Canopy: Tree-Aware Rollout Scheduling for Agent Reinforcement Learning

Feiyuan Zhang[1*] **Pengbo Li**[1] **Ziniu Li**[2†] **Yuhao Jiang**[2]
Di Chai[3] **Han Tian**[1] **Junhao Wang**[4] **Taiqiang Wu**[5]
Guanhua Huang[2] **Chaoliang Zeng**[1] **Yihao Liu**[2] **Kai Chen**[1†]

[1]The Hong Kong University of Science and Technology [2]Tencent
[3]Shanghai University of Finance and Economics [4]The Chinese University of Hong Kong
[5]The University of Hong Kong

Abstract

Reinforcement learning (RL) for LLM agents spends most of its time generating rollouts, and the requests of a rollout step share most of their input: GRPO samples share a prompt, tree branches share the history before the branch, and each turn re-reads earlier turns. RL frameworks send each turn to the serving stack as an independent request, so the stack computes a shared prefix on several instances and recomputes histories evicted while trajectories wait for tools; verl computes 2.5–7.9× the minimum prefill on our workloads. Yet the RL controller already knows which trajectories share a prefix, when they end, and how much memory the previous step needed. We present Canopy, a rollout scheduler that passes these facts to the serving stack and makes the group of trajectories sharing a prefix the unit of placement and memory accounting. Canopy keeps each group on one instance and starts or resumes a trajectory only when the instance can hold it, counting the group's shared prefix once and charging each trajectory at least the previous step's estimate. On five synthetic tree-rollout, search, and long-context workloads modeled on Qwen3-8B agent trajectories, Canopy finishes a rollout step 1.6–3.6× faster than verl and 1.2–2.8× faster than program-level scheduling, and it is 1.6–2.3× faster than verl on long-horizon trajectories that branch late.

1 Introduction

Reinforcement learning (RL) of LLM agents spends most of its time generating rollouts [Sheng et al., 2026, Fu et al., 2025, Qin et al., 2026], and the requests of one rollout step share most of their input. Each turn of an agent [Jin et al., 2025, Chen et al., 2025, Dong et al., 2026b] is a request whose input is the whole history so far. GRPO samples several trajectories per prompt [Shao et al., 2024], and tree rollouts branch trajectories from a shared history, at turn boundaries [Ji et al., 2026, Zong et al., 2026] or inside a turn [Dong et al., 2026b,a]; we call the trajectories that share a prefix a *group*. In our workloads, an earlier request of the same step already processed 75–92% of the input tokens (Table 1).

The serving stack sees these requests as unrelated. verl [Sheng et al., 2025] routes each trajectory to the instance (a data-parallel copy of the model) with the fewest in-flight requests, and vLLM [Kwon et al., 2023] evicts unreferenced KV blocks in least-recently-used order. Figure 1 shows the two ways this causes excess prefill. A group's trajectories land on different instances, and each instance computes the shared prefix again. When an instance's live trajectories need more KV memory than it has, the engine evicts the histories of those waiting for tools and recomputes them at

*Contact: fzhangax@connect.ust.hk. †Corresponding authors: ziniuli@tencent.com, kaichen@cse.ust.hk.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

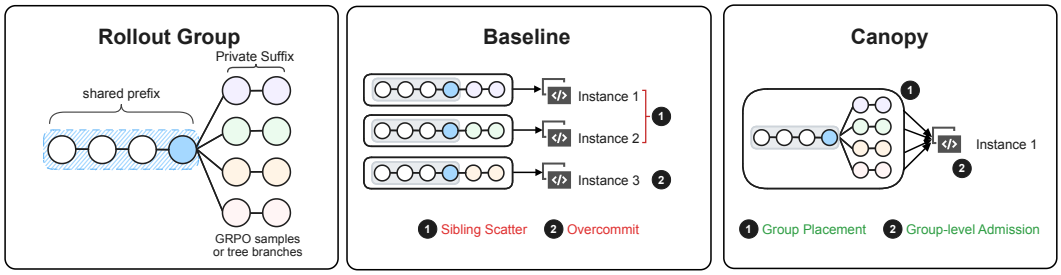


Figure 1: A rollout group (GRPO samples or tree branches) shares a long prefix (left), but request-level serving, the baseline in current RL frameworks, schedules each request on its own (middle): siblings land on different instances, each of which computes the shared prefix (1), and instances admit more live trajectories than their KV memory holds, so the histories of tool-waiting trajectories are evicted and recomputed (2). Canopy schedules the group (right): group placement (1) and group-level admission (2).

