DynaRL: Flexible and Dynamic Scheduling of Large-Scale Reinforcement Learning Training

Yuanqing Wang, Hao Lin, Junhao Hu, Chunyang Zhu, Quanlu Zhang, Zhen Guo, Yuchen Zhang, Xu Fu, Si Xu, Bo Dai, Zixiao Huang, Chao Yu, Boxun Li, Guohao Dai, Zhi Yang, Yu Wang

2026/8/4

https://github.com/RLinf/RLinf

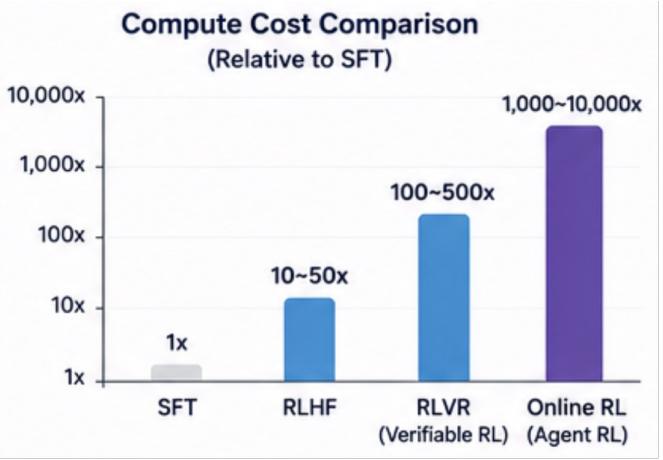
wangyuanqing@pku.edu.cn

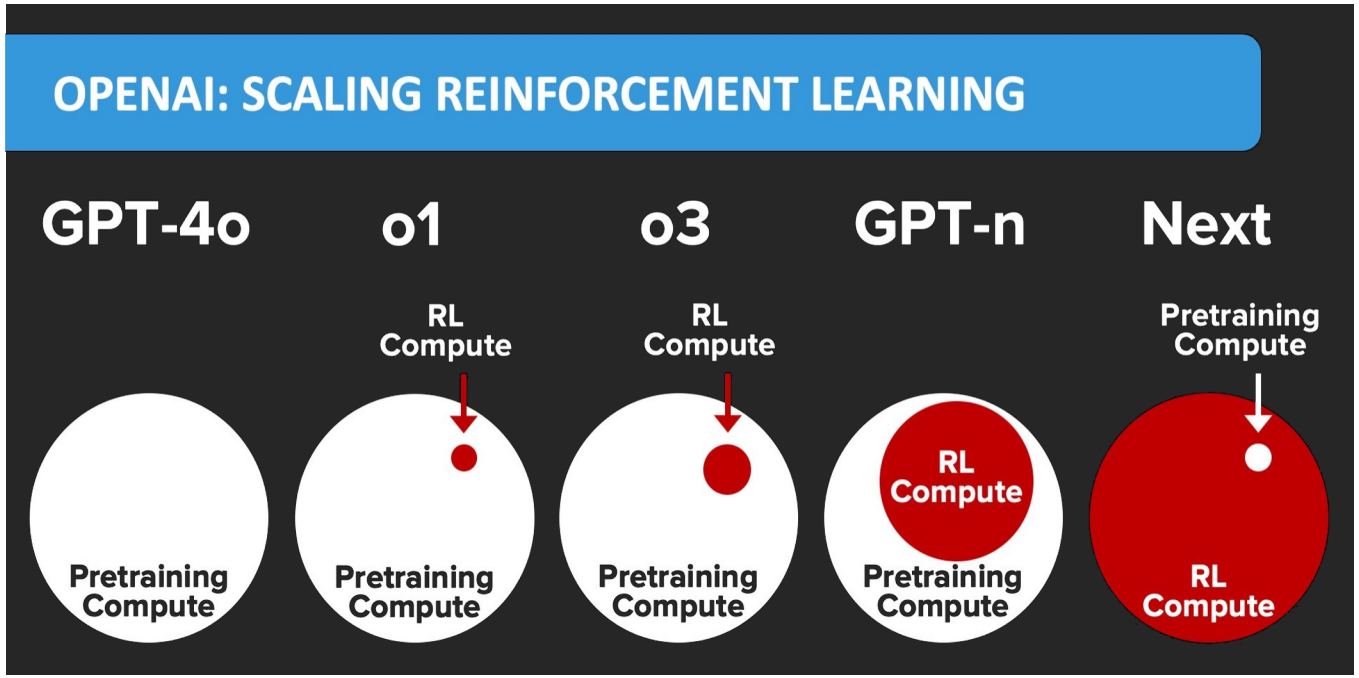
RL Is Now the Backbone of Frontier AI

❑ RL trains today's most capable reasoning & agentic models

OpenAI o1

deepseek R1





❑ RL requires an ever-increasing amount of resources.

