



Reinforcement Learning Systems

Carnegie Mellon University

Language Technologies Institute

Apurva Gandhi

11-768: AI Agents

RL Systems: Overview

What operations must an LLM RL Framework support?

Policy
Gradient
Update:

$$\theta \ += \sum_t (R(\tau) - b(h_t)) \nabla_{\theta} \log \pi_{\theta}(a_t | h_t)$$

The diagram illustrates the components of the policy gradient update equation and the operations required to compute them:

- env.verify**: Points to the $R(\tau)$ term in the equation.
- env.state / env.step**: Points to the $b(h_t)$ term in the equation.
- + agent harness**: Points to the $\nabla_{\theta} \log \pi_{\theta}(a_t | h_t)$ term in the equation.
- optim_step**: Points to the θ term in the equation.
- backward**: Points to the ∇_{θ} term in the equation.
- forward**: Points to the $\log \pi_{\theta}(a_t | h_t)$ term in the equation.

Why not just hand-roll custom implementation in vanilla PyTorch?

Policy

Gradient

Update:

$$\theta += \sum_t (R(\tau) - b(h_t)) \nabla_{\theta} \log \pi_{\theta}(a_t | h_t)$$

↑
optim_step

↑
backward

↑
forward

PyTorch
API:

`optimizer.step()`

`model.backward()`

`model.forward()`

RL Frameworks are composed of two fundamentally different workloads!

Policy

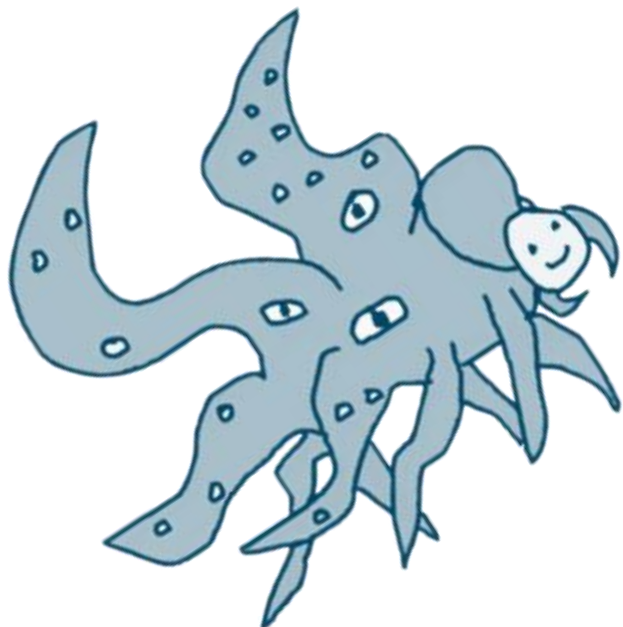
Gradient
Update:

$$\theta += \sum_t (R(\tau) - b(h_t)) \nabla_{\theta} \log \pi_{\theta}(a_t | h_t)$$

↑
optim_step

↑
“sample”

↗ ↖
“forward_backward”



Inference Engine
e.g., VLLM, SGLang

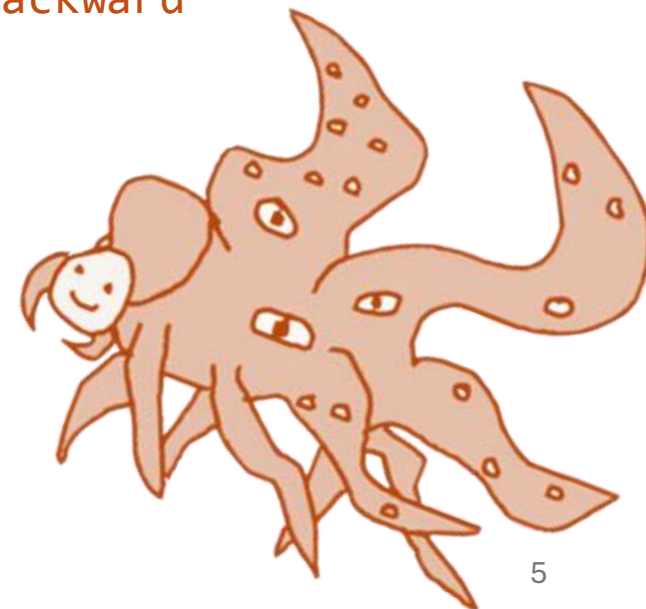
Decode-heavy
(Sequential)

No grad

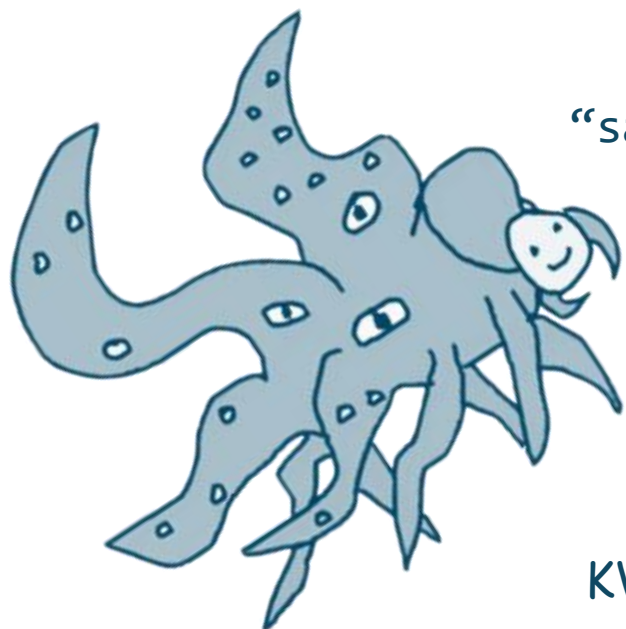
Trainer Engine
e.g., Megatron, FSDP2

Prefill-style
(Parallelizable)

Autograd



RL Frameworks are composed of two fundamentally different workloads!



“sample”

Inference Engine
e.g., VLLM, SGLang

Decode-heavy
(Sequential)

KV Caching

Radix/Prefix Caching (across requests)

Continuous Batching

Speculative Decoding/MTP

Model sharding (PP, CP, TP, EP, etc.)

“forward_backward”

Trainer Engine
e.g., Megatron, FSDP2

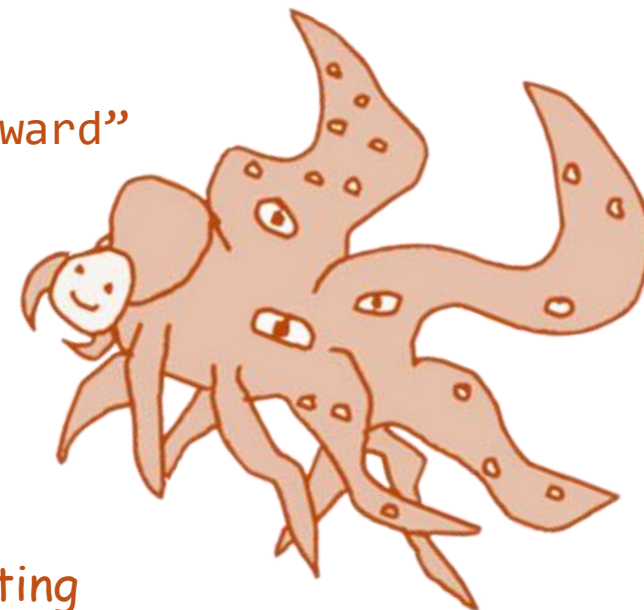
Prefill-style
(Parallelizable)

Activation Checkpointing

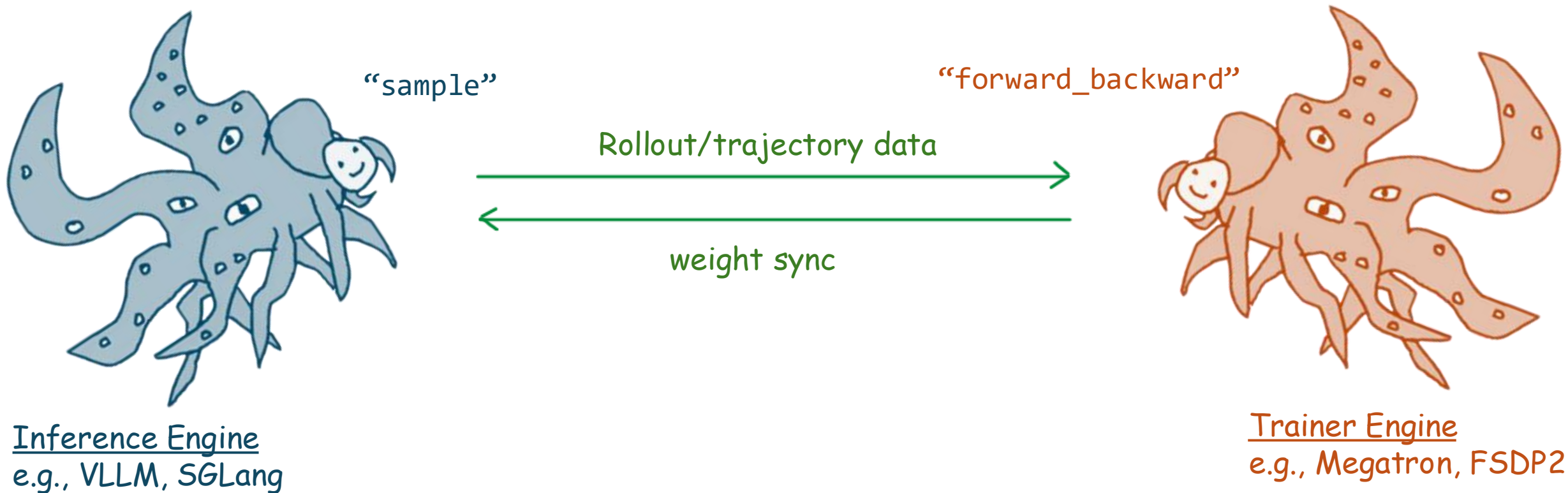
Prefix sharing

Batch Sample Packing

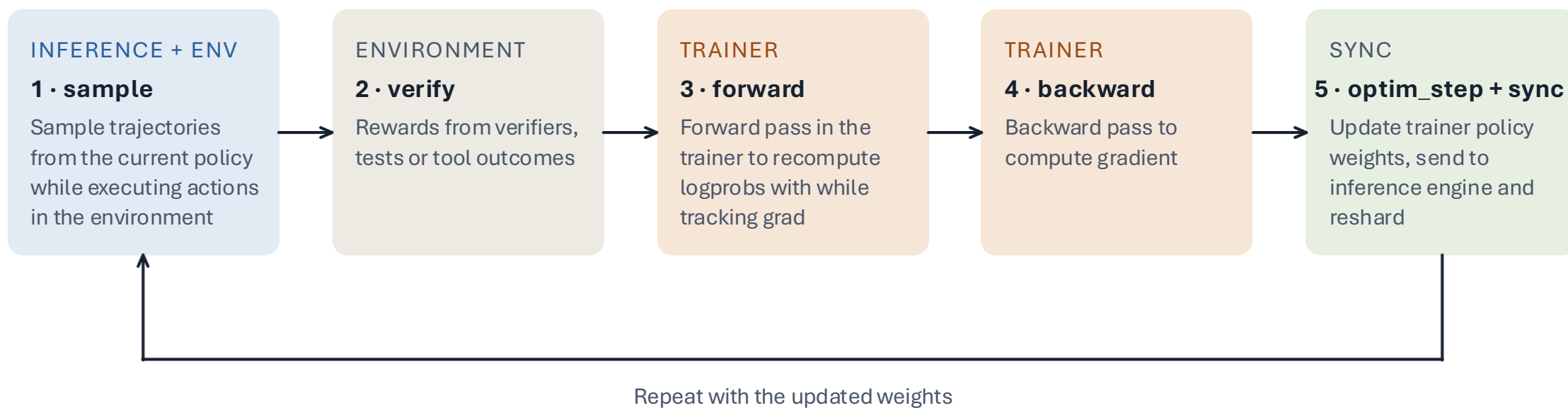
Model sharding (PP, CP, EP, etc.)