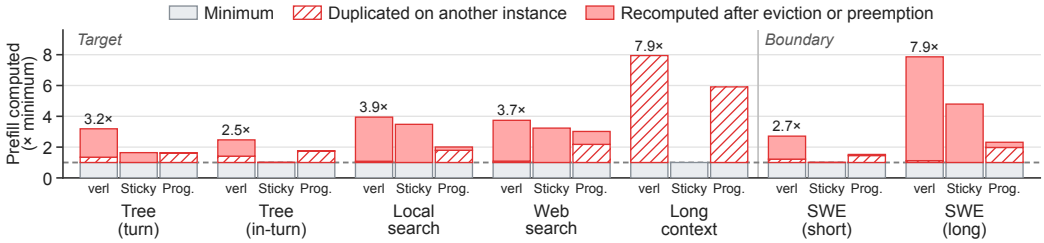


Figure 2: Prefill computed in one rollout step as a multiple of the minimum (eight instances, median of three runs). Bars: verl, TreeSticky, and program-level scheduling (Prog.); hatched: duplicated on another instance; red: recomputed after eviction or preemption. Canopy computes 1.00–1.05× the minimum (dashed line).

their next turn. Relative to the minimum prefill, in which every shared prefix and history is computed once per step, verl computes 2.5–7.9×. Duplication alone accounts for 6.9× in long-context GRPO, and recomputation for 2.6–6.7× in search and long SWE tasks (Figure 2).

Prior systems fix parts of this loss, but each decides per request, per tool call, or per trajectory. Prefix-aware routers [The SGLang Team, 2024, Srivatsa et al., 2025, llm-d Project, 2025] do not limit how much work an instance takes, and agent-aware KV managers [Abhyankar et al., 2024, Li et al., 2025, Bian et al., 2025, Pan et al., 2025] do not decide where requests run. Seer [Qin et al., 2026] uses GRPO groups to predict output lengths, but deliberately splits each group across instances to balance load. Program-level scheduling, as in ThunderAgent [Kang et al., 2026], pauses tool-waiting trajectories when an instance is over-committed, which removes most recomputation, but it places and charges trajectories, not groups, so it moves the loss to other instances (Figure 2). None of these systems places requests or accounts for memory by group.

Making the group, rather than the request, the unit of scheduling needs facts that the RL controller already has. It sets group sizes and branch points and runs the agent loops, so it knows each group, its shared prefix, and when each trajectory and group ends. Because RL repeats similar steps, the previous step also lets it estimate how much memory a trajectory will need. The serving stack cannot learn these facts by itself: a prefix match shows which tokens are shared but not whether they will be read again, and a trajectory's future memory depends on turns it has not taken yet.

Canopy passes these facts to the serving stack with each request (Figure 4) and makes the group the unit of two decisions. *Group placement* keeps each group on one instance, so its shared prefix is computed once per step. *Group-level admission* starts or resumes a trajectory only when its instance can hold it: like program-level scheduling, it pauses tool-waiting trajectories, but it counts a group's shared prefix once for the group's lifetime and charges each trajectory at least the previous step's peak private KV per trajectory. Two retention mechanisms complete the design but matter little in our workloads: *lifetime-aware eviction* evicts finished work first and a live group's shared prefix last, and *prefix-preserving preemption* frees only a preempted request's private suffix. Appendix I lists the changes since the reviewed version.

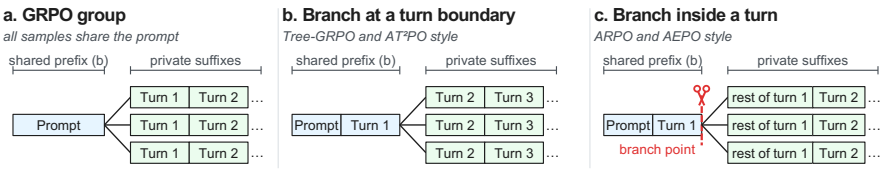


Figure 3: How the requests of a rollout step share input (schematic). Blue: a group's shared prefix, of length b; green: private suffixes, which later turns re-read. (a) GRPO samples share the prompt. (b) Trees that branch at a turn boundary share whole turns. (c) Trees that branch inside a turn also share the part of the turn generated before the branch point; the children continue from it without a tool call. In our tree workloads, trees branch after the second turn or inside the third (Appendix B).