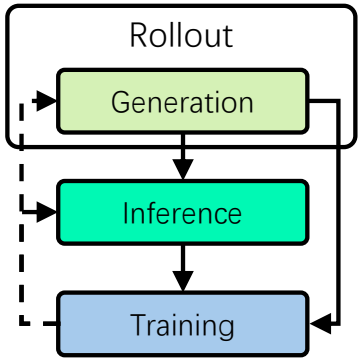
Codex

CLAUDE CODE

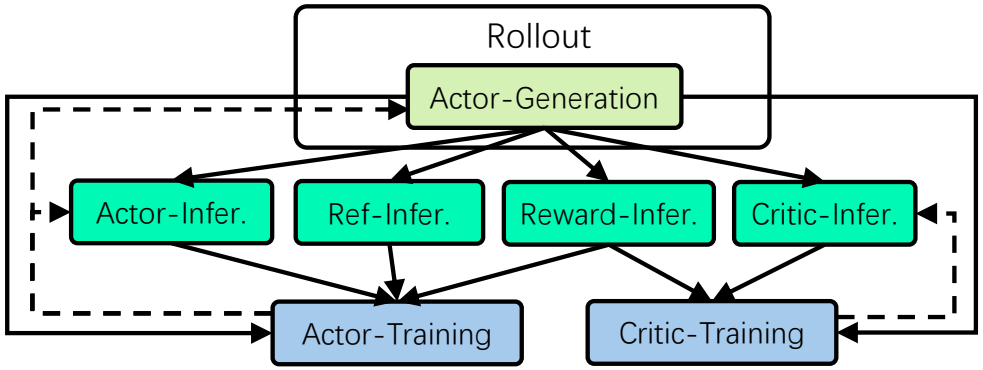
❑ RL is a multi-component pipeline

RL Workloads are Complex and Interactive

❑ RL workflow is composed of multiple components



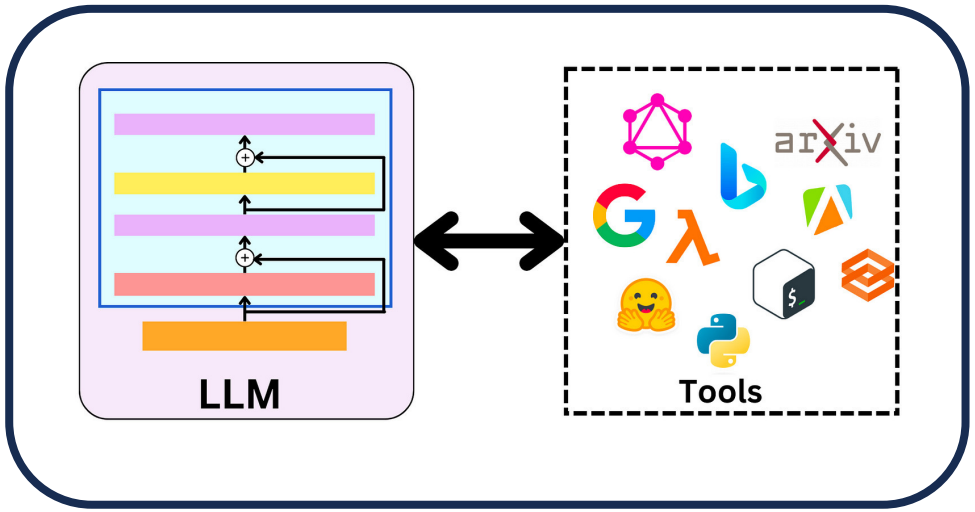
GRPO workflow

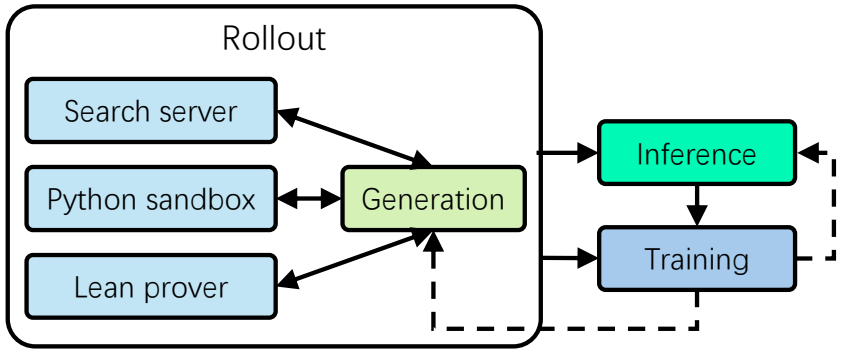


PPO workflow

Data transmission Model weight update

❑ Agentic RL is highly interactive

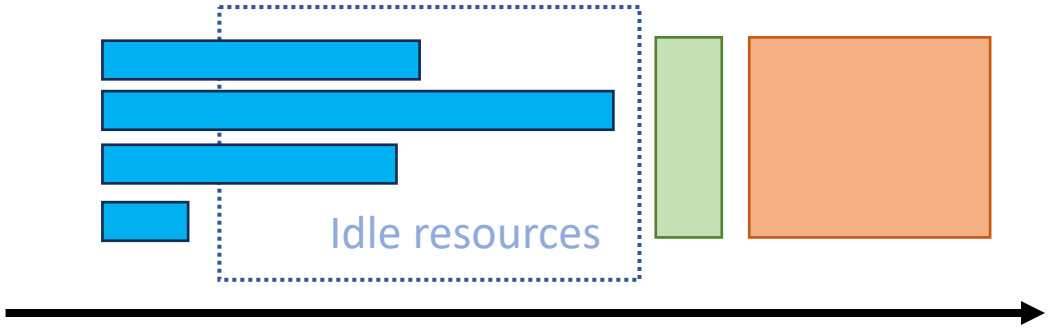


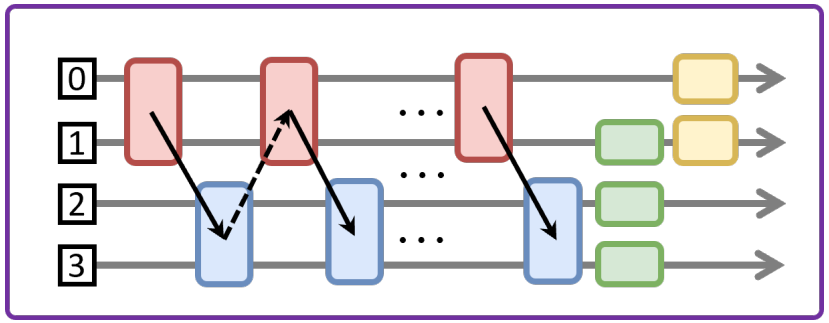


Agentic RL

Mismatch between rigid placement and dynamic patterns

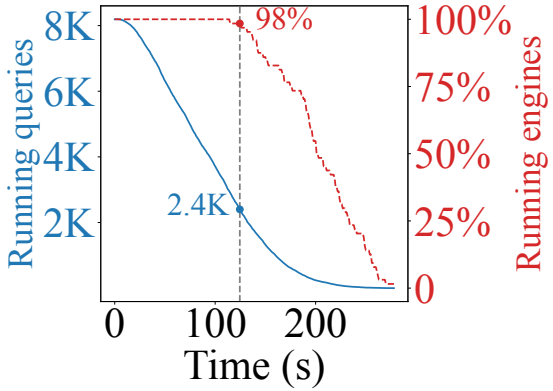
❑ The Dynamicity Challenge of RL





Long-tail rollout generation

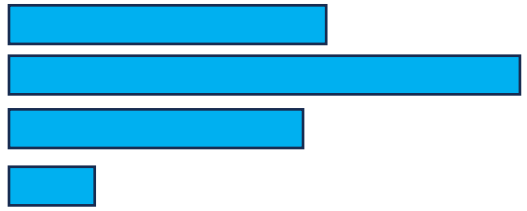
Existing frameworks use a fixed placement strategy

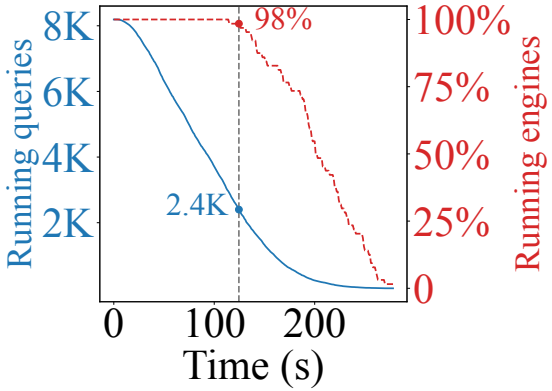


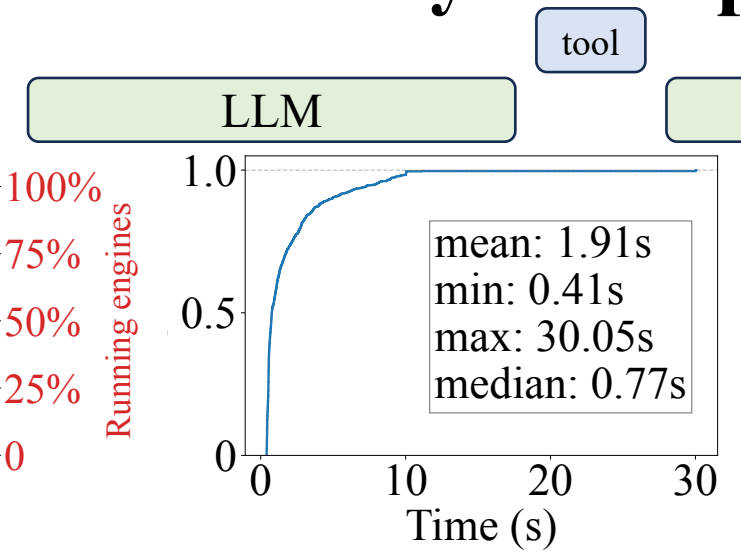
Resource: static allocation wastes resources

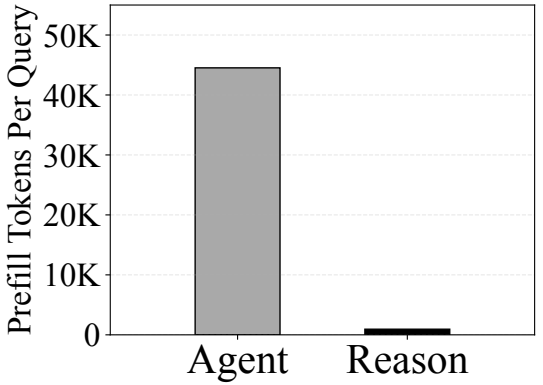
Mismatch between rigid placement and dynamic patterns

❑ The Dynamicity Challenge of RL









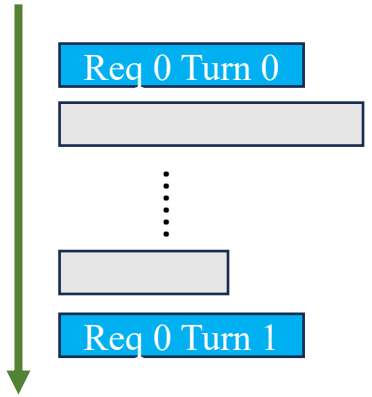
Long-tail rollout generation

Irregular external interactions

❑ Existing frameworks use a fixed placement strategy

Resource: static allocation wastes resources

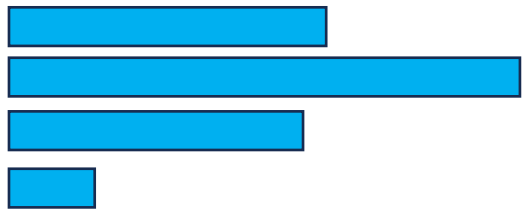
FIFO execution order

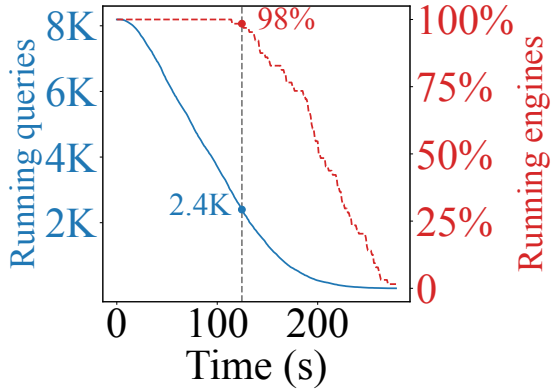


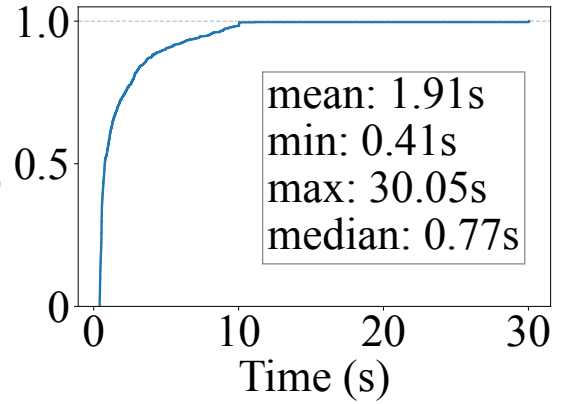
Data: A naïve data ordering leads to unnecessary recomputation

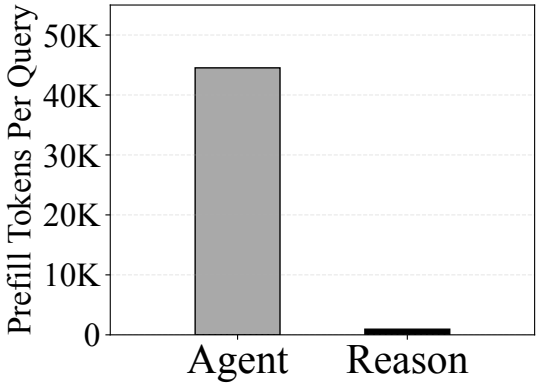
Mismatch between rigid placement and dynamic patterns