Interaction Between Train and Inference Engines



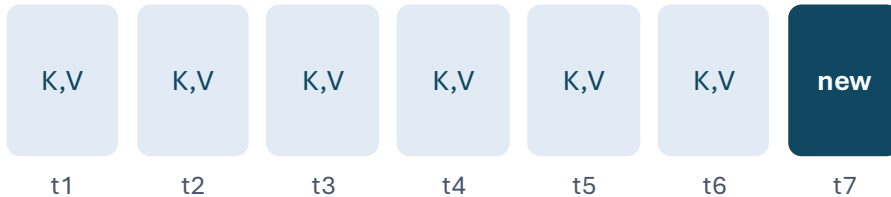
Breaking down a training step





Inference Optimizations: KV Caching

KV CACHE · ONE ENTRY PER PAST TOKEN



Step t7 computes q, k, v for **the new token only**, attends over every cached k, v, then appends its own.

HOW BIG IS IT?

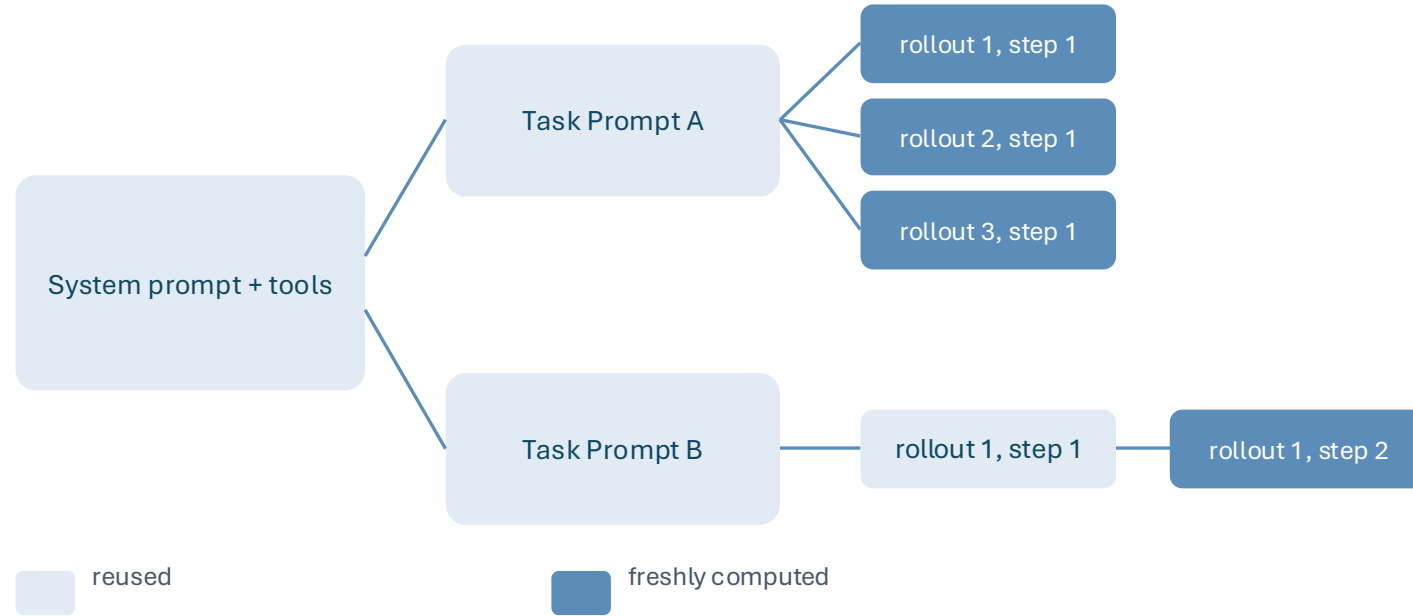
Llama-3-8B in bf16:

- **128 KB per token**
- 1 32k-token trace $\approx$ **4 GB**;
- 64 of them $\approx$ **256 GB**

KV Caching trades a lot of recomputation for a lot of memory.



Inference Optimizations: Prefix Caching



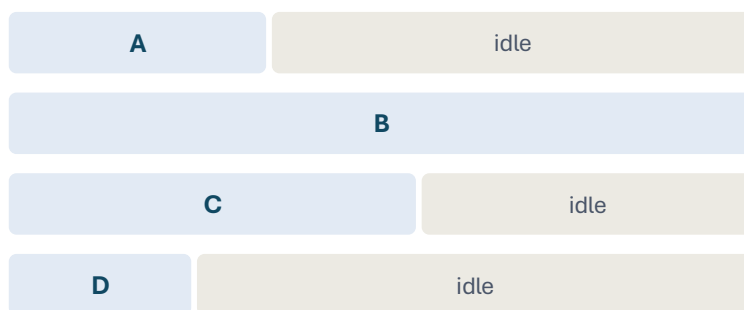
Agentic RL especially benefits from prefix caching.

- In Group Rollouts (e.g., GRPO), many rollouts share the same task prompts.
- KV from past steps/history of a rollout can be reused in future steps.

Inference Optimizations: Continuous Batching

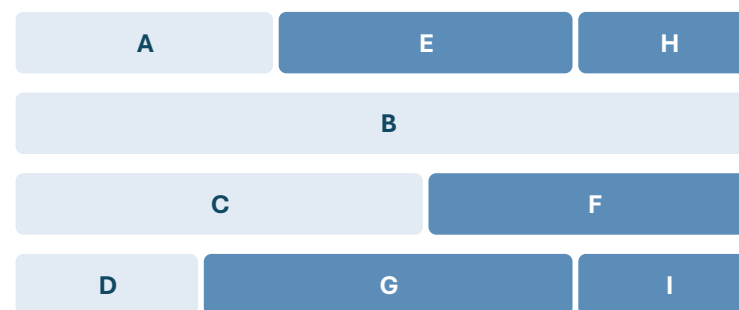


STATIC BATCHING



Every slot waits for the longest request (B).

CONTINUOUS BATCHING



Freed slots are refilled at the next step.

Schedule at the granularity of one decode step: finished sequences leave the batch at once, and queued requests take their slots.

Usually accompanied with chunked prefill to avoid long prompts from causing decode latency spikes. Alternatively, can disaggregate prefill and decode.

Inference Optimizations: Speculative Decoding



A cheap drafter guesses k tokens; the policy checks all k in **one parallel pass**. Rejection sampling keeps the policy's exact distribution.



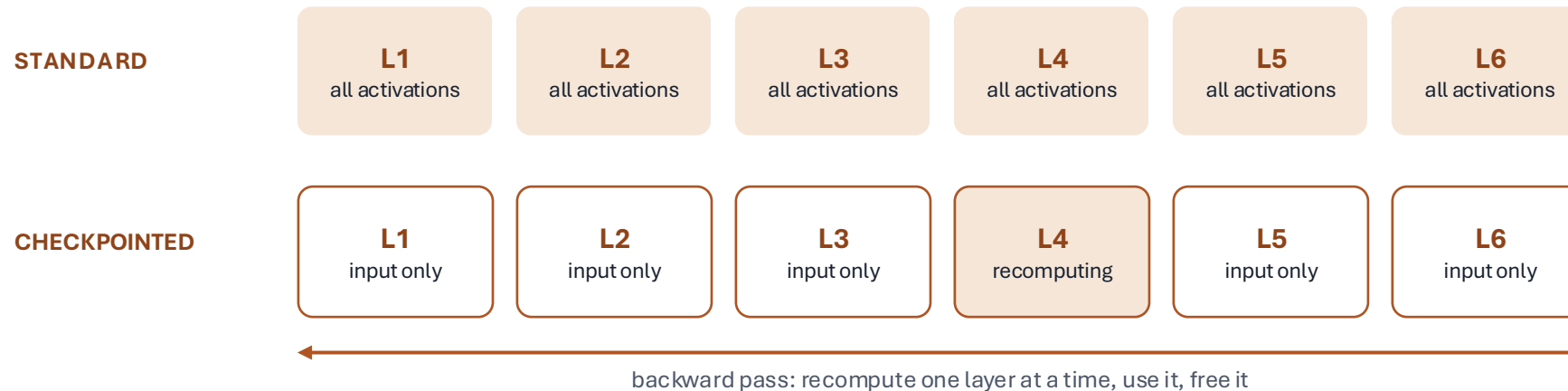
RL-SPECIFIC CATCH

The policy changes every training step, so a frozen drafter's acceptance rate decays. Train it online with an auxiliary objective (e.g., SFT, distillation).

Trainer Optimizations: Activation Checkpointing



Keep only each layer's **input** in the forward pass; recompute the layer's internals when the backward pass reaches it.



MEMORY

From every layer's activations to one input per layer, plus a single layer's activations at a time.

COST

Full recompute adds ~one forward pass: **≈ +33% compute**. Selective recompute (only attention) costs far less.