We compare serving policies on identical work: every policy replays the same synthetic rollout steps, modeled on small samples of Qwen3-8B agent trajectories, on the same vLLM instances, and Canopy's previous-step estimate comes from a second sample of each workload (§4.1). On five workloads, Canopy is 1.6–3.6× faster than verl and 1.2–2.8× faster than program-level scheduling, and at most 1% slower than the best of three static caps chosen after the fact; placement gives most of the gain on trees and long-context GRPO, and admission on search. On synthetic trajectories of 64k–122k tokens that branch after 4–19 turns, Canopy is 1.6–2.3× faster than verl, and the gain shifts from placement to admission as the shared prefix shrinks (§4.4). Holding shared prefixes and prefix-preserving preemption, off in the main runs, change rollout time by at most 3.5%. Canopy is 13% slower on a math workload, and 300-step training runs without admission show no detectable change in validation accuracy.

In summary, this paper makes three contributions:

- We identify why serving wastes prefill in agent RL rollouts, whose GRPO groups and rollout trees share long prefixes: request-level serving computes a group's shared prefix on several instances and recomputes histories evicted while trajectories wait for tools. On our workloads, verl computes 2.5–7.9× the minimum prefill, although the RL controller already knows the facts needed to avoid both: the group structure and the previous step's memory use.
- We present Canopy, a rollout scheduler that passes these facts to the serving stack and makes the group, rather than the request or the trajectory, the unit of placement and memory accounting. Group placement keeps each group on one instance so that its shared prefix is computed once per step, and group-level admission starts or resumes a trajectory only when the instance can hold it, counting the group's shared prefix once rather than once per trajectory and charging each trajectory at least an estimate from the previous step.
- We evaluate Canopy on synthetic rollout steps modeled on Qwen3-8B agent trajectories and replayed identically for every policy, including GRPO groups and trees that branch at turn boundaries or inside a turn. On five tree-rollout, search, and long-context workloads, Canopy computes within 5% of the minimum prefill and is 1.6–3.6× faster than verl, 1.2–2.8× faster than program-level scheduling, and up to 2.1× faster than prefix-aware routing, and without per-workload tuning it matches the best of three static concurrency caps chosen after the fact.

Code is available at `https://github.com/FarrenZhang/canopy-rl`.

2 Background and Motivation

Rollouts in agent RL. Each turn of an agent trajectory is one generation request whose input is the whole history so far. We call the trajectories that share a prefix a *group*: the samples of one GRPO prompt (8 per prompt in our workloads) or the branches below one node of a rollout tree (16 leaves per tree). Their common tokens form the group's *shared prefix*, of length b; the rest of each trajectory is its *private suffix* (Figure 3). A trajectory is *live* while it will send more requests, including while it waits for a tool; a group is *done* when all its trajectories have ended. In synchronous RL the last trajectory sets the step's rollout time.

How rollouts are served. RL frameworks such as verl [Sheng et al., 2025], AReaL [Fu et al., 2025], and slime [The slime Team, 2025] run one agent loop per trajectory and send its turns to a pool of vLLM [Kwon et al., 2023] or SGLang [Zheng et al., 2024] instances. verl sends a trajectory's first

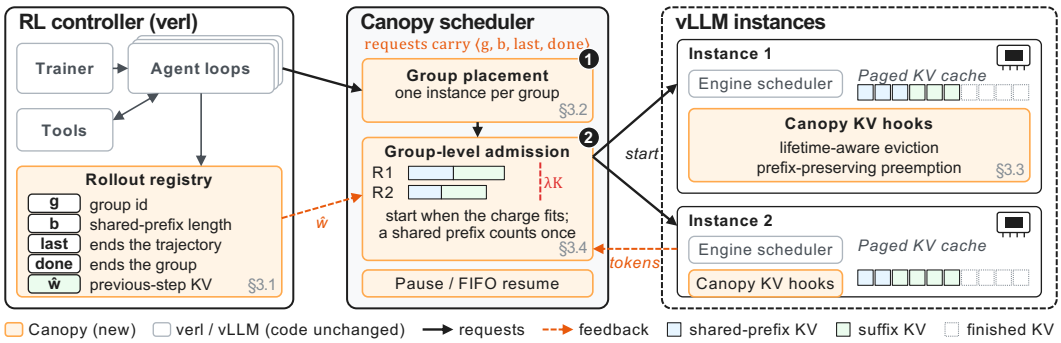


Figure 4: Canopy's design. Each request carries the descriptor $\langle g, b, last, done \rangle$ from its agent loop to the scheduler (§3.1). The scheduler keeps each group on one instance (1, §3.2) and starts or resumes a trajectory only when the instance's charge, which counts a shared prefix once, fits within its capacity λK (2, §3.4); each engine evicts by lifetime and keeps the shared prefix when it preempts a request (§3.3). Orange: added by Canopy; white: unchanged verl and vLLM.