❑ The Dynamicity Challenge of RL









Long-tail rollout generation

Irregular external interactions

❑ Existing frameworks use a fixed placement strategy

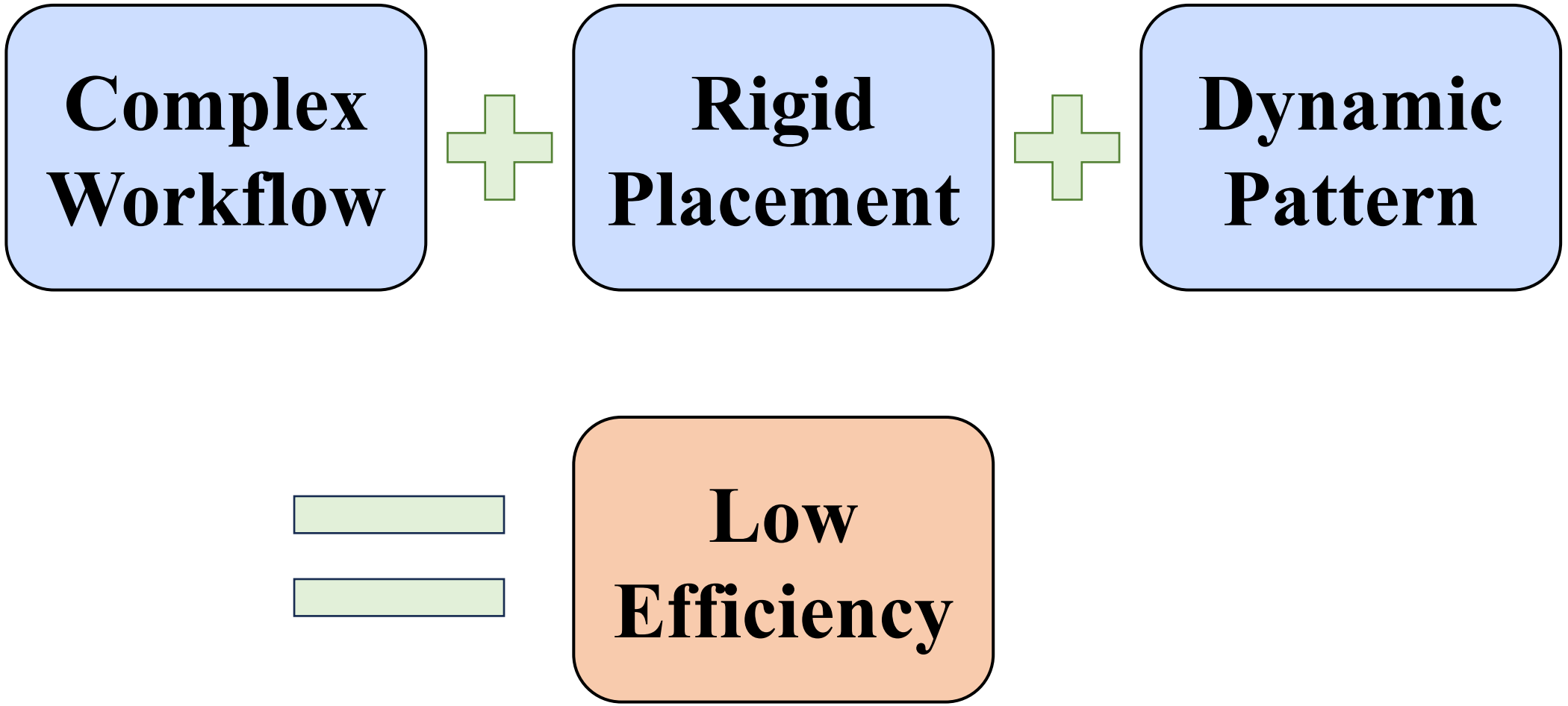
Resource: static allocation wastes resources

Rigid Placement

Data: A naïve data ordering leads to unnecessary recomputation

Dynamic Pattern

Easy-to-use abstraction? Flexible online adjustment? Real-time monitor?

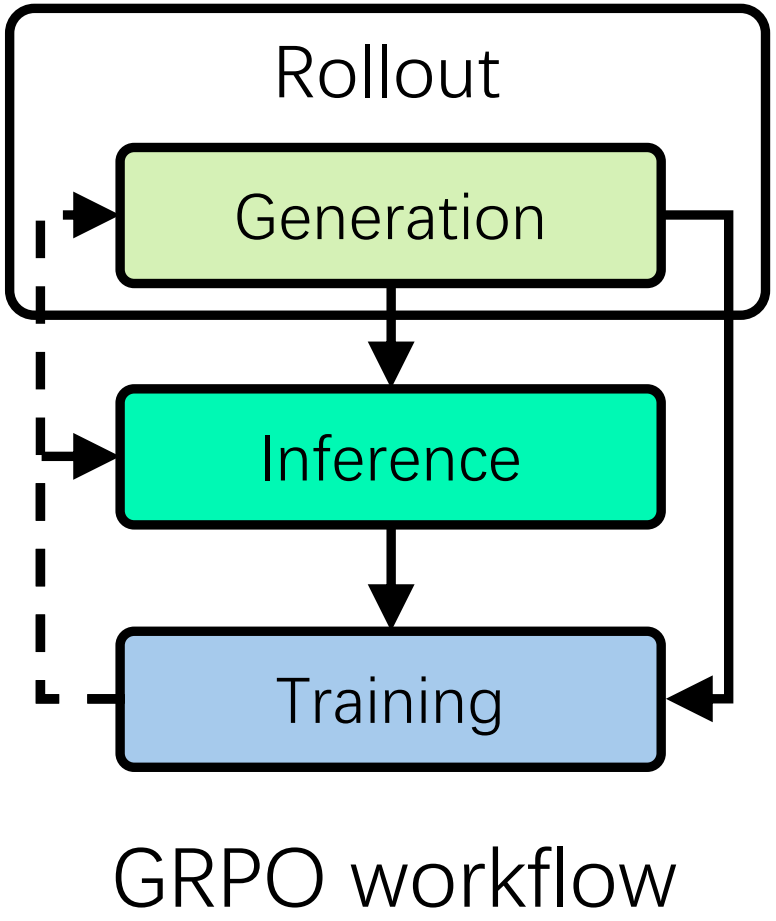


An RL system must be able to dynamically coordinate resources!

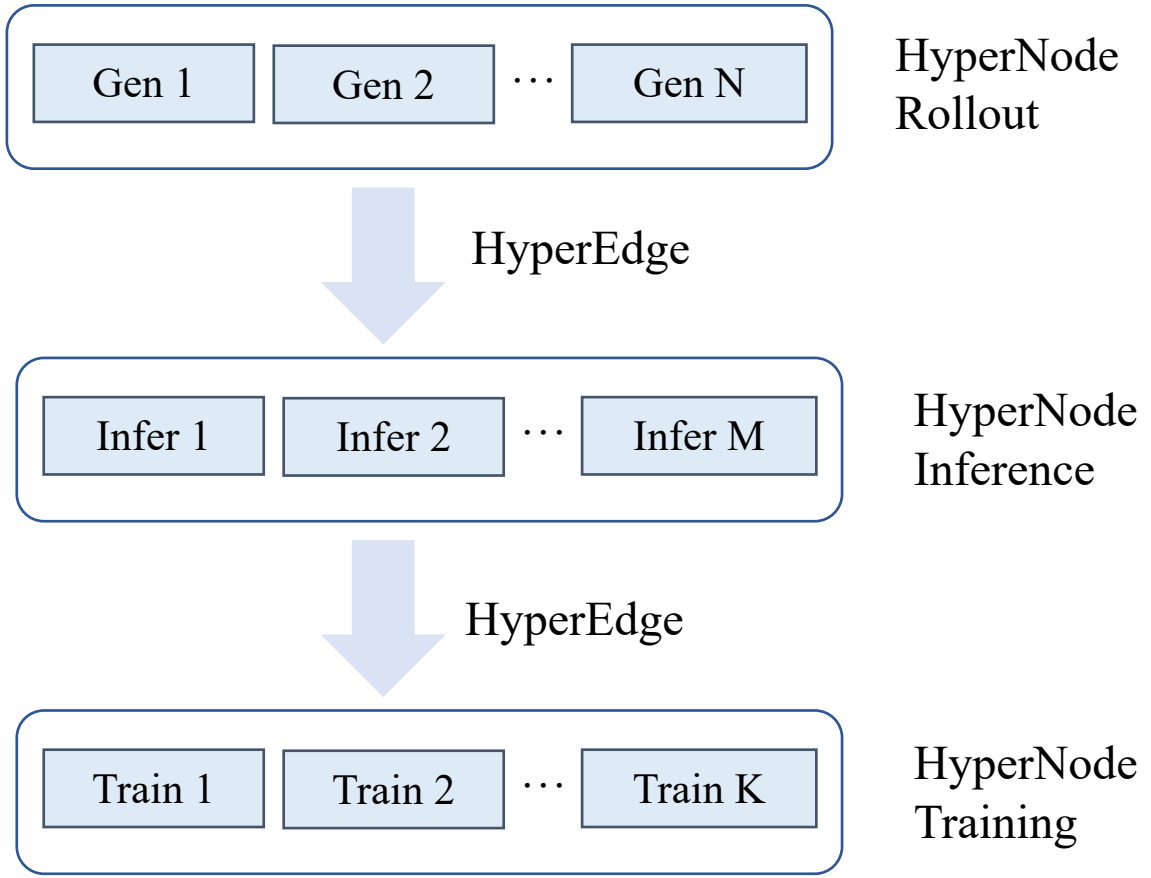
Contributions

- **DynaRL**: A highly flexible RL framework with Dynamic Scheduling
  - **First RL system to jointly reallocate the resources of all RL components at runtime.**
  - Dynamic HyperGraph to abstract RL tasks
  - Unified Interface to facilitate resource migration
  - Context-Aware Data Router to preserve locality
  - **Multi-level scheduling** driven by overprovisioning
  - Proven at scale: Up to 1.98× end-to-end speedup on 128 H100s
  - Fully open-sourced at https://github.com/RLinf/RLinf