Trainer Optimizations: Prefix Sharing

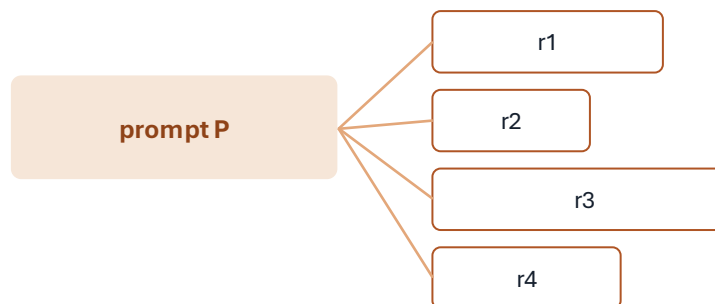


RL batches repeat prefixes: the prompt across a GRPO group, the history across turns. **Compute each once, forward and backward.** (*Counter-part to prefix caching in inference*)

DENSE · PROMPT REPEATED



SHARED · PROMPT ONCE



Tree attention mask (no attention across branches) · position IDs as in each standalone sequence · gradients from every branch accumulate into P

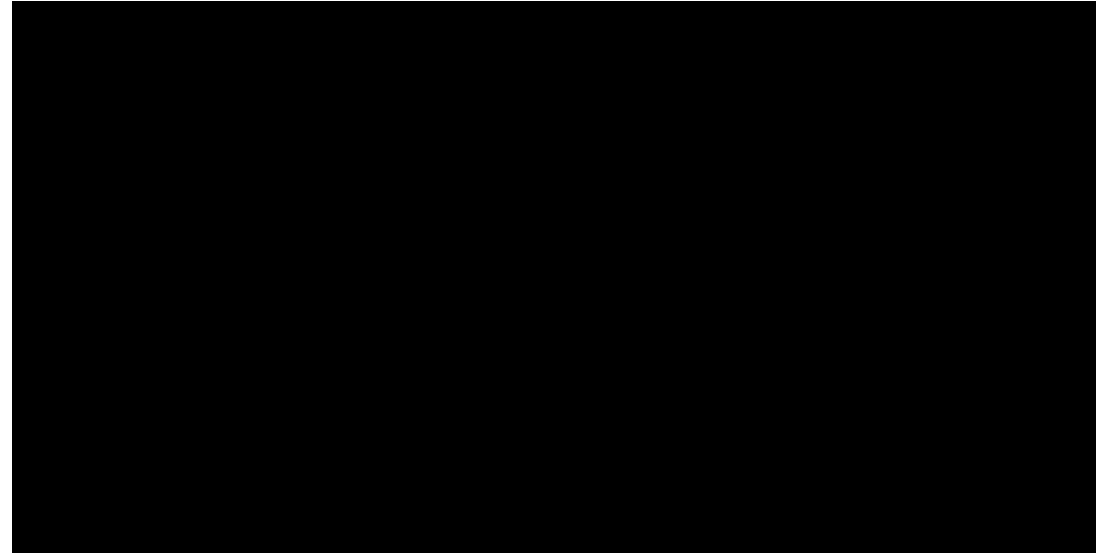
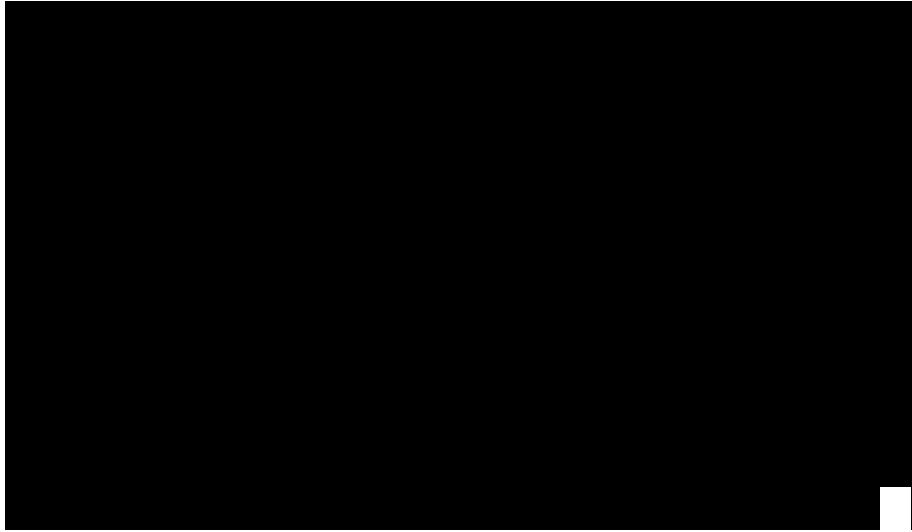
SHARED PROMPTS · SNOWFLAKE ZoRRo

In long-prompt GRPO batches, **80–95%** of tokens are duplicate prompt tokens. Deduplicating them gives up to **6×** faster actor updates.

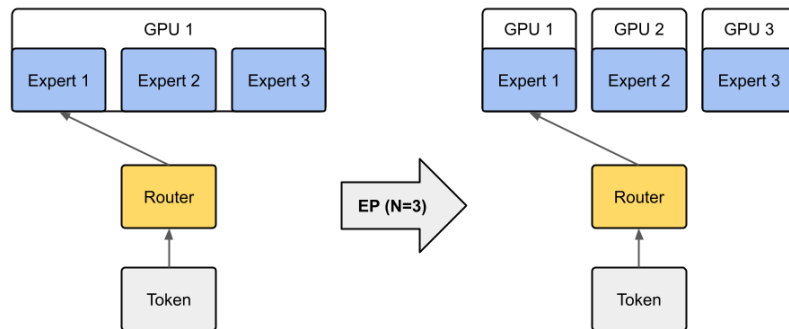
PREFIX TREES · AReaL-DTA

Multi-turn τ^2 -bench rollouts compress **9.43×** as a prefix tree. A DFS walks one root-to-leaf path at a time: up to **8.31×** vs dense training.

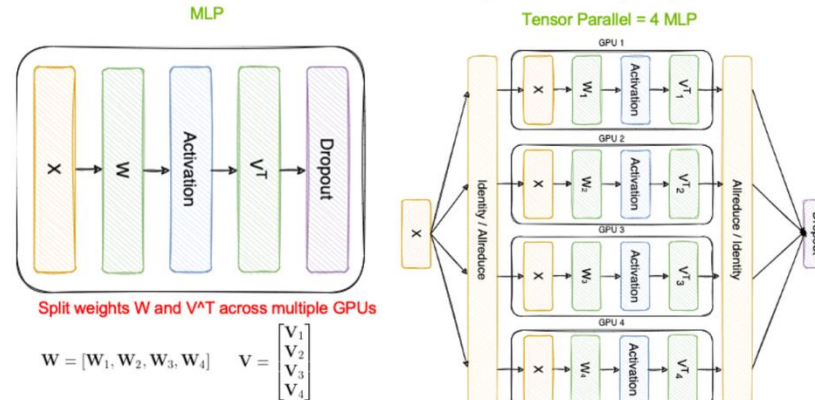
Optimizations: Parallelism



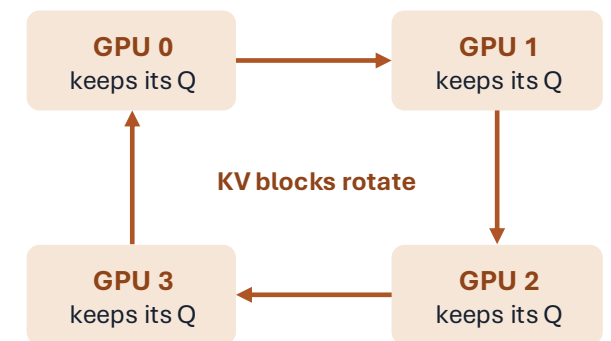
Expert Parallelism applied on Mixture-of-Experts.

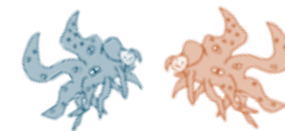


Tensor Parallelism (Intra-Layer)



Context Parallelism





Train-Inference Interactions: Weight Sync

Full Weight Sync:

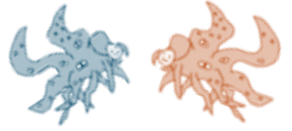
- Full broadcast to every inference rank.
- 1T parameter model $\rightarrow$ ~2TB sync

- **Delta Weight Sync:**

- ~99% of parameters unchanged between RL weight updates bf16 change.
- Sync only the delta.

Note: Parameters need to be re-sharded for inference engine parallelism layout.

Train-Inference Interactions: Rollout Routing Replay (R3)



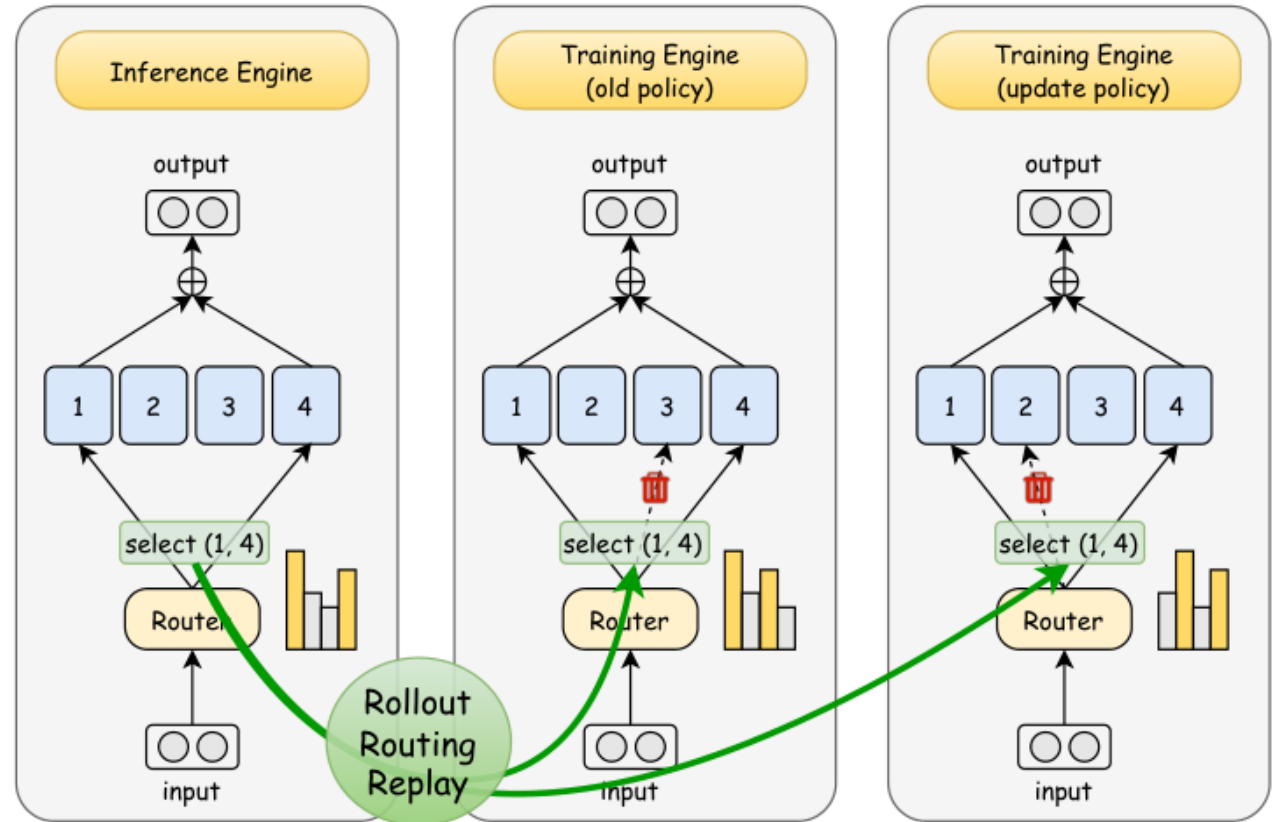
Problem

Tiny numerical differences can select different MoE experts.

R3 Solution

Record rollout expert choices and replay the selections in training.

Inference engine must provide routing decisions to training engine.