turn to the instance with the fewest in-flight requests and keeps its later turns there, so the samples of one prompt can land on different instances. The engine stores KV cache in fixed-size blocks; with prefix caching, a block keeps its content after its request finishes and serves later requests with the same prefix. Blocks that a running request references are never evicted; the others are evicted in least-recently-used (LRU) order. When a running request cannot get new blocks, vLLM preempts another request and makes its blocks evictable; when the preempted request runs again, the engine recomputes those of its blocks that were evicted in the meantime [vLLM Team, 2026].

Where the extra prefill goes. We measure excess prefill as prefill computed beyond the *minimum*, in which every shared prefix and history is computed once per step, and split the excess into prefill *duplicated on another instance* and prefill *recomputed after eviction or preemption*. Recomputation after preemption is at most 2% of the minimum in every run. Figure 2 gives the results (method in Appendix D). Request-level serving incurs both kinds, and the workload decides which dominates: long-context GRPO duplicates each group's document on up to eight instances, while search and long SWE tasks recompute histories because each instance runs 48–256 live trajectories whose histories grow to 12–39k tokens. Keeping groups together (TreeSticky) removes duplication but still recomputes 2.2–3.8× the minimum on these three workloads. Program-level scheduling, as in ThunderAgent [Kang et al., 2026], cuts recomputation to at most 0.8× but duplicates histories of resumed trajectories on other instances (0.23–1.09×) and prefixes of split groups (4.9× in long-context GRPO). Most reusable input is not shared between siblings: in local search, web search, and SWE tasks, 65–85% of it is the trajectory's own history (Table 1). The excess is mainly whole histories of live trajectories, evicted because an instance runs more trajectories than its KV memory holds. Avoiding this requires starting trajectories only when the memory they will need fits, and avoiding duplication requires keeping groups together.

3 Canopy

Canopy changes how the serving stack schedules the requests of a rollout step, not what they compute. The RL controller attaches a descriptor to every request (§3.1), and the serving stack uses it for three decisions (Figure 4): which instance runs a request (§3.2), which KV blocks stay cached (§3.3), and when a trajectory may start or resume (§3.4). Placement and admission produce the speedups; holding a live group's shared prefix and prefix-preserving preemption change rollout time by at most 3.5% and are off in the main runs. No decision changes the model, the sampling, the rewards, or the update.

3.1 The rollout descriptor

For a request u of trajectory τ in group g, the controller sends

$$d_u = \langle\, g,\ b_u,\ last_u,\ done_u \,\rangle. \tag{1}$$

g identifies the group. b_u is the number of leading input tokens that u shares with other live trajectories of g: the prompt for a GRPO sample, or the history up to the branch point for a tree branch, including a branch point inside a turn. $last_u$ marks the last request of τ, and $done_u$ marks a request after which no trajectory of g remains live. The controller knows g and b_u when it sends u, because it chooses how many trajectories to sample and where to branch. It sets $last_u$ and $done_u$ only when the end is known in advance, as with a turn limit; otherwise the flags stay unset and u's blocks are evicted in LRU order. The descriptor is advisory: a wrong value can cost time but cannot change what a request computes.

3.2 Group placement

Placement removes duplicated prefill. The first trajectory of a group goes to the least-loaded instance, and every later request of the group goes to the same instance, so the group computes and caches its shared prefix once. Load is the number of in-flight requests, as in verl, or, with admission, the memory charged to the instance (§3.4). Balancing whole groups works because a group is small next to an instance's load: 8–16 trajectories against 48–256 per instance in our workloads. Because a group never leaves its instance, placement cannot cascade or oscillate; the cost is coarser balance when group sizes differ (§4.7).