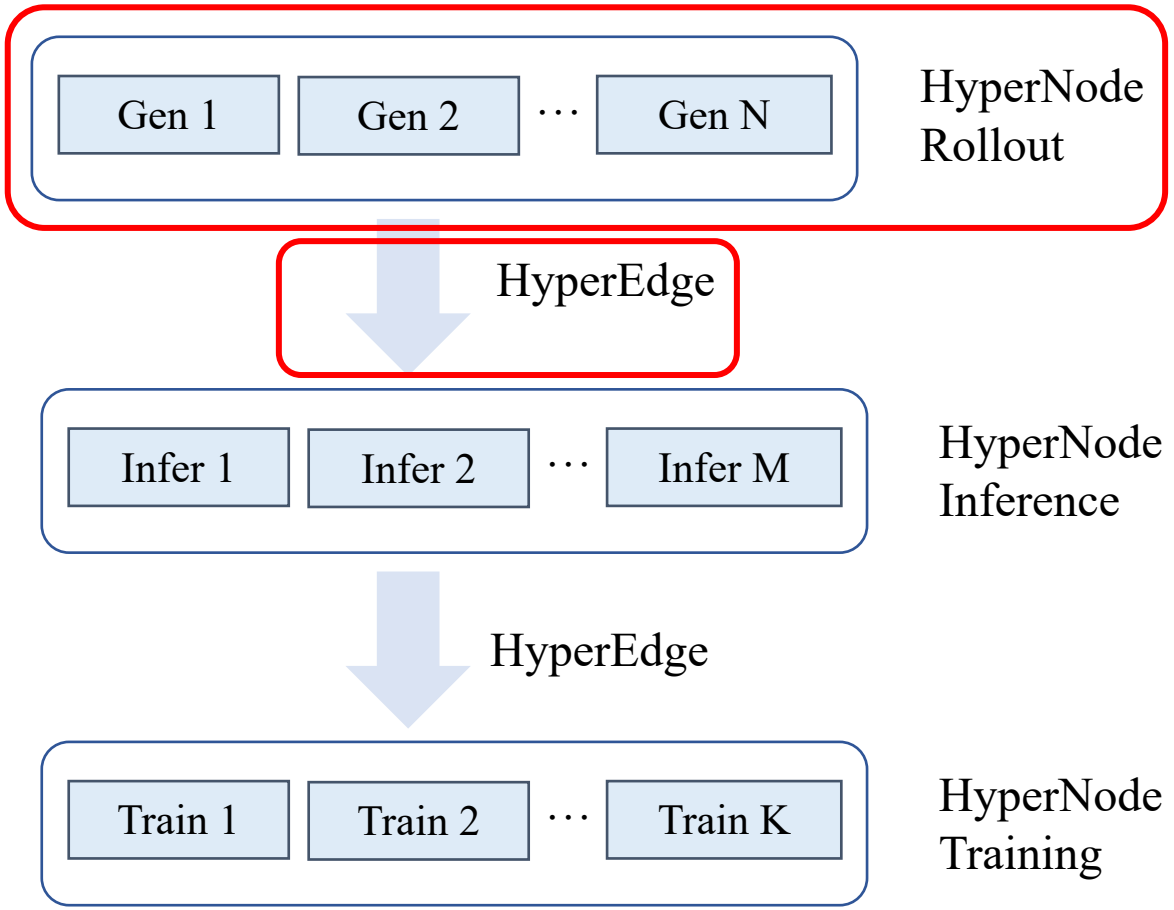
Abstraction: Dynamic HyperGraph



GRPO workflow

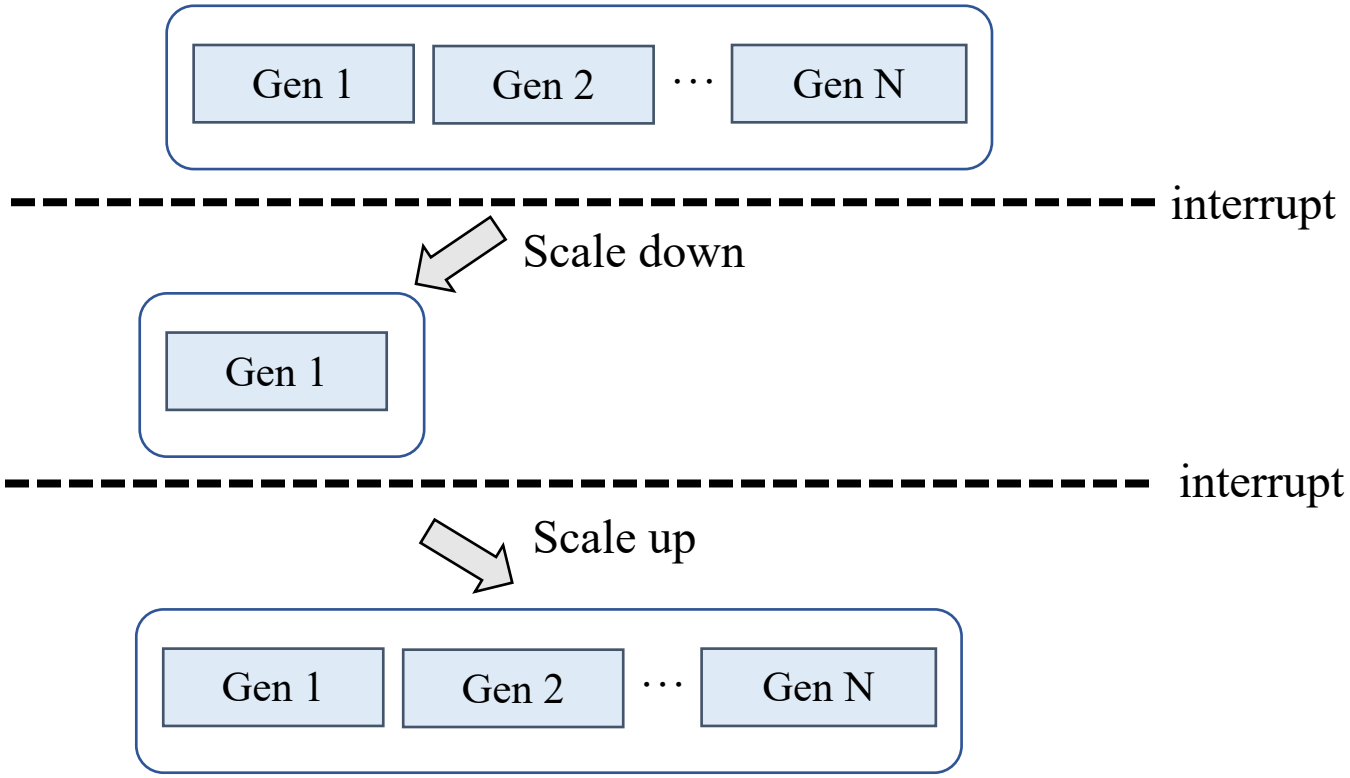


Abstraction: Dynamic HyperGraph



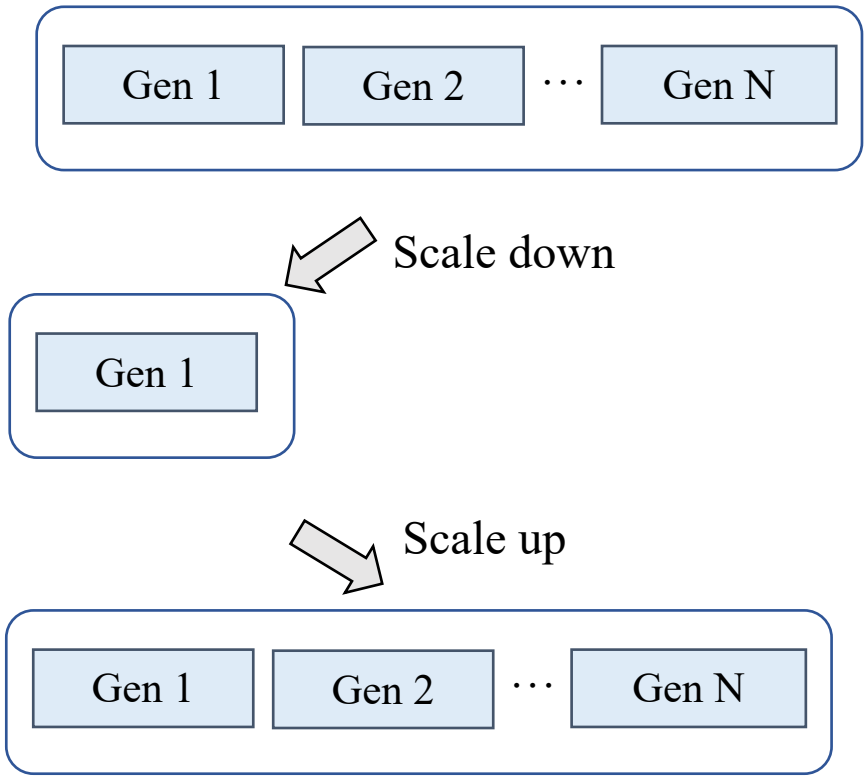

```
class HyperNode:
    # static node attributes
    component_type    # dynamic or static
    interruptible     # always, regular, or never
    predecessors: HyperNode[]
    successors: HyperNode[]
    # dynamic node status
    nodes: Node[]     # list of homogeneous nodes
class Node:
    allocated_resources: Resource[]
    utilization: number
class HyperEdge:      # dataflow between nodes
    affinity: Node[] # the preferred nodes
    progress          # data processing progress
#dynamic schedule rollout/train_workers
dynamic_schedule(graph: HyperNode[], resource:
 ↪  Resource[])
```

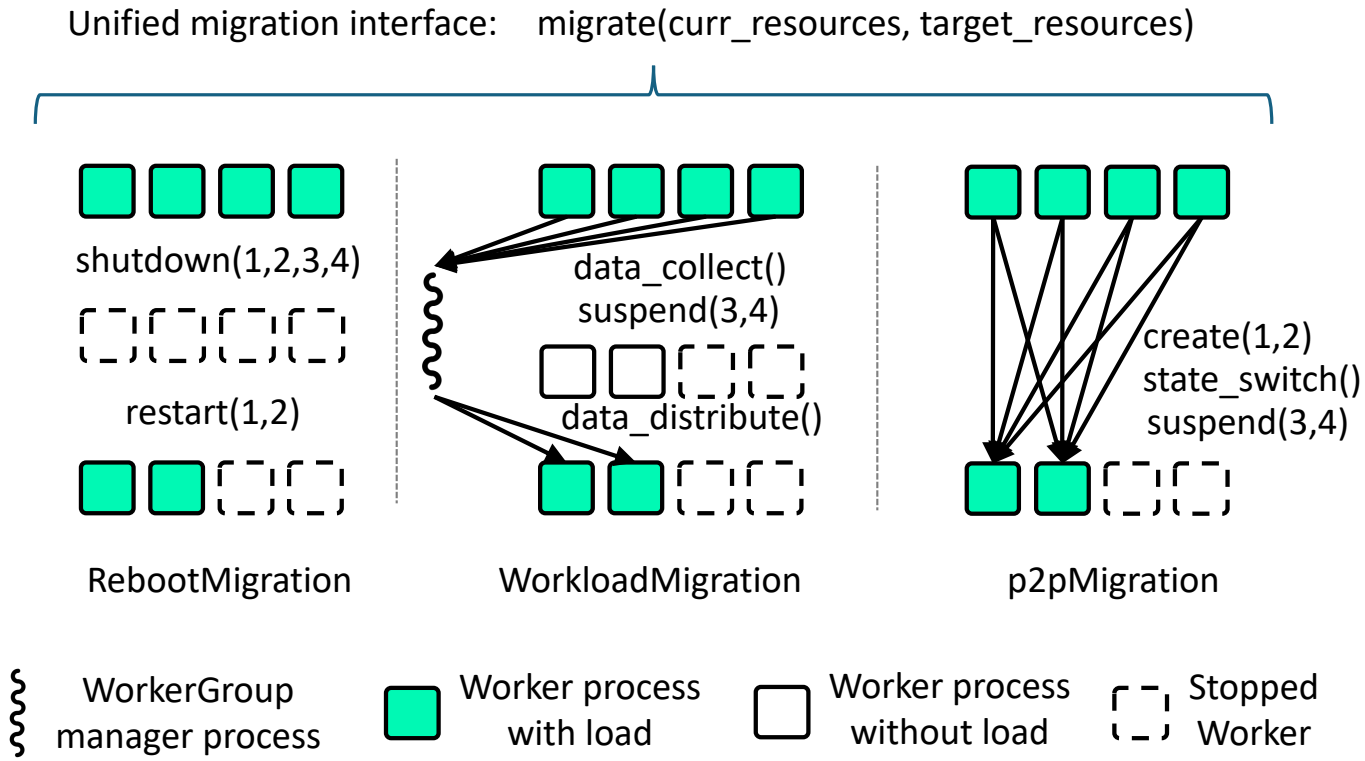
Resource: Unified Interface to Facilitate Resource Migration




```
# The unified migration interface
class MigrationManager:
    def migrate(dest_resources)
    def interrupt()
```