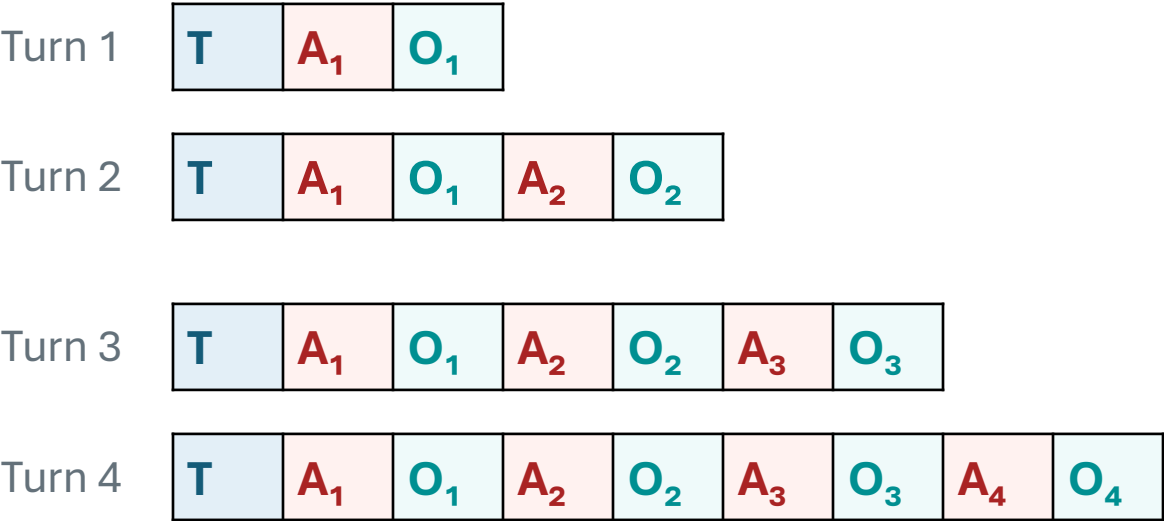


Agentic RL: The Sharp Bits

Sequence Extension

Harness A

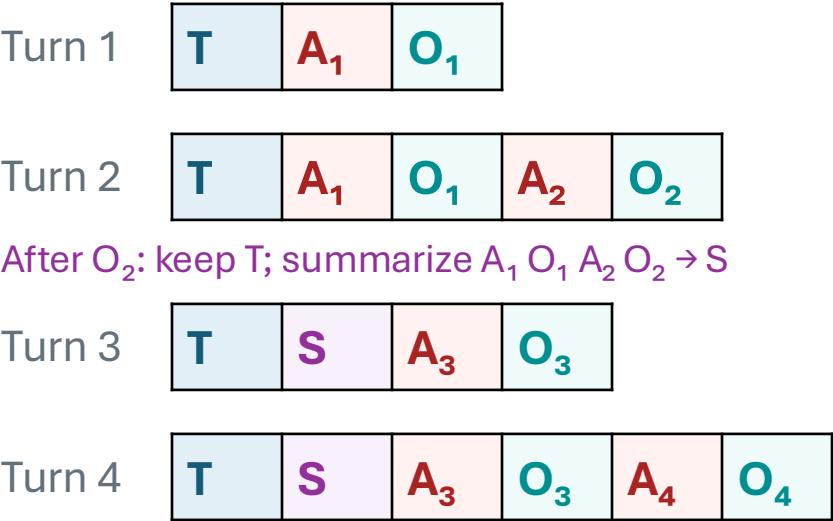
Append to the existing history



T = task A_i = action O_i = observation S = compacted interaction history

Harness B

Compact every two observations



Which harness is better for training efficiency?

Sequence Extension: Training Batch

One trajectory advantage $\hat{A}$, broadcast to every action token

Harness A: one concatenated datapoint

Token span	Task	A_1	O_1	A_2	O_2	A_3	O_3	A_4	O_4
Loss mask	0	1	0	1	0	1	0	1	0
Advantage	0	$\hat{A}$	0	$\hat{A}$	0	$\hat{A}$	0	$\hat{A}$	0

Harness B: two datapoints, both containing Task

Token span	Task	A_1	O_1	A_2	O_2	Task	S	A_3	O_3	A_4	O_4
Loss mask	0	1	0	1	0	0	0	1	0	1	0
Advantage	0	$\hat{A}$	0	$\hat{A}$	0	0	0	$\hat{A}$	0	$\hat{A}$	0

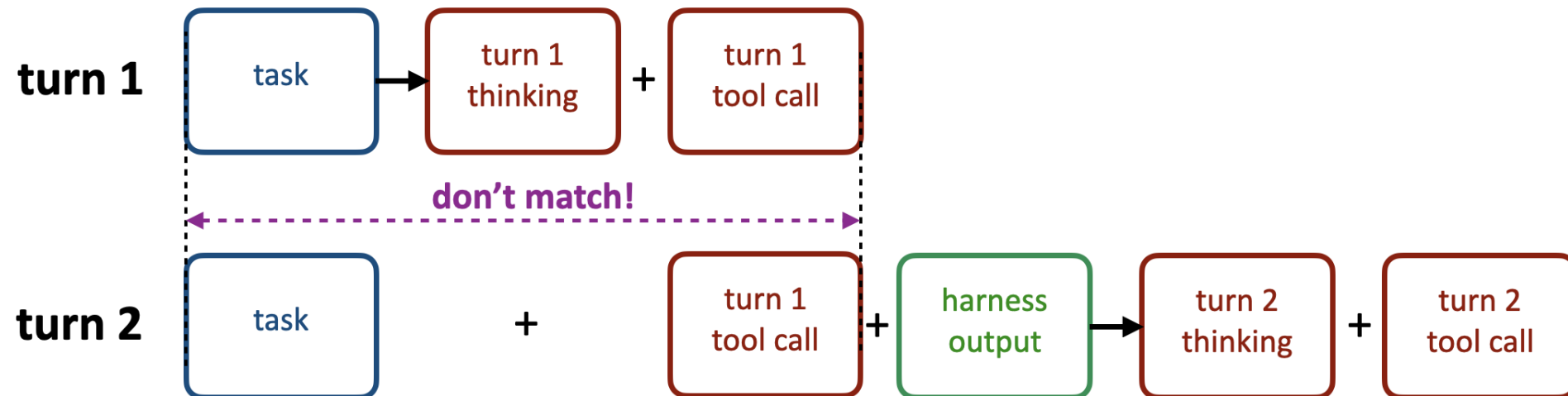
Repeated task tokens have zero direct loss, but still require context computation.

Preserve exact prefixes and concatenate for as long as practical.

Chat Template Quirks

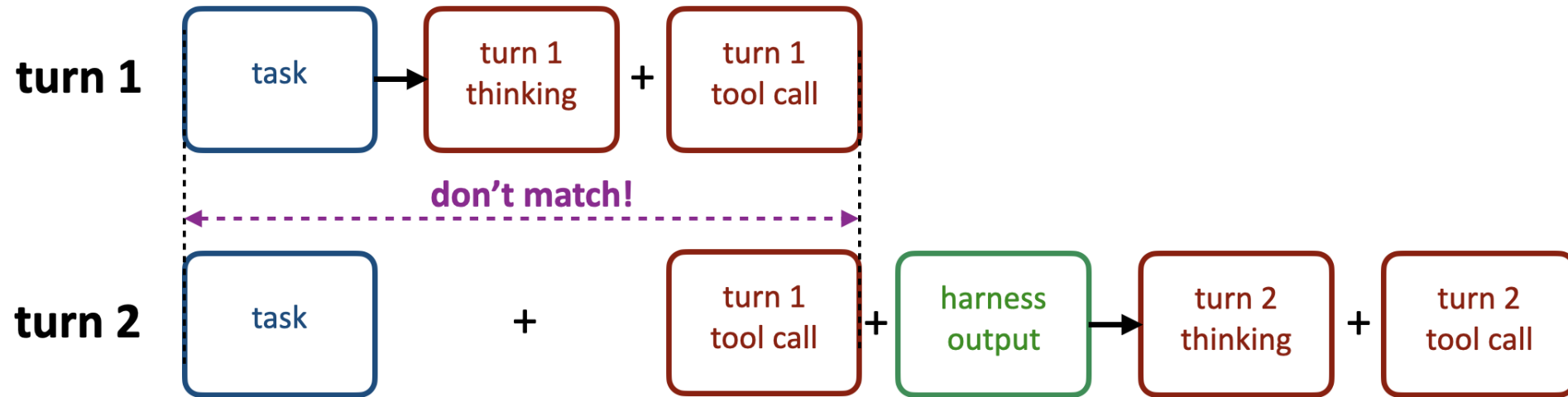
Popular chat templates may not satisfy the sequence extension property.

E.g., Qwen 3 / Qwen 3.5's default chat template removes reasoning from past history.



Chat Template Quirks

E.g., Qwen 3 / Qwen 3.5's default chat template removes reasoning from past history.



Agent RL training frameworks often ship with their own custom chat template or “renderers” that preserve sequence extension (e.g., Tinker, PrimeRL).

Token-In, Token-out (TITO)

In what format should we store and transport rollout/trajectory data from the inference engines to the trainer engines?

1. String
2. Tokens

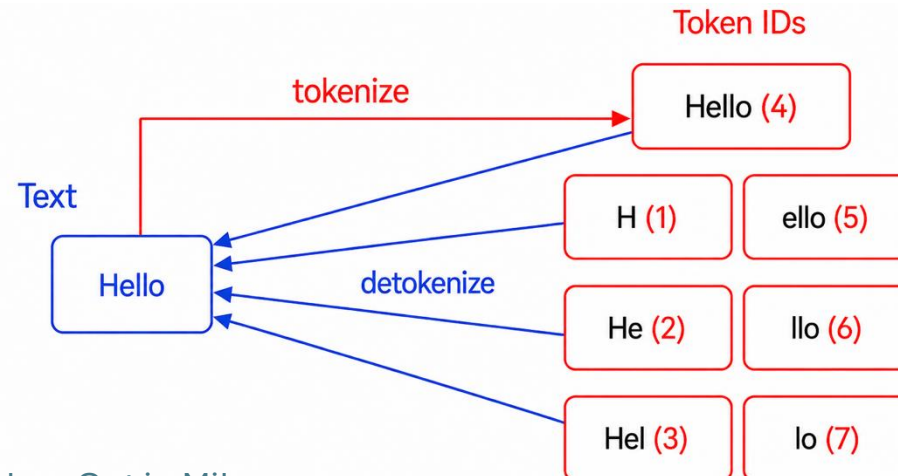
Token-In, Token-out (TITO)

In what format should we store and transport rollout/trajectory data from the inference engines to the trainer engines?