3.3 Lifetime-aware eviction and prefix-preserving preemption

Canopy evicts freed blocks in three tiers: first, blocks that no future request will read (the private blocks of an ended trajectory, marked by $last_u$, and all blocks of a done group, marked by $done_u$); then blocks of live trajectories, in LRU order; last, the first b_u tokens of a live group, on which the engine holds an extra reference until the group is done. Before the engine preempts a request u of a live group, it adds the same kind of reference to the cached blocks of u's first b_u tokens, so preemption frees only the private suffix (Figure 10). The references are soft: when an allocation fails, the engine drops them, least recently used group first, and a group that sends no request for 60 s loses them, so retention never blocks a request and retained blocks cannot accumulate.

The gain is small: a prefix that a running sibling uses is never evicted, and one that siblings read at every turn stays near the recent end of the LRU order. In our workloads, holding shared prefixes and prefix-preserving preemption change rollout time by at most 3.5% (§4.3), so the main runs keep only the first tier, finished-first eviction.

3.4 Group-level admission

When the live trajectories on an instance need more memory than it has, the engine evicts their histories between turns and recomputes them (§2). Program-level scheduling [Kang et al., 2026] prevents this by charging each program (trajectory) its full context, pausing tool-waiting programs when an instance's charge exceeds its KV capacity, and resuming them when memory frees. Canopy keeps pause and resume and changes the charge in two ways: a group's shared prefix is counted once, for the group's whole lifetime, and each trajectory is charged at least what trajectories needed in the previous step.

Charge. For instance r, let $\mathcal{G}_r$ be the groups placed on r that are not done, and $\mathcal{T}_r$ the trajectories started on r that have not ended and are not paused. A trajectory τ of group g has private context p_τ: the tokens of its current input and output beyond b_g, the prefix that all of g's trajectories share (for a tree, its prompt). The charge of r counts each group's shared prefix once and keeps it while the group is not done, even when all of its trajectories wait or are paused, because they will read it again:

$$C_r = \sum_{g \in \mathcal{G}_r} b_g + \sum_{\tau \in \mathcal{T}_r} \omega_\tau \, p_\tau, \tag{2}$$

where $\omega_\tau = 1$ while τ runs or has waited for a tool for at most 1 s, and then halves every 5 s (ThunderAgent's code offers a similar discount, 2^{-t}, off by default).

Estimate. Whether a trajectory may start depends on how large it will grow, which depends on turns it has not taken yet. RL repeats the same kind of step, so the controller uses $\hat{w}$, the previous

Algorithm 1 One scheduler pass, run when a request arrives or finishes and at least every 0.5 s. Q: queued and paused trajectories in arrival order; $h(g)$: group g's instance; n_g: branches of g that start without a tool call.

for each instance r and each tool-waiting $\tau \in \mathcal{T}_r$, largest p_τ first, while $C_r > \lambda K_r$ **do**
    pause τ and append it to Q
for each trajectory τ of group g in Q, in order **do**
    for r in $[\,h(g)\,]$ if $h(g)$ is set, else all instances by increasing $\hat{C}_r/K_r$ **do**
        if $\hat{C}_r + \max(p_\tau, \hat{w}) + [g \notin \mathcal{G}_r]\,(b_g + n_g\hat{w}) \le \lambda K_r$ **or** $\mathcal{T}_r = \emptyset$ **then**
            start or resume τ on r; set $h(g) \leftarrow r$ if unset; remove τ from Q; **break**

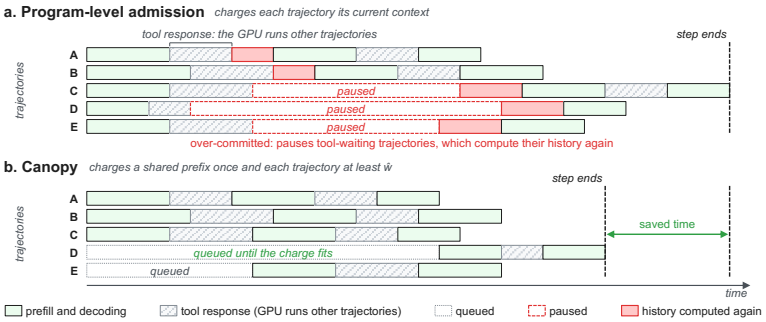


Figure 5: Admission on one instance in one rollout step (schematic; rows A–E are trajectories; green: prefill and decoding; gray: tool responses). (a) Program-level admission charges each trajectory its current context, so it over-commits as contexts grow and pauses tool-waiting trajectories, whose histories are recomputed when they resume. (b) Canopy charges a shared prefix once and each trajectory at least $\hat{w}$, so it starts trajectories only when they fit.