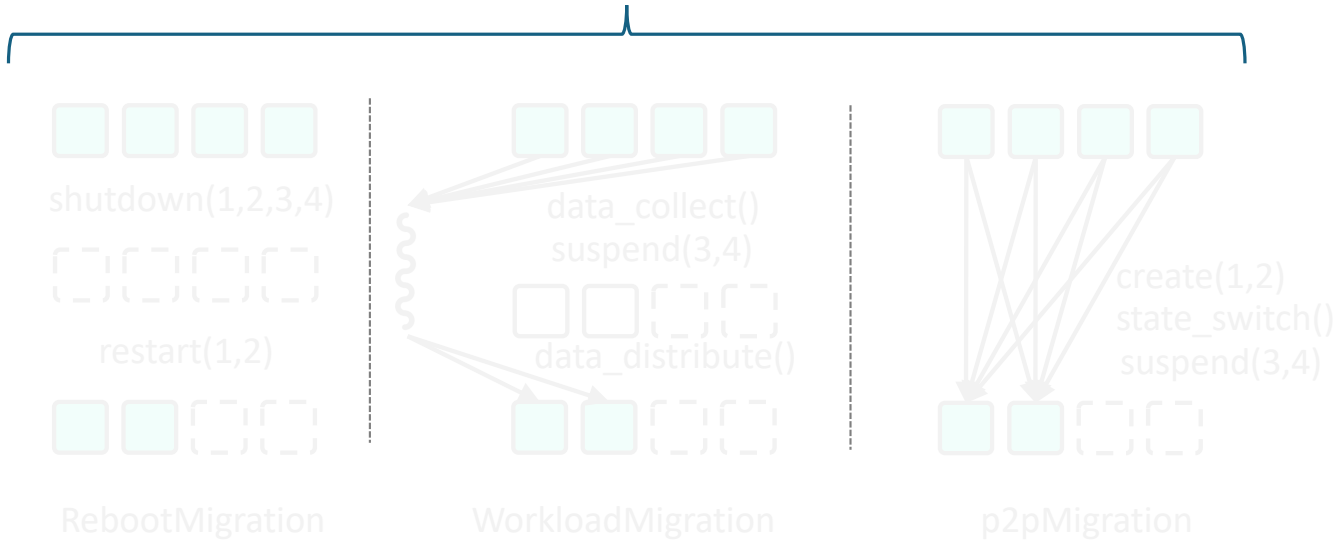
Resource: Unified Interface to Facilitate Resource Migration





Resource: Unified Interface to Facilitate Resource Migration

Unified migration interface: migrate(curr_resources, target_resources)




```python
# Migration strategy interfaces
class RebootMigration:
    def shutdown()
    def restart(dest_resources)
class WorkloadMigration:
    def data_collect()
    def data_distribute()
    def suspend()
class p2pMigration:
    def create()
    def state_switch()
    def suspend()
```

WorkerGroup manager process

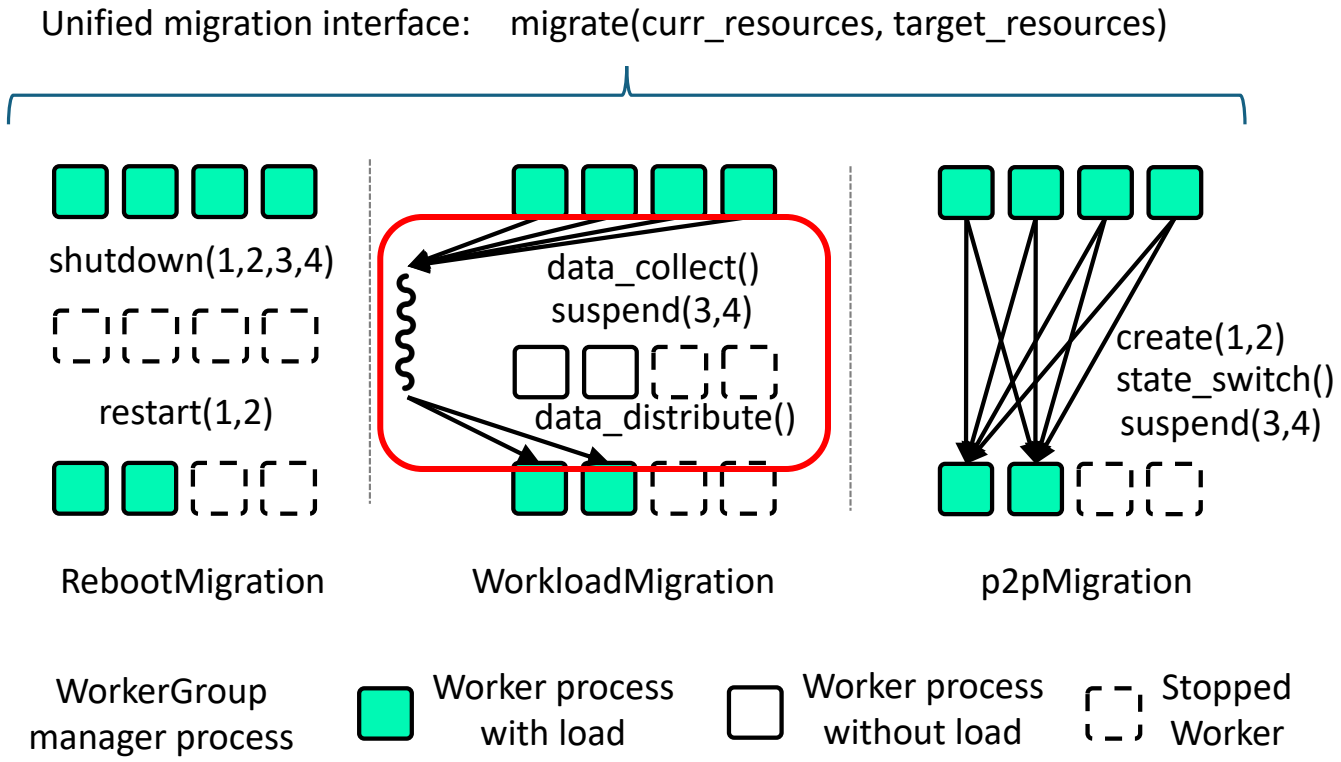
Worker process with load

Worker process without load

Stopped Worker

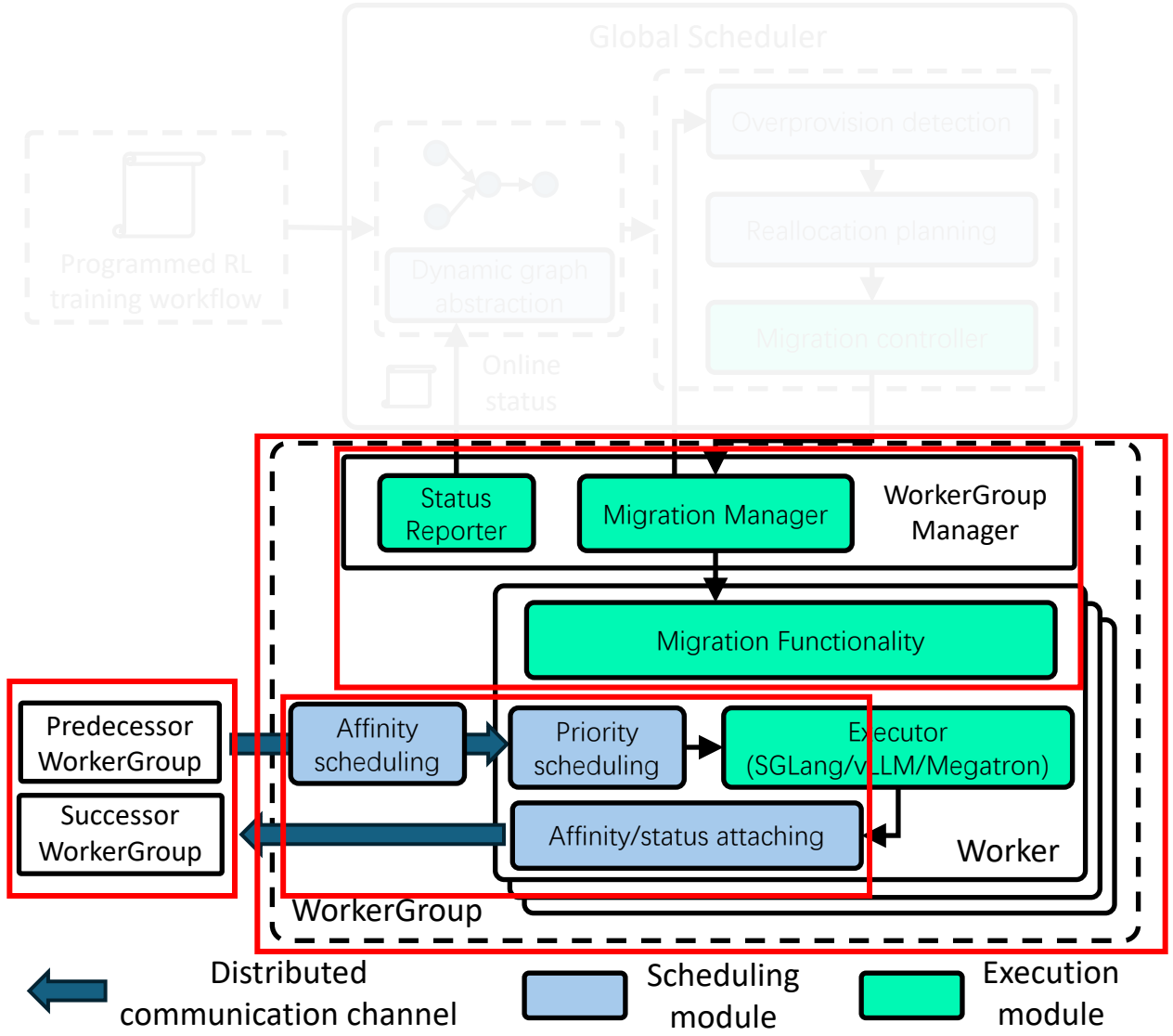
Data: Context-Aware Data Router to Preserve Locality