~~1. String~~

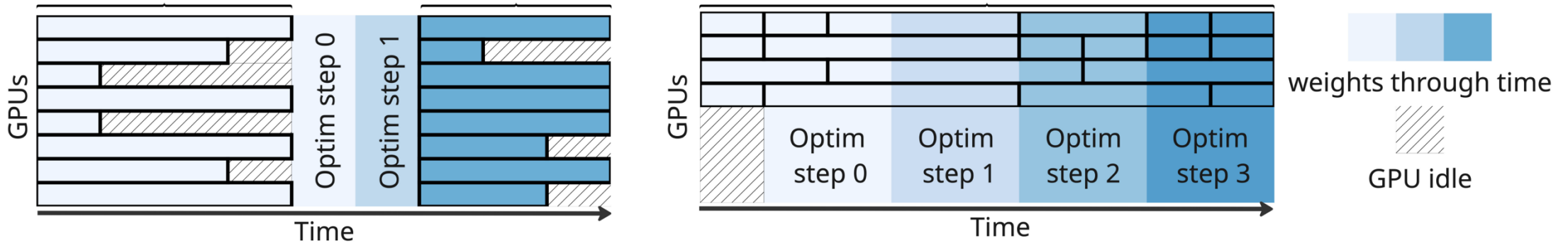
2. Tokens

- Small, inconsistent transformations in string space can lead to inconsistent tokenizations between train and inference (e.g., chat template adding some whitespace)*
- Tokenization is one-to-one, but de-tokenization is many-to-one.*



Async RL

Overlap inference and weight updates. Reduces GPU idle due to very long rollouts. Requires staleness/off-policy corrections.



Sync RL

- Co-locate Train & Inference Engines
(same GPUs shared by both)

Async RL

- Split GPUs between Train & Inference
- Agentic RL is often rollout-bound (long trajectories), so can dedicate more GPUs to inference (e.g., 3:1 ratio)
- Requires *continuous batching* and *in-flight weight-update support*.

In-flight Weight Updates

Inference frameworks have evolved to accommodate RL systems needs.

Example from [SGLang Docs](#).

Pause Generation

Endpoint: `POST /pause_generation`

Request body:

Field	Description	Defaults	Options
<code>mode</code>	Pause mode.	<code>abort</code>	<code>abort</code> , <code>retract</code> , <code>in_place</code>

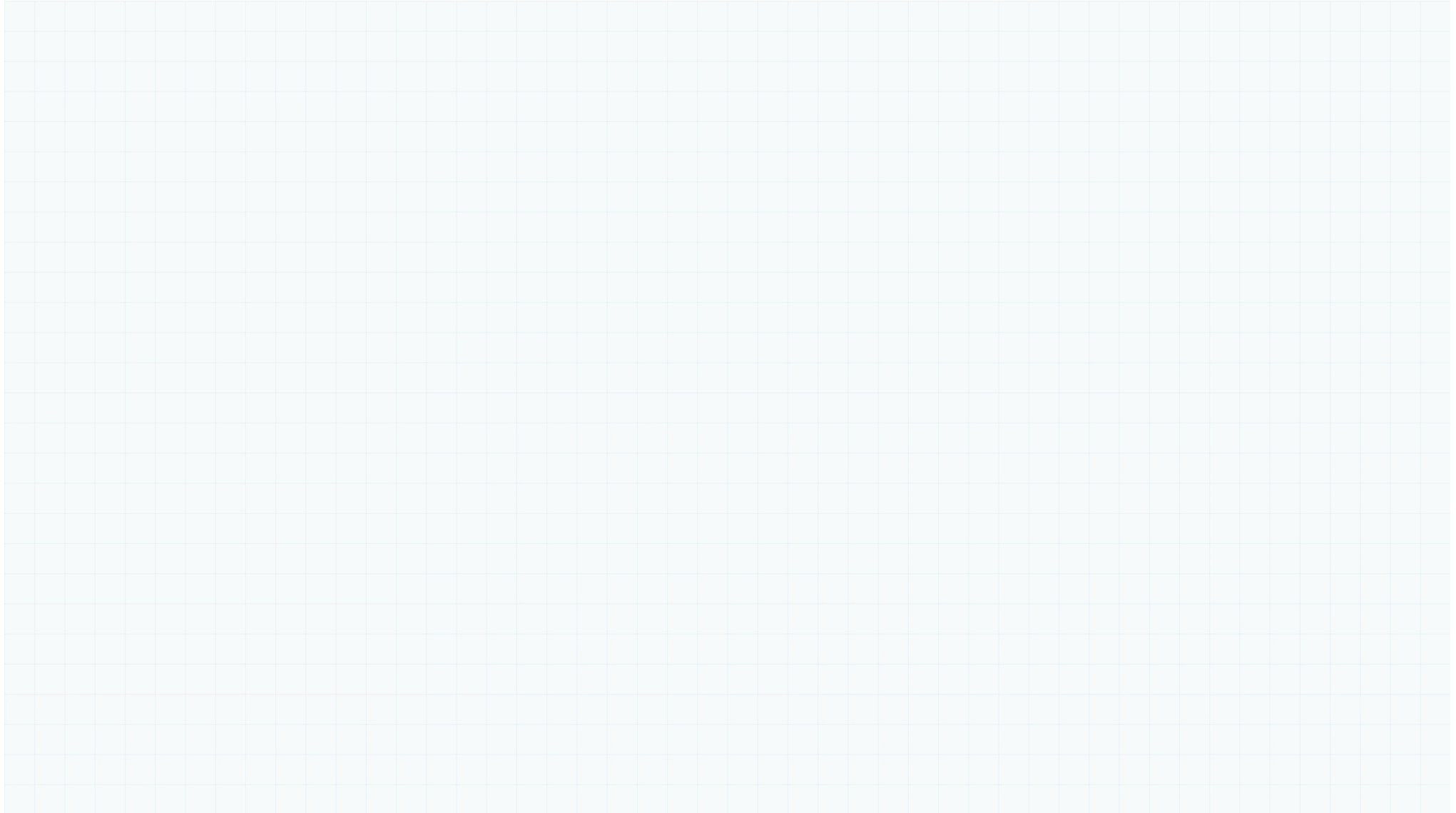
Modes:

- `abort` : Default behavior, identical to `abort` endpoint with `abort_all` set. Pending requests from `waiting_queue` and `running_queue` will be returned immediately to the caller.
- `retract` : Put engine in "paused" state. Move running requests back to waiting queue. KV cache can be flushed and recomputed later.
- `in_place` : Put engine in "paused" state without changing states of the requests. Running requests rely on availability of KV caches to continue, so any subsequent `flush_cache` call will be unsuccessful.

Continue Generation

Endpoint: `POST /continue_generation`

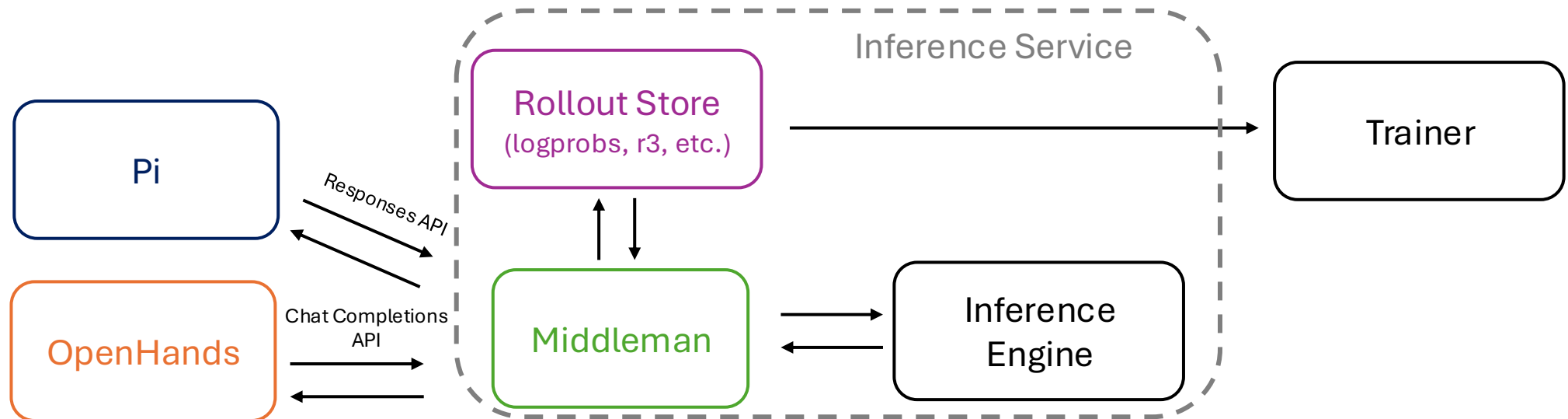
Complex Harnesses (Sub-agents, Context compaction, etc.)



System Design Choices

Harness Agnostic Design

- Many popular agent harnesses: Codex, Claude Code, OpenHands, Pi, Hermes, Prime Agent, etc.
- How to support all?
 - **Anti-pattern:** Custom rollout/environment code for each harness.
 - **Harness-agnostic pattern:** Middleman/Proxy service that emulates inference APIs (chat-completions, responses, anthropic, etc.) → 0 code changes when using new harness



Single Program Multiple Data (SPMD)

One Python program runs on every GPU rank (simplified example pseudocode below)

```
rank = dist.get_rank()
train = dist.new_group([0, 1, 2, 3])
infer = dist.new_group([4, 5, 6, 7])

if rank < 4:
    while True:
        batch = next_batch_shard(train)
        forward_backward(batch, train)
        optimizer.step()
        publish_weights_async(train)
else:
    while True:
        refresh_weights_if_ready(infer)
        prompt = next_prompt(infer)
        trace = rollout(prompt, infer)
        enqueue_trajectory(trace, infer)
```

All ranks

Create both groups
in the same order.