step's peak total private KV divided by its number of trajectories (our replay experiments compute it from a second sample of the workload; Appendix A), and charges each trajectory at least $\hat{w}$:

$$\hat{C}_r = \sum_{g \in \mathcal{G}_r} b_g + \sum_{\tau \in \mathcal{T}_r} \max(p_\tau, \hat{w}). \tag{3}$$

Rules. Algorithm 1 gives one pass of the scheduler (K_r: KV capacity; $\lambda = 1$). It first pauses tool-waiting trajectories on over-committed instances and then starts or resumes queued trajectories in FIFO order wherever they fit; a trajectory that does not fit stays queued without blocking later ones, and the instance that starts a group's first trajectory becomes the group's instance. Branches inside a turn start as soon as their parent's turn ends, before admission can act, so their $\hat{w}$ is reserved when the group starts. Because an instance with no started trajectory always starts the first trajectory queued for it, every trajectory starts. Figure 5 shows the effect on one instance.

Approximations. History that only some branches share is charged to each of them, which overestimates memory. $\hat{w}$ comes from the previous step, so it is too small or too large when the policy drifts; §4.6 measures errors of 2–4×. The reservation for branches inside a turn uses the known tree rather than a prediction.

4 Evaluation

4.1 Setup

Method: replaying synthetic rollout steps. Model outputs change from run to run, so to compare policies on identical work we generate each rollout step from a seed as a request graph with fixed inputs, output lengths, tool latencies, and dependencies. We set the ranges by hand from small samples of Qwen3-8B trajectories on LongBench v2 [Bai et al., 2025], FRAMES [Krishna et al., 2025] with a web search API, SWE-bench Verified [Jimenez et al., 2024], and a local retriever (Appendix B). Siblings share their prefixes token by token, and a turn's input is the previous input plus the generated tokens and the tool output, so prefix-cache hits are those of a live rollout; outputs

Table 1: Workloads (totals per step on eight instances). Reusable: share of input already processed by an earlier request of the step; own history: the part that is the trajectory's own earlier turns.

Workload	Groups × size per instance	Turns	Tool latency per call	Input (M)	Output (M)	Min. prefill (M)	Reusable input	Own history
Tree search (turn)	8 trees × 16	6	0.2–1.5 s	21.0	1.2	4.60	78%	49%
Tree search (in-turn)	8 trees × 16	6	0.2–1.5 s	22.0	1.1	3.79	83%	27%
Local search (GRPO)	32 × 8	3–8	0.2–1.5 s	42.3	2.8	9.89	77%	85%
Web search (GRPO)	16 × 8	3–8	median 8 s	46.1	2.0	11.51	75%	82%
Long context (GRPO)	24 × 8	1	none	24.6	0.5	3.10	87%	0%
SWE agent (short)	8 × 8	6–14	median 0.3 s	43.1	0.4	5.37	88%	65%
SWE agent (long)	6 × 8	10–24	median 0.3 s	89.8	0.5	7.32	92%	79%
Math (control)	16 × 8	1	none	0.3	4.5	0.05	82%	0%

have fixed lengths. In each experiment all policies run on the same vLLM instances from an empty cache, in rotating order after a warm-up. Rollout time runs from the first request to the last response.

Workloads. Table 1 lists the workloads. Trees branch at turn boundaries (Tree-GRPO and AT^{2}PO style) or inside a turn (ARPO and AEPO style; Figure 3), with 16 leaves and six turns. Local and web search are multi-turn GRPO groups; long-context GRPO is a single-turn group over a 12–20k-token document. Two SWE-style workloads are boundary cases, and math reasoning, whose input is small next to its output, is a control.

Systems. All policies are our implementations in one rollout driver on the same engine (Appendix C). *verl* routes each trajectory to the instance with the fewest in-flight requests. *Prefix-aware routing* routes by the longest cached prefix unless load is imbalanced, as the SGLang router [The SGLang Team, 2024] does. *TreeSticky* only keeps each group on one instance. *TTL* holds a trajectory's KV for 2 s after each turn, a fixed version of Continuum's time-to-live [Li et al., 2025]. *Program-level* re-implements ThunderAgent's policy [Kang et al., 2026]: it charges each trajectory its full context, pauses tool-waiting trajectories when an instance is over-committed, and places trajectories to balance memory. Appendix C also runs a re-implementation of ThunderAgent's released scheduler; Canopy is 1.21–2.78× faster than the faster of the two.

