        policy_loss = policy_loss + kl_loss * self.config.kl_loss_coef
        loss.backward()
```



3.4 Customizing the Training Loop

As mentioned above, the main training logic is concentrated in the **fit** function of the trainer classes like RayPPOTrainer.

For example, the DAPORayTrainer class overrides the fit function to implement “dynamic sampling”:

```
class RayDAPOTrainer(RayPPOTrainer):
  def fit(self):
    for epoch in range(self.config.trainer.total_epochs):
      batch = None
      for batch_dict in self.train_dataloader:
        new_batch = DataProto.from_single_dict(batch_dict)
        num_gen_batches += 1
        gen_batch_output = self.actor_rollout_wg.generate_sequences(gen_batch)
        new_batch = new_batch.union(gen_batch_output)
        if not self.config.algorithm.filter_groups.enable:
          batch = new_batch
        else:
          # Getting `kept_traj_idxs` ...
          new_batch = new_batch[kept_traj_idxs]
          batch = new_batch if batch is None else DataProto.concat([batch, new_batch])
          prompt_bsz = self.config.data.train_batch_size
          if num_prompt_in_batch < prompt_bsz:
            max_num_gen_batches = self.config.algorithm.filter_groups.max_num_gen_batches
            if max_num_gen_batches <= 0 or num_gen_batches < max_num_gen_batches:
                continue
          else:
            traj_bsz = self.config.data.train_batch_size * self.config.actor_rollout_ref.rollout.n
            batch = batch[:traj_bsz]
        # ...
```

4 Latest Features

4.1 Efficient RL Training for DeepSeek V3 671B

verl is approaching finishing the support for efficient RL training for huge MoE like DeepSeek-V3-671B, based on the following features:

1. MoE models with GPTModel class for actor and critic
2. Multi-node inference
3. Parameter sharding manager for Megatron-Core V0.12 + latest version of inference engines

For more details, please check our PR #708.

4.2 Async Engine for Multi-Turn Conversation

The awesome SGLang RL team

1. has integrated the SGLang async engine into verl
2. is validating an extensible tool interface OpenAIFunctionTool with end-to-end training

For more details, please check their PR #1037.

Besides, our team also integrates the async engine based on vLLM V1 AsyncLLM. Kudos to our awesome collegue **Xibin Wu** for his great work!

4.3 Other Updates

1. Full support for RL with AMD (ROCm Kernel) hardwares
2. Full support for VLM RL based on SGLang
3. …

For the most timely updates of important features, please keep an eye on verl’s README.

4.3 Other Updates

For related resources like

- Slack channel for Q&A
- Out-of-the-box verl demo
- Slides
- Other links

etc., please scan the QR code:

GOSIM AI Paris 2025

THANK YOU