Ranks 0–3: training

FSDP collectives
and optimizer updates

Ranks 4–7: inference

Tensor/Context-parallel decoding
and weight refresh

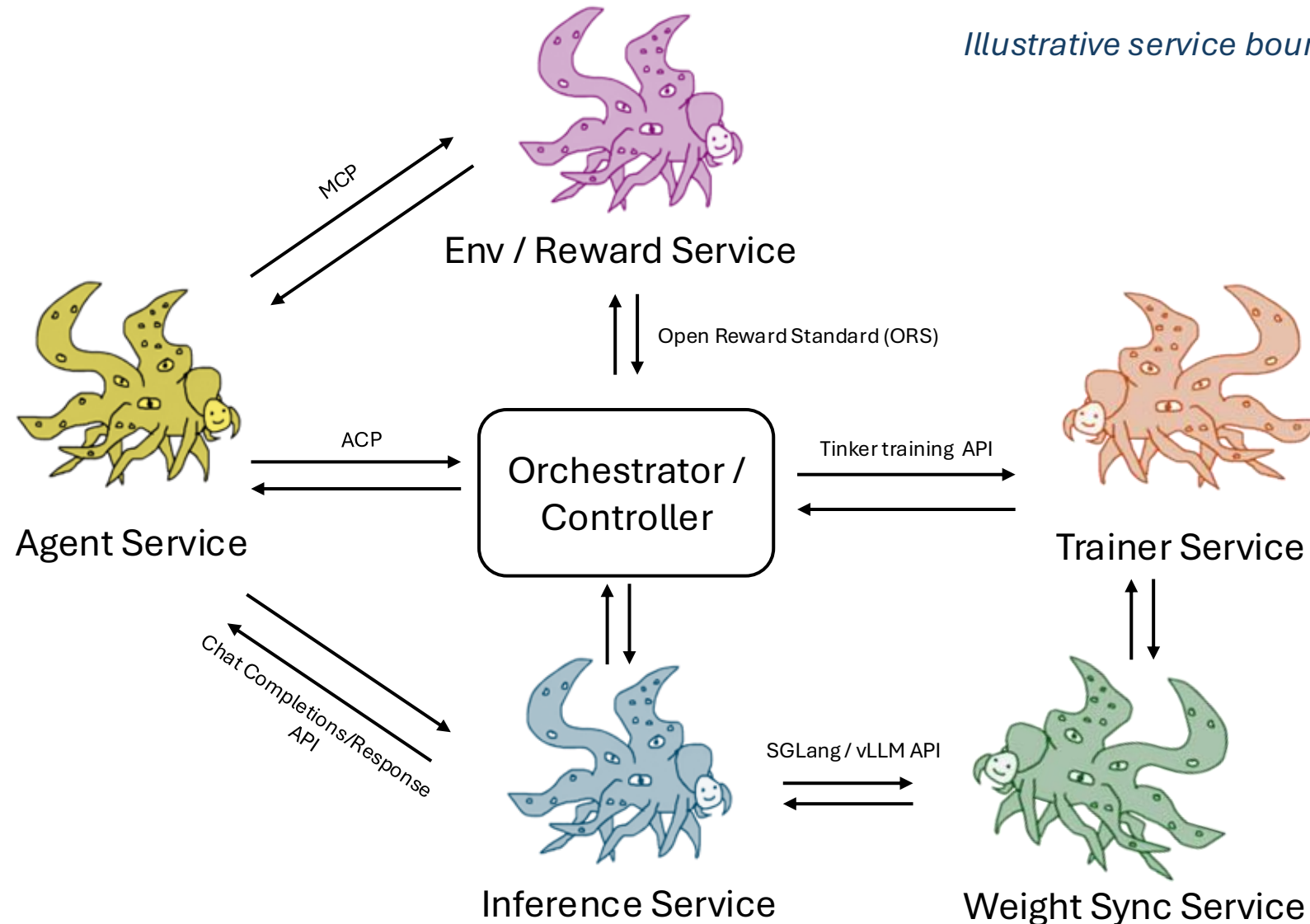
Coordination lives in the program running on every rank.

Straightforward with direct control of everything in a single program, **but less flexible.**

[AResL versions < 1.0](#)

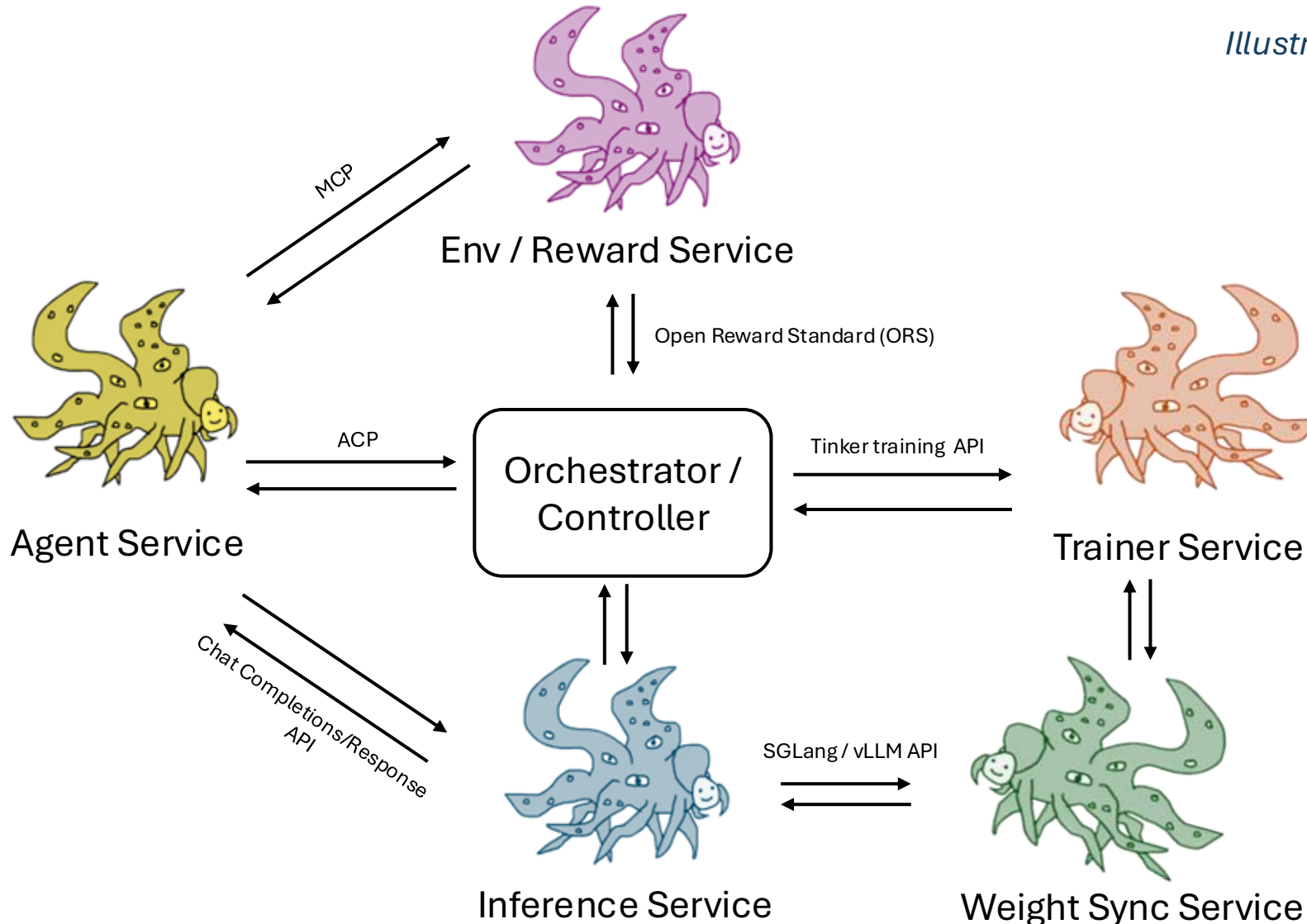
Everything-as-a-service w/ Single Controller

Illustrative service boundaries and API examples.



Everything-as-a-service w/ Single Controller

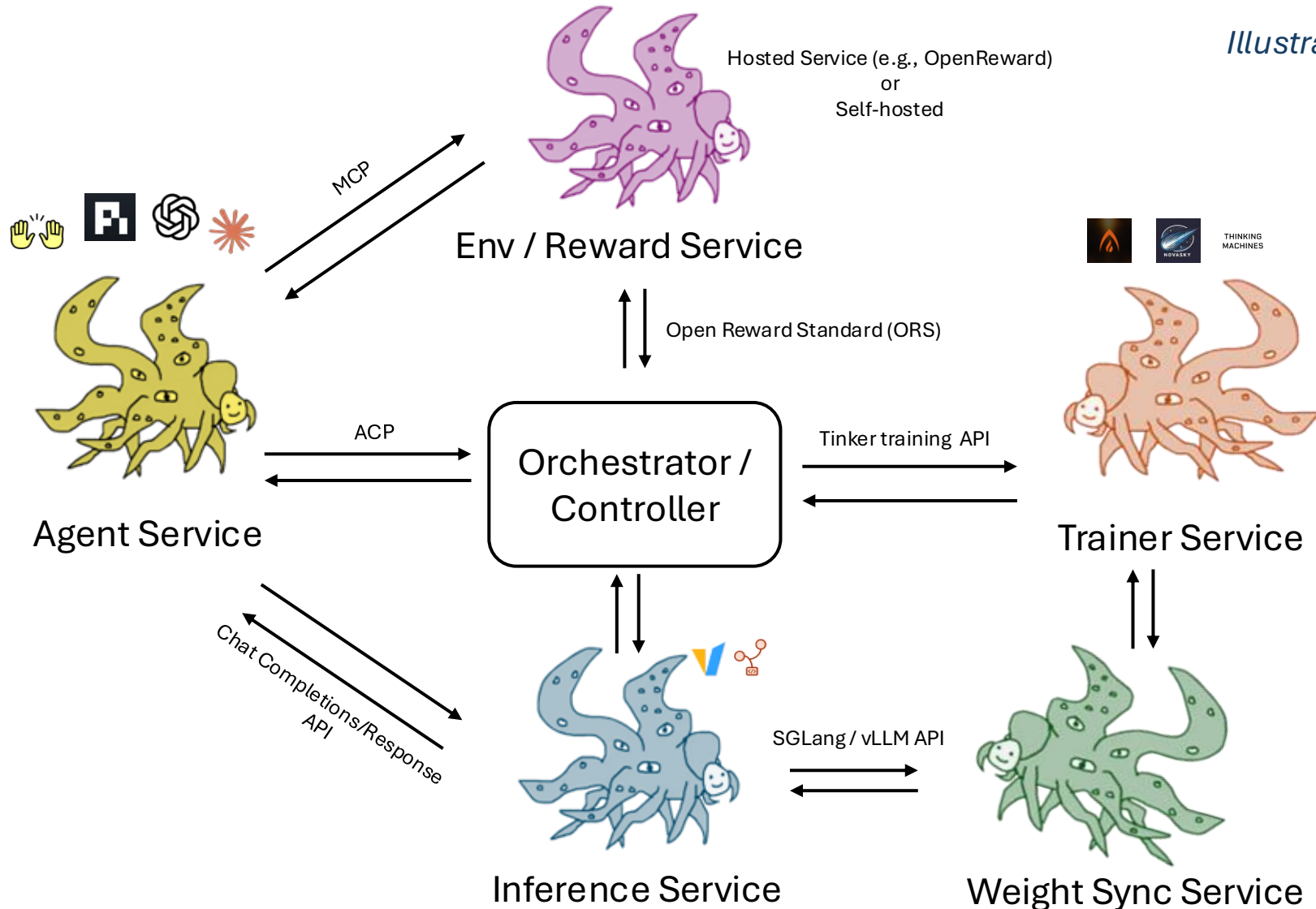
Illustrative service boundaries and API examples.



Separation of concerns (decoupling):

- Each of these components is fairly complex and specialized.
- Instead all living in the same codebase:
 - Lightweight orchestrator (CPU training loop) → researchers can easily prototype different algorithms without complex dependencies.
 - Project maintenance, updates and releases don't need to be synchronized.

Everything-as-a-service w/ Single Controller

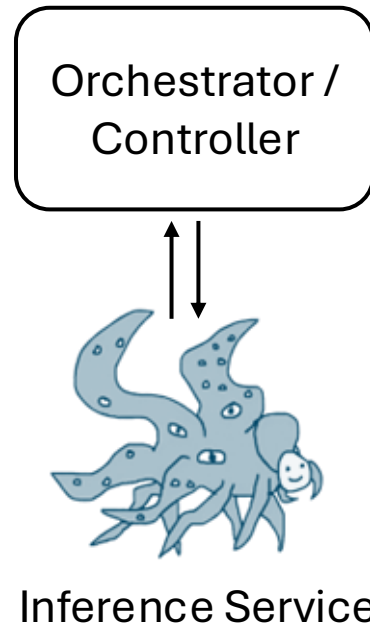


Illustrative service boundaries and API examples.