Static cap admits whole groups while at most K root trajectories run on an instance (best of three K, chosen after the fact). *Canopy w/o admission* runs group placement, the three eviction tiers, and prefix-preserving preemption. *Canopy* adds group-level admission; in the main runs shared-prefix references and prefix-preserving preemption are off (turning them on changes time by at most 1.1%, §4.3). Admission in Canopy and the program-level baseline knows each request's output length (§4.6 uses estimates); Canopy's previous-step estimate comes from a second sample of the workload.

Hardware and metrics. We use H20 GPUs with vLLM 0.20.1 and Qwen3-8B, one instance per GPU, prefix caching on, and default batching limits (Appendix B). Main results use eight instances and three repetitions; we report the median and the range.

4.2 Main results

Canopy is the fastest policy, or within 2% of it, on every target workload. On the five target workloads, Canopy is 1.64–3.61× faster than verl (geometric mean 2.14×), 1.21–2.84× faster than the program-level scheduler and 1.02–2.07× faster than prefix-aware routing (Figure 6). It is between 1% slower and 7% faster than the best static cap, whose value differs by workload, and TTL computes as much prefill as verl.

The source of the gain follows Figure 2. In long-context GRPO, placement alone gives 3.5× and the program-level scheduler, which splits groups, 1.27×. In tree search (turn), placement gives 1.42× and admission adds 1.31×; in tree search (in-turn), placement gives 1.67× and admission changes little (−2%). In local and web search, placement gives only 1.09–1.10×, because the instances still recompute 2.2–2.5× the minimum after eviction; admission removes this, for 2.19× and 1.86×

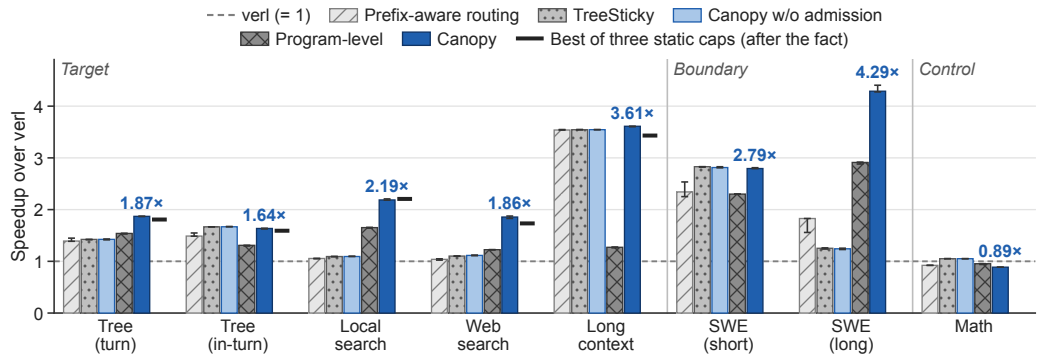


Figure 6: Speedup over verl on eight instances (median and range of three runs). Black ticks: best of three static caps. TTL is within 3% of verl and not drawn; times for the target workloads are in Table 5.

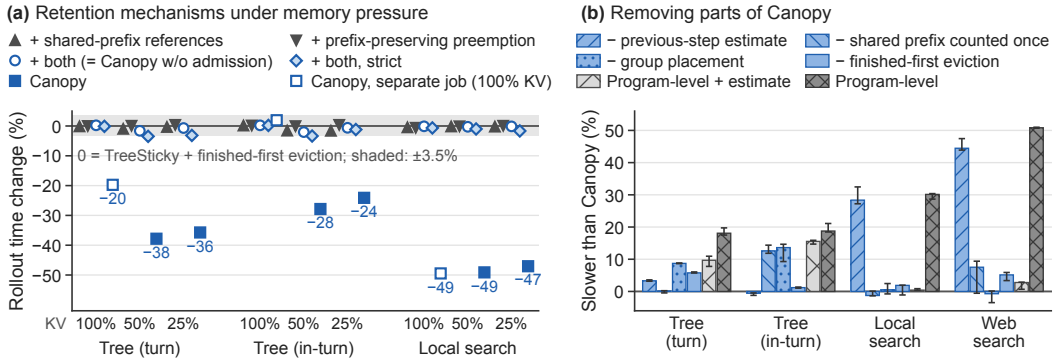


Figure 7: Where the gains come from (four instances). (a) Shared-prefix references and prefix-preserving preemption added to TreeSticky with finished-first eviction, at 100%, 50%, and 25% of the KV memory: change in median rollout time ("strict": references also hold memory against new requests; squares: Canopy). (b) Canopy with one part removed, and the program-level scheduler with and without Canopy's estimate: slowdown relative to Canopy (median and range of three runs).