- Every data item carries where it's been and how far it's gotten.




```python
class DataRouter:
    # HyperEdge
    # affinity: required, preferred, noneed
    def attach_affinity(output, affinity)
    # status: steps, etc
    def attach_status(output, status)
    # requests scheduling
    def distribute_data(data)
    def priority_schedule(data)
```

DynaRL: System Overview

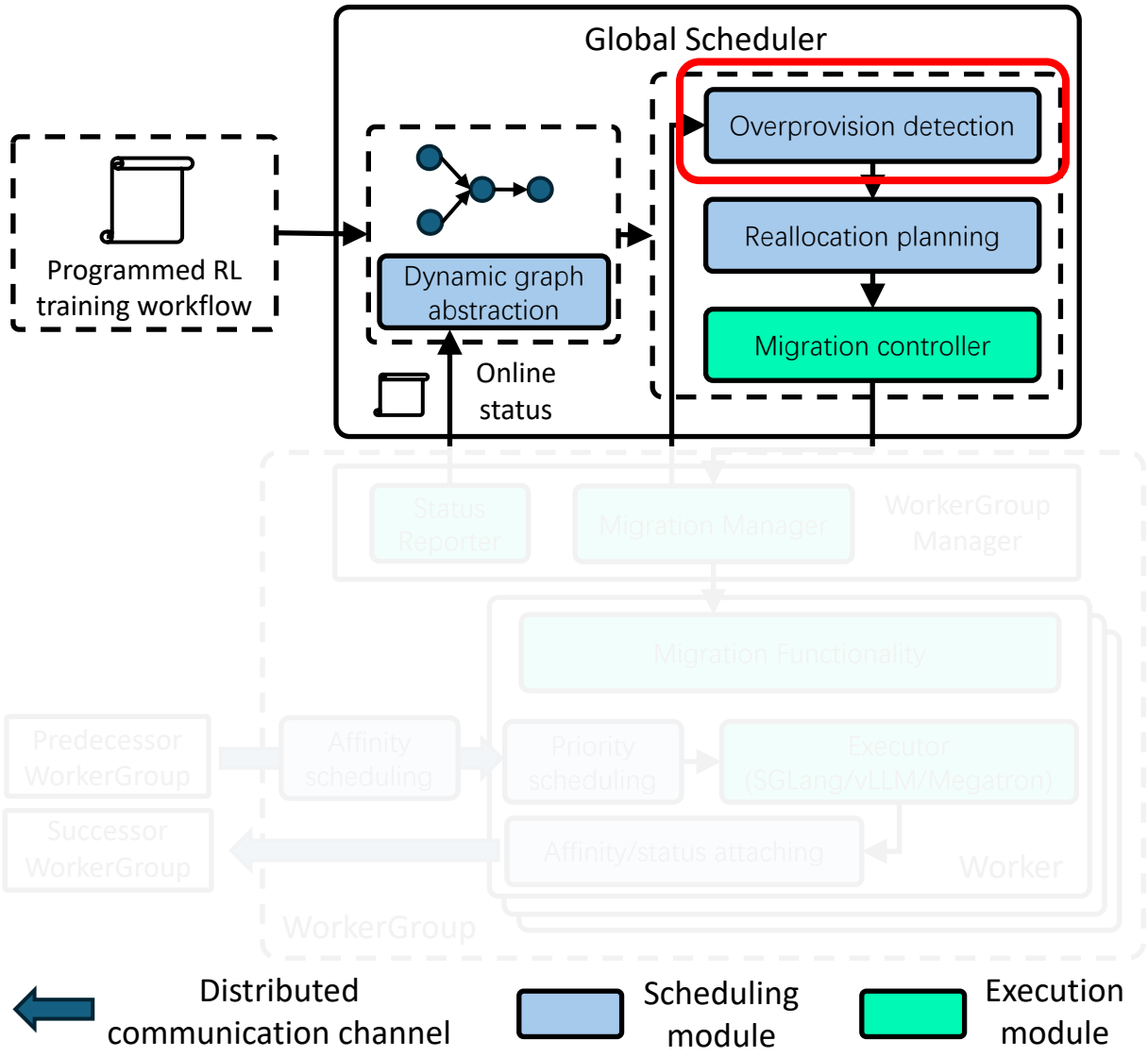


Resource-Level Scheduling : Overprovision-Driven

- Signal: resource overprovisioning
- A component is overprovisioned when:
  - adding GPUs → negligible throughput gain
  - removing a few GPUs → negligible loss

Static components
(trainer: throughput ∝ GPUs)

Dynamic components
(rollout: diminishing returns as trajectories complete)

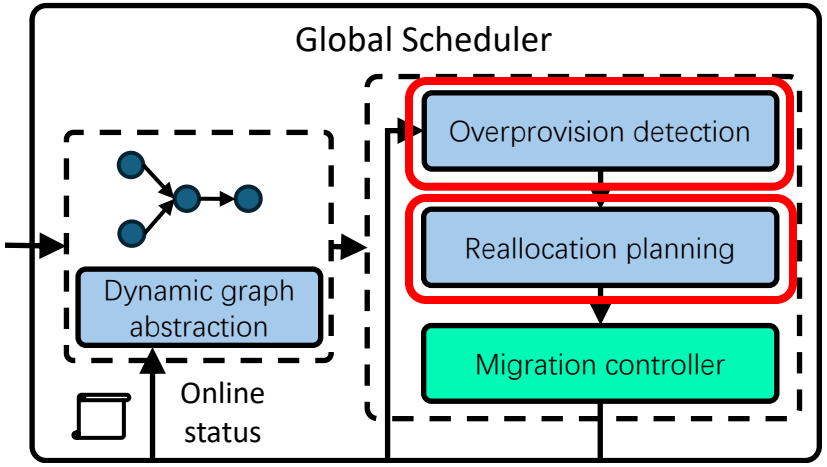


The Overprovision-Driven Scheduling Algorithm

- Goal: detect overprovision, reclaim and reallocate

Input: Component set $\mathcal{C}$, total resources R_{total}, thresholds $U_{\text{low}}, T, \theta$
Output: Updated resource allocation $\mathcal{R}$

```
1 Function MonitorLoop():
2     while true do
3         foreach C_i ∈ C do
4             (U_i, metrics_i) ← monitor(C_i);
5         C_opv ← DetectOverprovision(C_D, U_low, T, θ);
6         if C_opv ≠ None then
7             R_new ← PlanReallocation(C_opv, R, metrics);
8             migrate(R, R_new);
```



The Overprovision-Driven Scheduling Algorithm

❑ Goal: detect overprovision, reclaim and reallocate

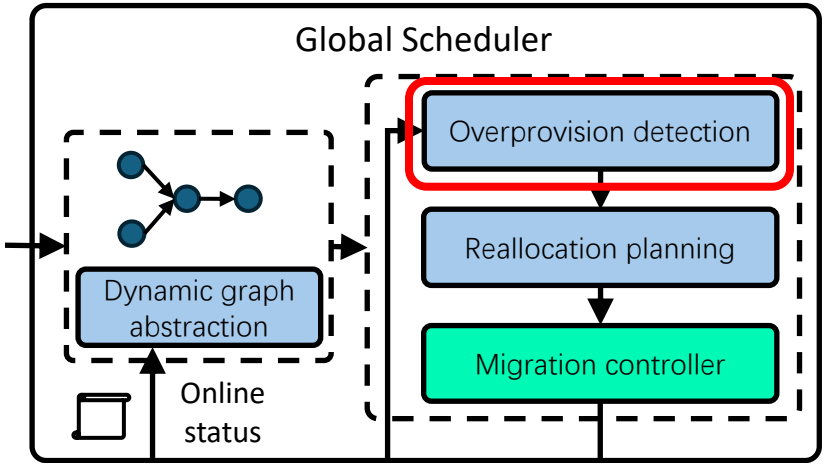
Input: Component set $\mathcal{C}$, total resources R_{total}, thresholds $U_{\text{low}}, T, \theta$
Output: Updated resource allocation $\mathcal{R}$

1 **Function** `MonitorLoop()`**:**
2 **while** *true* **do**
3 **foreach** $C_i \in \mathcal{C}$ **do**
4 $(U_i, \textbf{metrics}_i) \leftarrow \text{monitor}(C_i)$;
5 $C_{\text{opv}} \leftarrow$ `DetectOverprovision`$(\mathcal{C}_D, U_{low}, T, \theta)$;
6 **if** $C_{opv} \neq None$ **then**
7 $\mathcal{R}_{\text{new}} \leftarrow$ `PlanReallocation`$(C_{opv}, \mathcal{R}, \textbf{metrics})$;
8 migrate$(\mathcal{R}, \mathcal{R}_{\text{new}})$;
9 **Function** `DetectOverprovision`$(\mathcal{C}_D, U_{low}, T, \theta)$**:**
10 **foreach** $C_D \in \mathcal{C}_D$ **do**
11 low_count $\leftarrow 0$;
12 **foreach** *worker* $\in C_D.workers$ **do**
13 $U_{\text{avg}} \leftarrow \text{avg_utilization}(\text{worker}, T)$;
14 **if** $U_{avg} < U_{low}$ **then**
15 low_count $\leftarrow$ low_count $+ 1$;
16 **if** $low_count/|C_D.workers| > \theta$ **then**
17 **return** C_D;
18 **return** *None*;

Phase 1: Overprovision Detection

Rescheduling is triggered if:
the utilization of more than θ of C_D 's engines is below U_{low} for a continuous duration of T seconds

To avoid fluctuations.