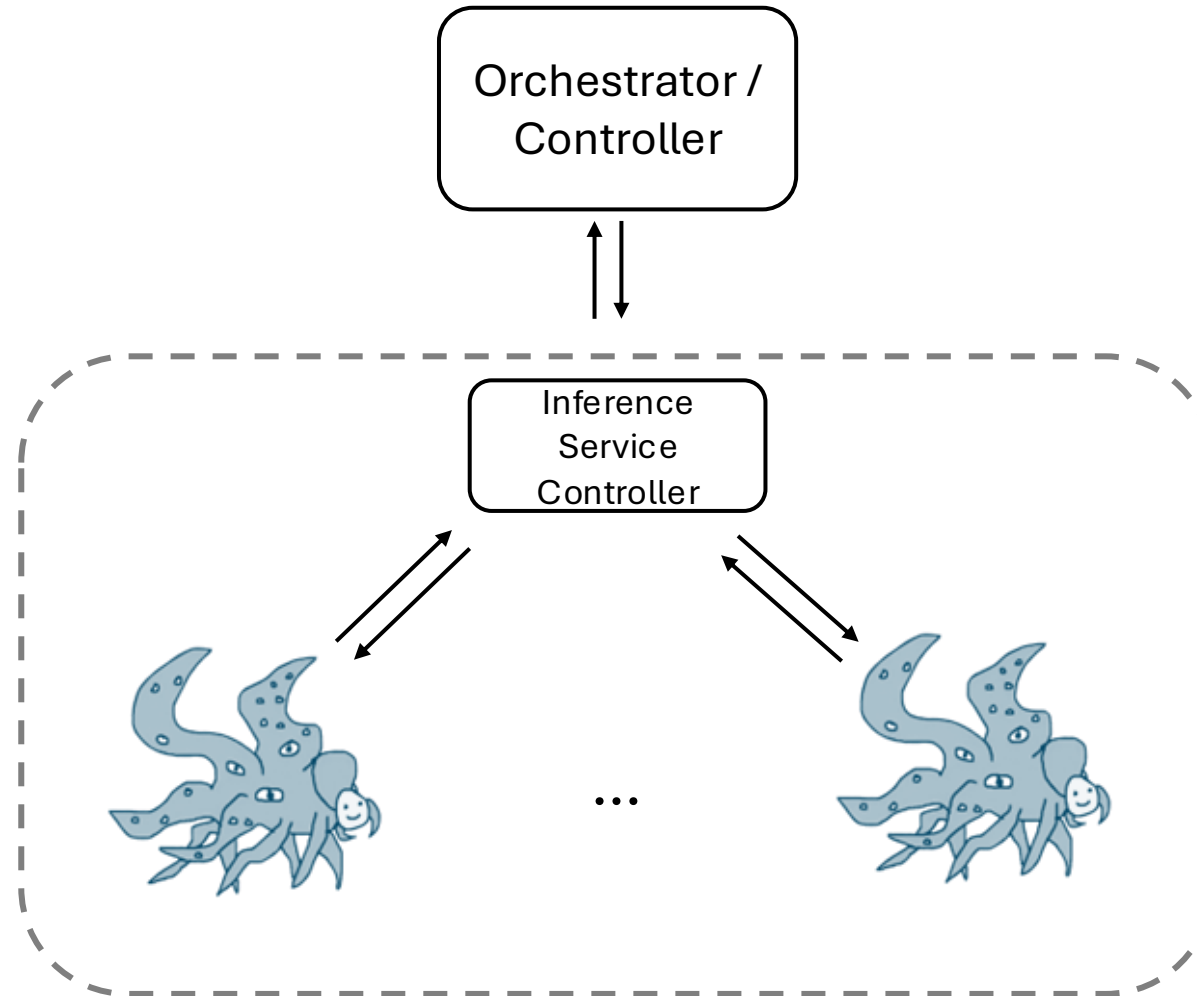
Modularity

- Easy to swap a service implementation for another one as long as the same protocols are used for inter-service communication.

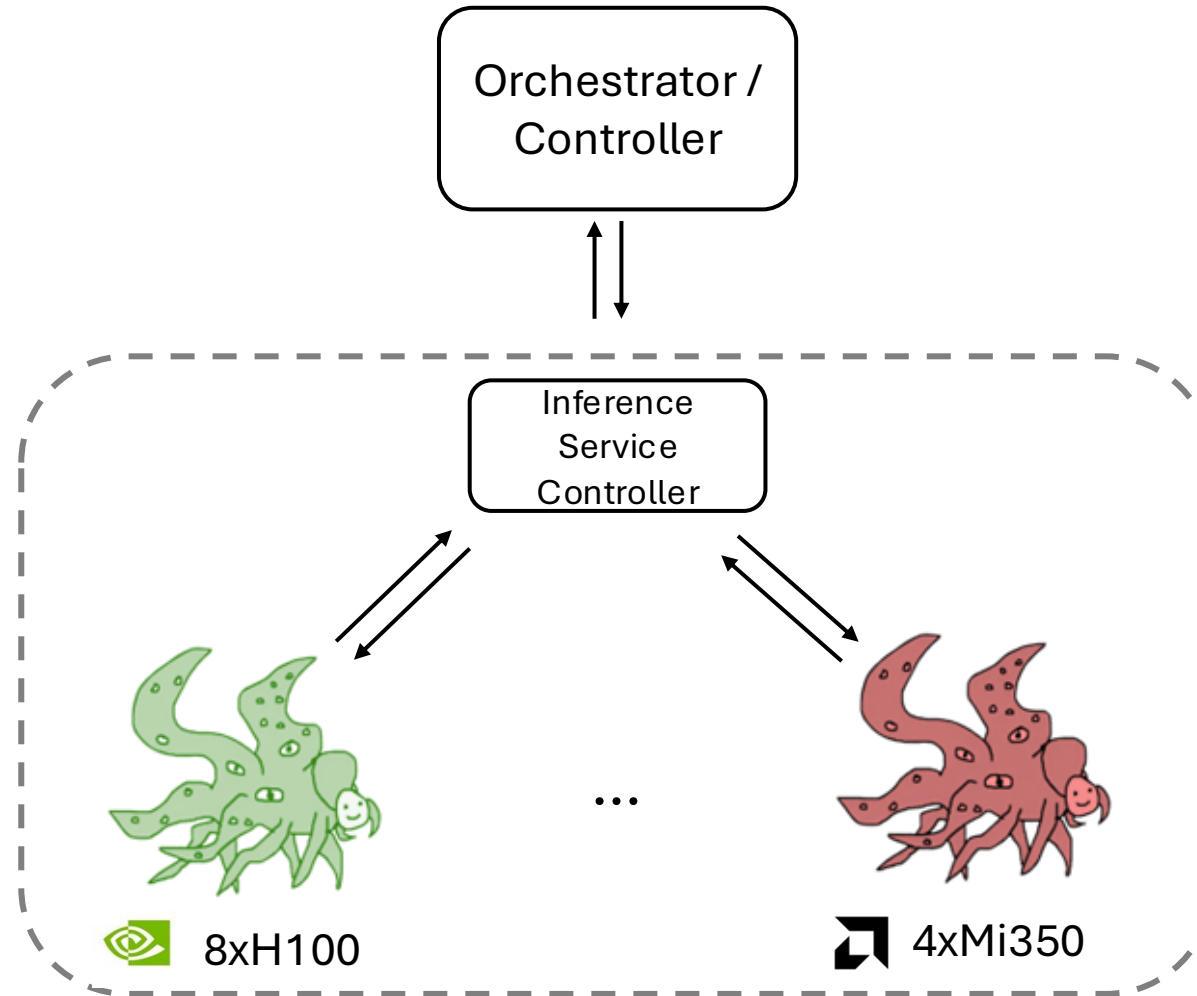
Elastic Scale Up



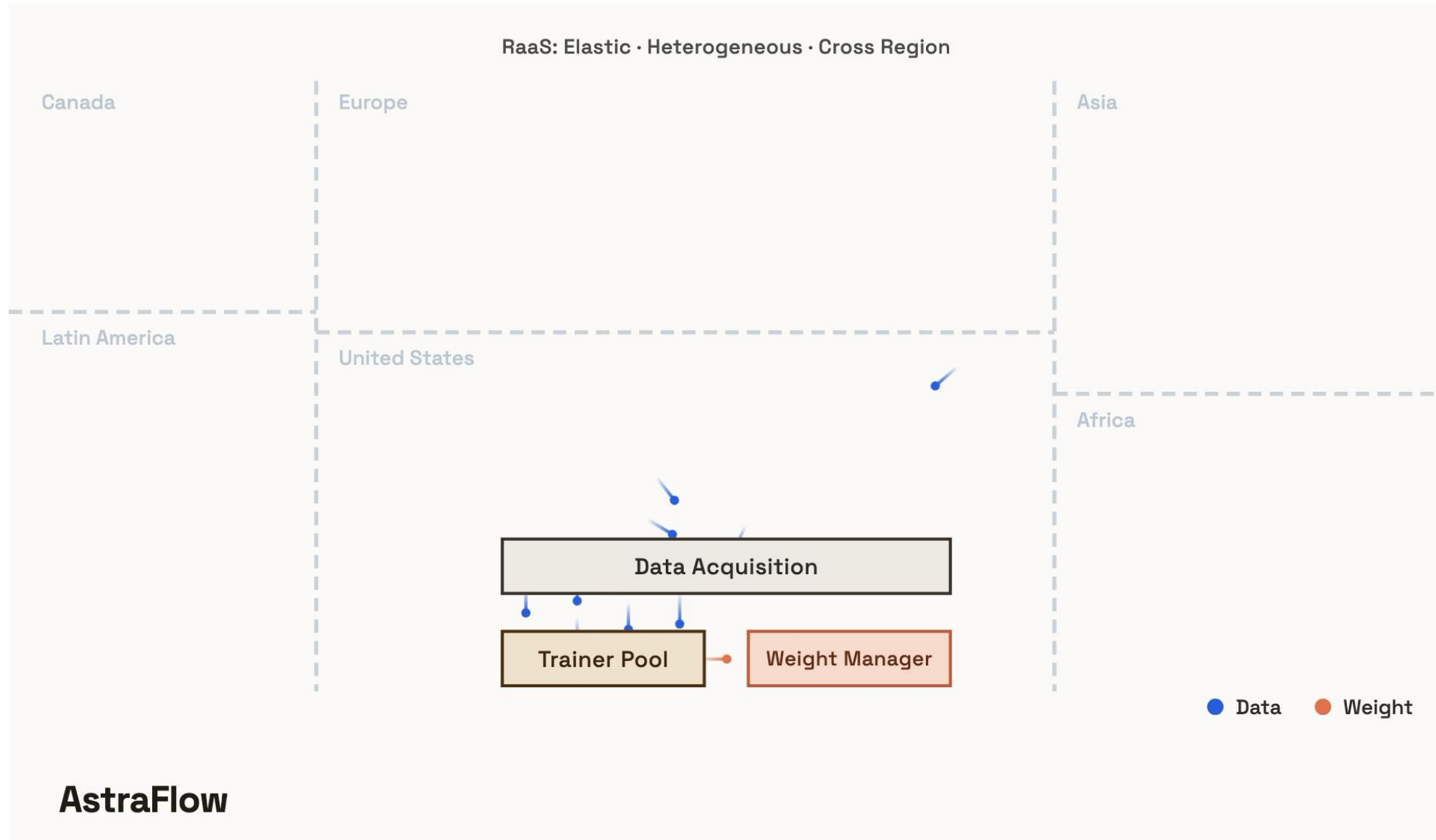
Elastic Scale Up



Multi-cluster / Heterogeneous Compute

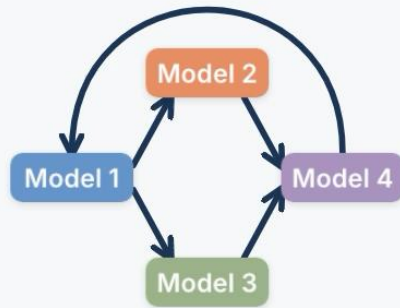


Example: AstraFlow (Haizhong Zheng et. al, 2026)