The Overprovision-Driven Scheduling Algorithm

❑ Goal: detect overprovision, reclaim and reallocate

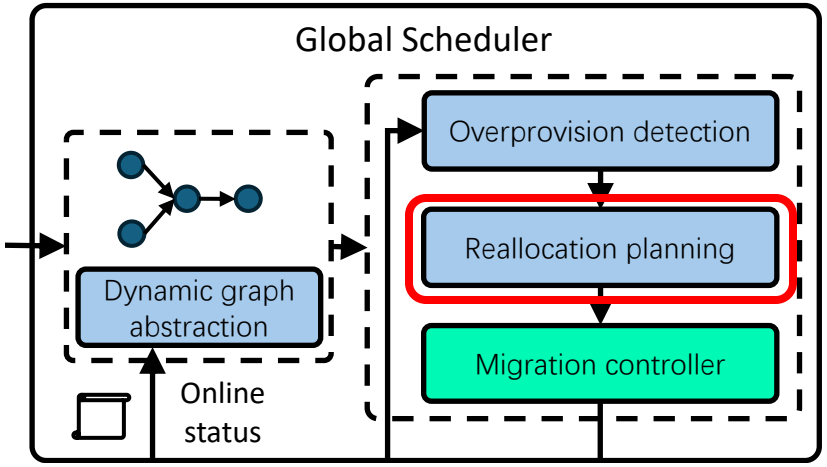
Input: Component set $\mathcal{C}$, total resources R_{total}, thresholds $U_{\text{low}}, T, \theta$
Output: Updated resource allocation $\mathcal{R}$

1 **Function** `MonitorLoop()`:
2 **while** *true* **do**
3 **foreach** $C_i \in \mathcal{C}$ **do**
4 $(U_i, \textbf{metrics}_i) \leftarrow \text{monitor}(C_i)$;
5 $\mathcal{C}_{\text{opv}} \leftarrow$ `DetectOverprovision`$(\mathcal{C}_D, U_{low}, T, \theta)$;
6 **if** $\mathcal{C}_{opv} \neq None$ **then**
7 $\mathcal{R}_{\text{new}} \leftarrow$ `PlanReallocation`$(\mathcal{C}_{opv}, \mathcal{R}, \textbf{metrics})$;
8 migrate$(\mathcal{R}, \mathcal{R}_{\text{new}})$;

19 **Function** `PlanReallocation`$(\mathcal{C}_{opv}, \mathcal{R}, \textbf{metrics})$:
20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(\mathcal{C}_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;
21 best $\leftarrow \varnothing$;
22 $\tau_{\text{best}} \leftarrow -\infty$;
23 **for** $k \leftarrow 1$ **to** K **do**
24 $R_{\text{opv}}^{(k)} \leftarrow \mathcal{R}[\mathcal{C}_{\text{opv}}] - \Delta R^{(k)}$;
25 $T_{\text{opv}}^{(k)} \leftarrow$ PredictThroughput$(\mathcal{C}_{\text{opv}}, R_{\text{opv}}^{(k)}, \textbf{metrics})$;
26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$

Phase 1: Overprovision Detection

Phase 2: Reallocation Planning



Phase 2: Reallocation Planning

19 **Function** `PlanReallocation` $(C_{opv}, \mathcal{R}, \textbf{metrics})$ **:**

20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(C_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;

21 best $\leftarrow \varnothing$;

22 $\tau_{\text{best}} \leftarrow -\infty$;

23 **for** $k \leftarrow 1$ **to** K **do**

24 $R_{\text{opv}}^{(k)} \leftarrow \mathcal{R}[C_{\text{opv}}] - \Delta R^{(k)}$;

25 $T_{\text{opv}}^{(k)} \leftarrow$ PredictThroughput$(C_{\text{opv}}, R_{\text{opv}}^{(k)}, \textbf{metrics})$;

26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$ SearchAllocation$(\Delta R^{(k)}, C_{\text{opv}}, R_{\text{opv}}^{(k)}, T_{\text{opv}}^{(k)}, C, \mathcal{R}, \textbf{metrics})$;

27 **if** $\tau^{(k)} > \tau_{best}$ **then**

28 $\tau_{\text{best}} \leftarrow \tau^{(k)}$;

29 best $\leftarrow (R_{\text{opv}}^{(k)}, \text{alloc}^{(k)})$;

30 $\mathcal{R}_{\text{new}} \leftarrow \mathcal{R}$;

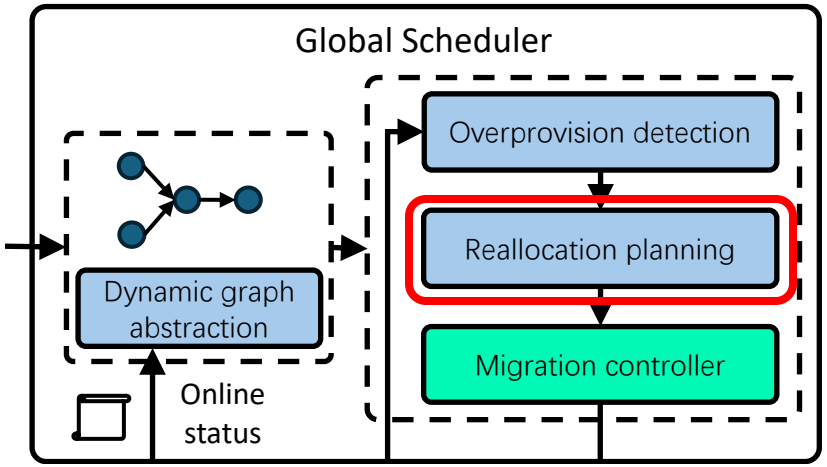
31 $\mathcal{R}_{\text{new}}[C_{\text{opv}}] \leftarrow$ best.R_{opv};

32 **foreach** $C_j \in best.alloc$ **do**

33 $\mathcal{R}_{\text{new}}[C_j] \leftarrow \mathcal{R}[C_j] +$ best.alloc$[C_j]$;

34 **return** $\mathcal{R}_{new}$;

1. Generate candidate reductions

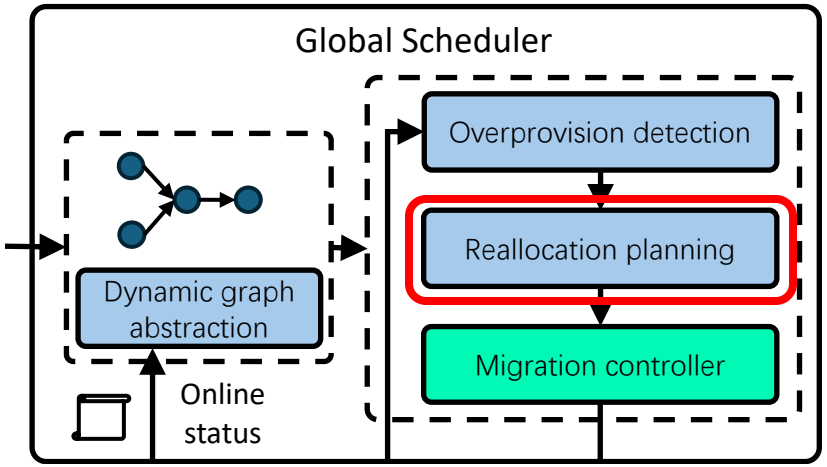


Phase 2: Reallocation Planning

19 **Function** `PlanReallocation` $(C_{opv}, \mathcal{R}, \textbf{metrics})$ **:**
20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(C_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;
21 best $\leftarrow \varnothing$;
22 $\tau_{\text{best}} \leftarrow -\infty$;
23 **for** $k \leftarrow 1$ **to** K **do**
24 $R_{\text{opv}}^{(k)} \leftarrow \mathcal{R}[C_{\text{opv}}] - \Delta R^{(k)}$;
25 $T_{\text{opv}}^{(k)} \leftarrow$ PredictThroughput$(C_{\text{opv}}, R_{\text{opv}}^{(k)}, \textbf{metrics})$;
26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$ SearchAllocation$(\Delta R^{(k)}, C_{\text{opv}}, R_{\text{opv}}^{(k)}, T_{\text{opv}}^{(k)}, C, \mathcal{R}, \textbf{metrics})$;
27 **if** $\tau^{(k)} > \tau_{best}$ **then**
28 $\tau_{\text{best}} \leftarrow \tau^{(k)}$;
29 best $\leftarrow (R_{\text{opv}}^{(k)}, \text{alloc}^{(k)})$;
30 $\mathcal{R}_{\text{new}} \leftarrow \mathcal{R}$;
31 $\mathcal{R}_{\text{new}}[C_{\text{opv}}] \leftarrow \text{best}.R_{\text{opv}}$;
32 **foreach** $C_j \in best.alloc$ **do**
33 $\mathcal{R}_{\text{new}}[C_j] \leftarrow \mathcal{R}[C_j] + \text{best.alloc}[C_j]$;
34 **return** $\mathcal{R}_{new}$;

1. Generate candidate reductions

2. Predict component performance



Phase 2: Reallocation Planning

19 **Function** `PlanReallocation` $(C_{opv}, \mathcal{R}, \textbf{metrics})$ **:**

20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(C_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;

21 best $\leftarrow \varnothing$;

22 $\tau_{\text{best}} \leftarrow -\infty$;

23 **for** $k \leftarrow 1$ **to** K **do**

24 $R_{\text{opv}}^{(k)} \leftarrow \mathcal{R}[C_{\text{opv}}] - \Delta R^{(k)}$;

25 $T_{\text{opv}}^{(k)} \leftarrow$ PredictThroughput$(C_{\text{opv}}, R_{\text{opv}}^{(k)}, \textbf{metrics})$;

26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$ SearchAllocation$(\Delta R^{(k)}, C_{\text{opv}}, R_{\text{opv}}^{(k)}, T_{\text{opv}}^{(k)}, C, \mathcal{R}, \textbf{metrics})$;

27 **if** $\tau^{(k)} > \tau_{best}$ **then**

28 $\tau_{\text{best}} \leftarrow \tau^{(k)}$;

29 best $\leftarrow (R_{\text{opv}}^{(k)}, \text{alloc}^{(k)})$;

30 $\mathcal{R}_{\text{new}} \leftarrow \mathcal{R}$;

31 $\mathcal{R}_{\text{new}}[C_{\text{opv}}] \leftarrow \text{best}.R_{\text{opv}}$;

32 **foreach** $C_j \in best.alloc$ **do**

33 $\mathcal{R}_{\text{new}}[C_j] \leftarrow \mathcal{R}[C_j] + \text{best.alloc}[C_j]$;

34 **return** $\mathcal{R}_{new}$;

1. Generate candidate reductions

2. Predict component performance

3. Evaluate global reallocation benefit

Phase 2: Reallocation Planning

19 **Function** `PlanReallocation` $(C_{opv}, \mathcal{R}, \textbf{metrics})$ **:**
20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(C_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;
21 best $\leftarrow \varnothing$;
22 $\tau_{\text{best}} \leftarrow -\infty$;
23 **for** $k \leftarrow 1$ **to** K **do**
24 $R_{\text{opv}}^{(k)} \leftarrow \mathcal{R}[C_{\text{opv}}] - \Delta R^{(k)}$;
25 $T_{\text{opv}}^{(k)} \leftarrow$ PredictThroughput$(C_{\text{opv}}, R_{\text{opv}}^{(k)}, \textbf{metrics})$;
26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$ SearchAllocation$(\Delta R^{(k)}, C_{\text{opv}}, R_{\text{opv}}^{(k)}, T_{\text{opv}}^{(k)}, C, \mathcal{R}, \textbf{metrics})$;
27 **if** $\tau^{(k)} > \tau_{best}$ **then**
28 $\tau_{\text{best}} \leftarrow \tau^{(k)}$;
29 best $\leftarrow (R_{\text{opv}}^{(k)}, \text{alloc}^{(k)})$;
30 $\mathcal{R}_{\text{new}} \leftarrow \mathcal{R}$;
31 $\mathcal{R}_{\text{new}}[C_{\text{opv}}] \leftarrow \text{best}.R_{\text{opv}}$;
32 **foreach** $C_j \in best.alloc$ **do**
33 $\mathcal{R}_{\text{new}}[C_j] \leftarrow \mathcal{R}[C_j] + \text{best.alloc}[C_j]$;
34 **return** $\mathcal{R}_{new}$;

1. Generate candidate reductions

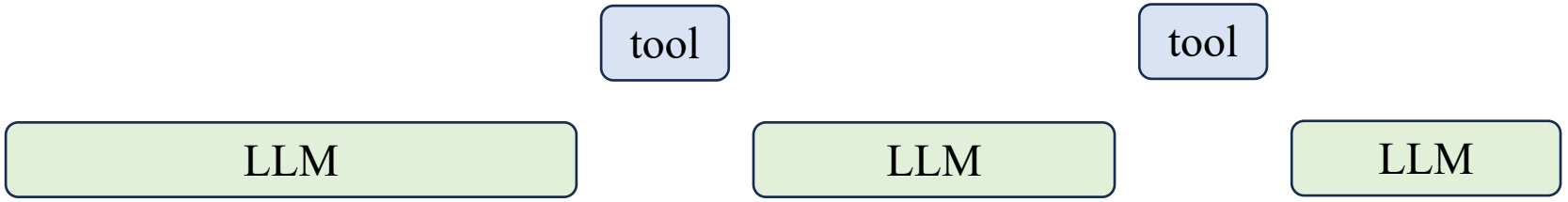
2. Predict component performance

3. Evaluate global reallocation benefit

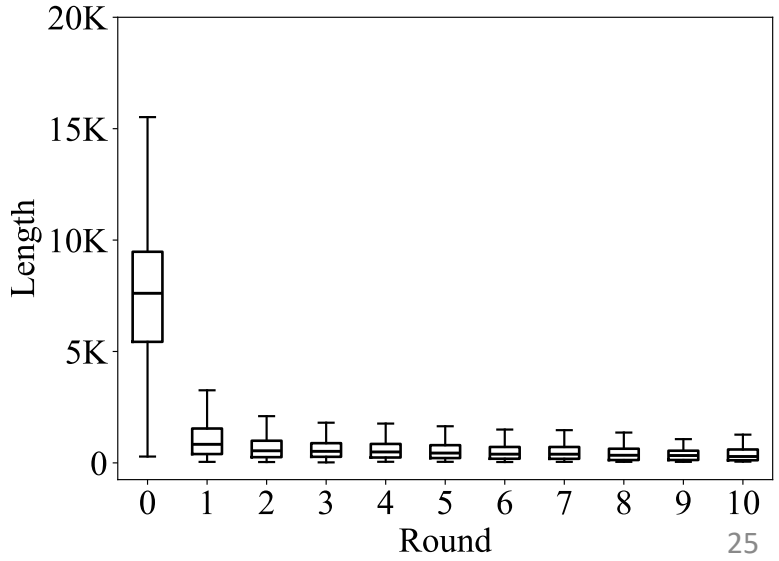
4. Select optimal plan

Data-Level Scheduling: Priority-Aware Requests

- ReAct-style agents interleave LLM generation with tool calls.

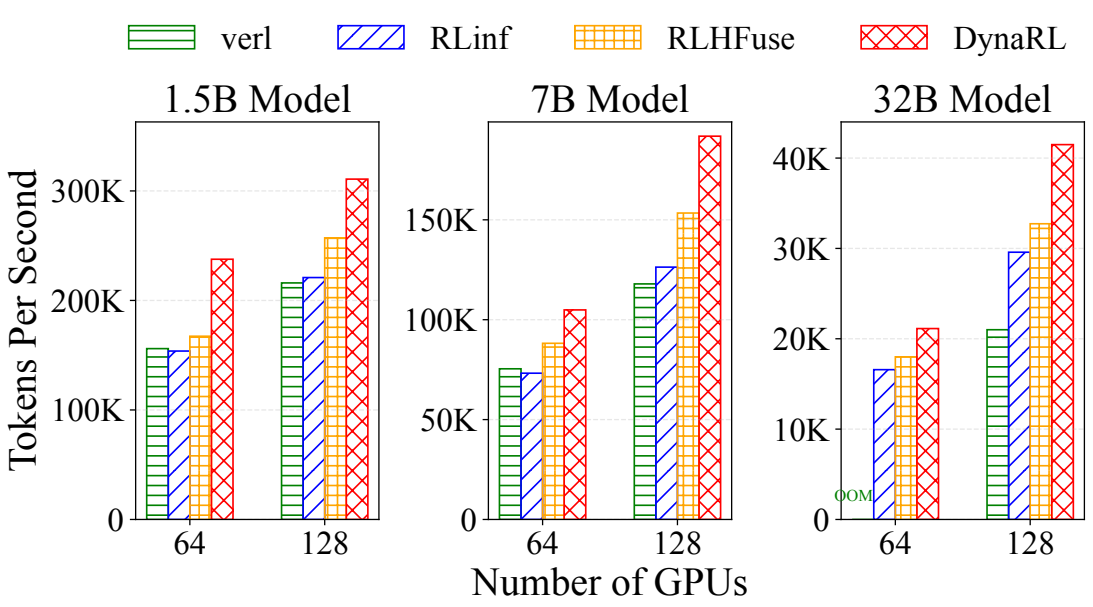


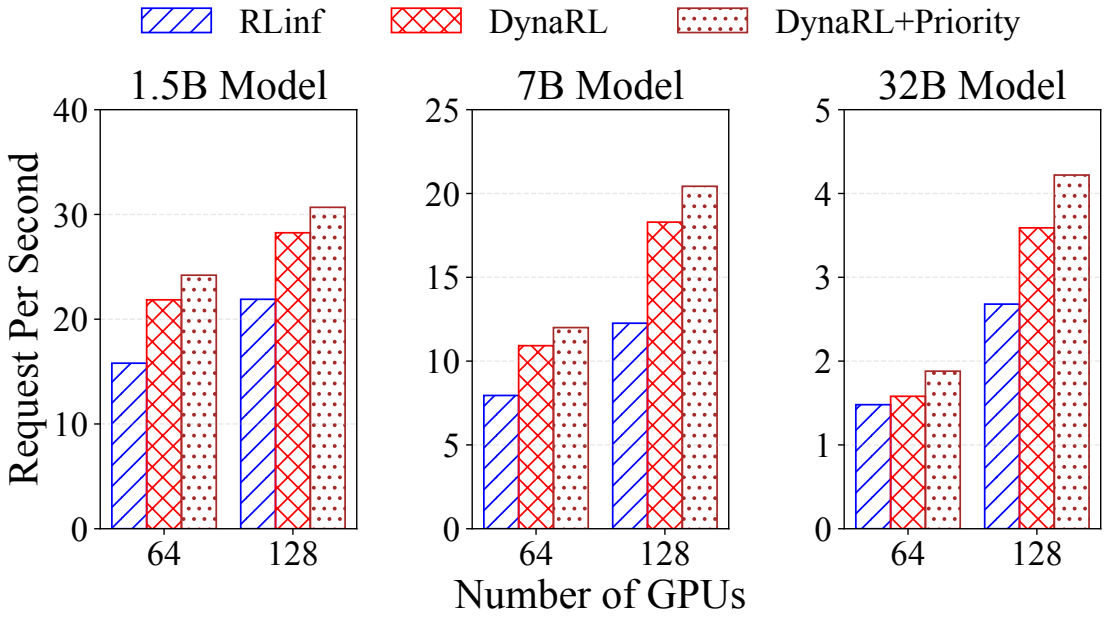
- Response of earlier turns tends to be substantially longer than subsequent ones
- We prioritize requests with larger the number of completed tool invocations so far
  - Increase the likelihood of KV-cache reuse
  - Accelerate the completion of late-stage episodes



Evaluation

- Testbed: 16 nodes × 8× NVIDIA H100-80GB = 128 GPUs, 400 Gbps RoCEv2 per NIC
- Workloads:
  - Math-reasoning RL — AReaL-boba dataset
  - Multi-turn agentic RL — rstar2-agent dataset
- Models: 1.5B, 7B, 32B
- Baselines: verl v0.5, RLinf, RLHFuse





Conclusions

- ❑ DynaRL: dynamic scheduling as a first-class principle for RL
- ❑ Characterize dynamicity in modern RL workloads
- ❑ Dynamic HyperGraph + unified migration + context-aware routing
- ❑ Multi-level scheduling (component / worker / request)
- ❑ https://github.com/RLinf/RLinf. Welcome your issues, PRs and star!