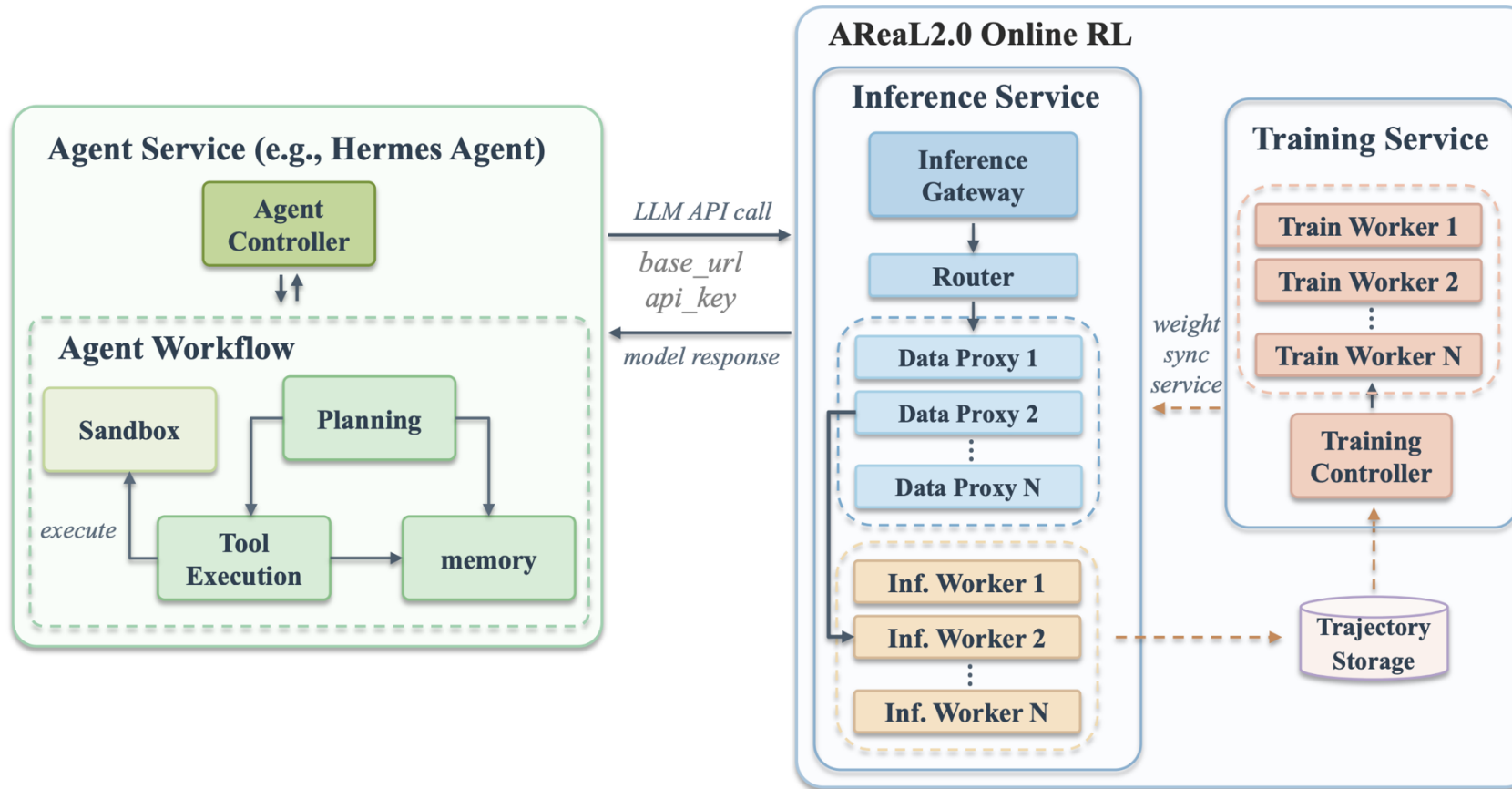
Multi-policy Training

Microservice architecture makes it easy to train multiple *different* models simultaneously (e.g., for multi-agent rollouts with heterogenous policies)



AstraFlow

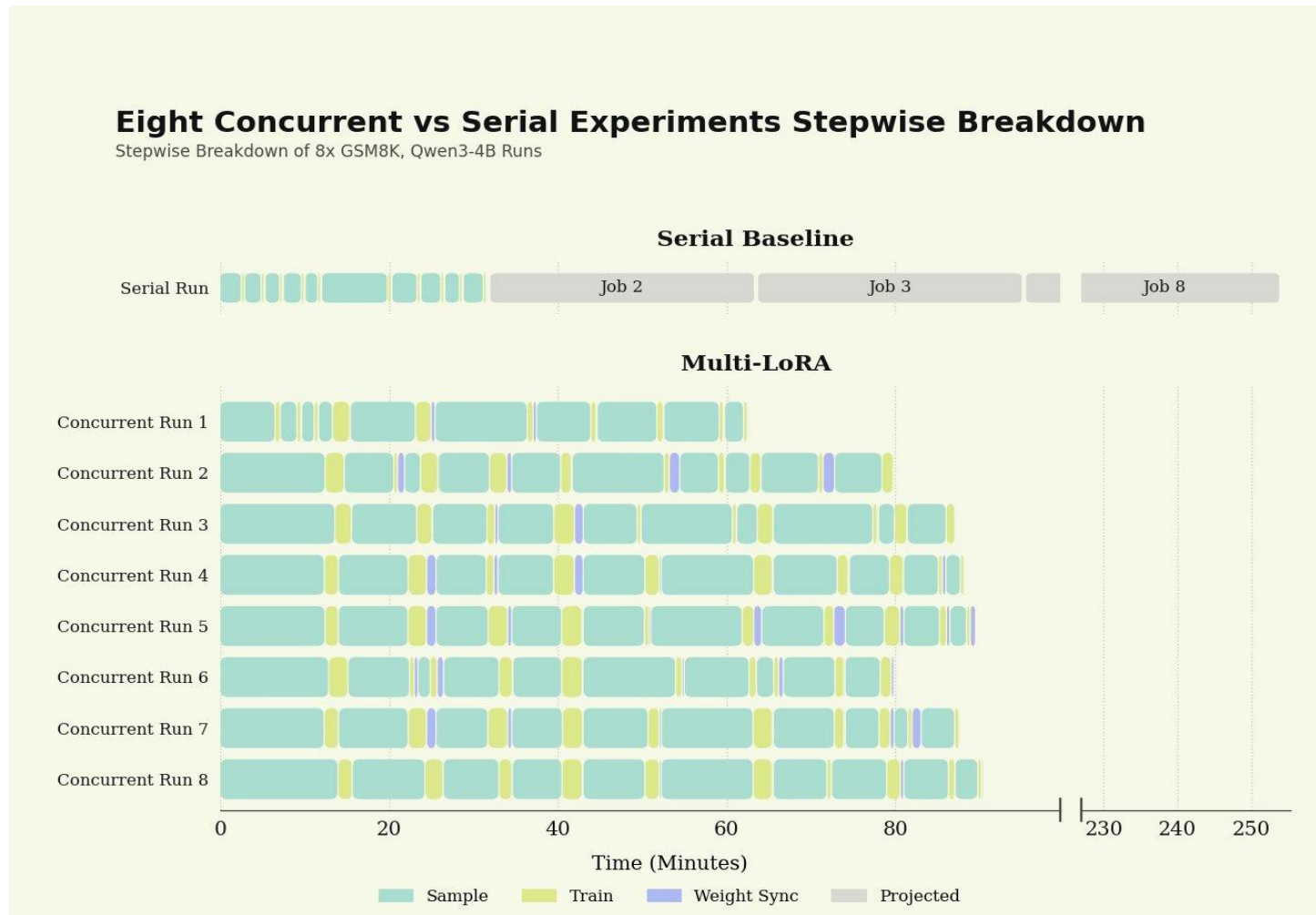
Online RL (Training after Deployment)



Online RL in production at Cursor:

- [Cursor Tab](#)
- [Composer](#) (new model checkpoint from real user interaction every 5 hours)

Multi-Tenant, Multi-LoRA Training



- 8 concurrent runs beat 8 serial runs in throughput (but with e2e time per run).
- Each concurrent LoRA run can be a different user of your training service.
- Requires special kernels Grouped-GEMM/SGMV kernels.



- Pioneered multi-tenant, multi-LoRA RL training and the decoupled training service design.
- Clean API design:
 - Training client per training run.
 - Sampling client per LoRA weight checkpoint.



`forward_backward`

Perform a forward pass and a backward pass, accumulating the gradient.



`optim_step`

Update weights based on the accumulated gradient.



`sample`

Generate tokens for interaction, evaluation, or RL actions.



`save_state`

Save training progress for resumption.



Thank you, Questions?

Next Class: Sandboxing and Credential Management

Some Useful Metrics to Track/Debug

Metric	What it tells you	Warning sign
Reward / pass@1	Is the policy improving on the training tasks?	Flat for many steps, or rising while held-out evals fall (reward hacking)
pass@k	Coverage: can any of k samples solve it?	Falls while pass@1 rises: diversity is collapsing
Entropy	How much the policy still explores	Sudden collapse toward 0, or a jump (degenerate text)
importance_weight_max	Worst token ratio, e.g. $\pi_{\text{train}} / \pi_{\text{rollout}}$	Spikes: mismatch or stale data; a few tokens dominate the update
Grad norm	Size of each update before clipping	Spikes ahead of divergence; steady upward drift
Average staleness	How many weight versions old the data is	Creeping up: rollouts can't keep pace with training
Trainer MFU	How well the trainer GPUs are used	Drops from padding, imbalance or small microbatches
Timing: rollout, update, stall, total	Where the wall-clock time of a step goes	Stall > 0: the trainer is waiting on rollouts