overall. The program-level scheduler gives 1.65× and 1.23× there: it computes histories again on other instances and pauses 2,367 and 2,156 times per run, against 29 and 22 for Canopy.

Stability. With admission, the engines preempt at most once per run (verl: 14–82 times), and every trajectory finishes before verl's step ends, although the longest wait is 53–230 s (Table 6).

4.3 Where the gains come from

Placement and admission produce the gains; holding shared prefixes and prefix-preserving preemption change rollout time by at most 3.5%.

Placement, eviction order, and preemption. Figure 7a adds shared-prefix references, prefix-preserving preemption, or both to TreeSticky with finished-first eviction. With full KV memory all variants are within 1% of TreeSticky. With 50% and 25% of the memory, references and preemption save at most 2.0% (3.5% with strict references), while at 25% Canopy is 1.31–1.89× faster than Canopy w/o admission: there TreeSticky recomputes 1.8–2.9× the minimum, the two mechanisms lower this by at most 0.30×, and admission brings it to 0.01–0.24× (Appendix D). With admission, adding both changes time by at most 1.1% on eight instances (Table 7) and by -3% to $+2.5\%$ on four (Figure 12).

Admission. Figure 7b removes one part of Canopy at a time. In search, the previous-step estimate matters most: without it, rollout takes 28% and 44% longer, and the program-level scheduler with the same estimate comes within 0.4% and 2.8% of Canopy. On trees, group information matters: without

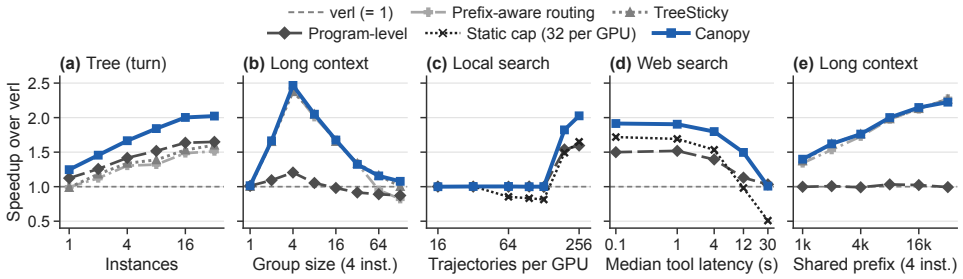


Figure 8: Speedup over verl when one factor changes (one run per point). Panels vary instances, group size, trajectories per GPU, tool latency, and shared-prefix length; settings in Appendix E.

placement they take 9% and 14% longer, without counting the shared prefix once in-turn trees take 13% longer, and the program-level scheduler with the estimate stays 10–16% slower. Finished-first eviction saves 1–6% when admission is on. We did not ablate the group-lifetime charge or the in-turn reservation.

4.4 Long trajectories that branch late

On longer trajectories that branch after many turns, Canopy is 1.6–2.3× faster than verl (Table 11); placement gives most of the gain when the shared prefix is long, and admission when it is short. Groups run as one trajectory for 4–19 turns and then fork, with 64k–122k-token trajectories, 4–16 branches, and shared prefixes of 32–95% of a trajectory; each step needs 1.3–1.9× the KV memory of four instances, and verl computes 3.6–5.9× the minimum prefill (Table 12). Here Canopy charges each request its own b (§3.1) rather than its group's first b; with the main-run charge it is 1.5–2.1× faster than verl (Appendix E). Positions beyond Qwen3-8B's 40,960-token window are extrapolated. Placement alone gives 1.3–1.9×; with four branches, its gain falls from 1.8× to 1.3× as the shared prefix shrinks from 85% to 32%. Admission adds 7–16% at 75–85% prefixes, 47–73% at 32–49%, where private histories fill memory, and nothing at 95%. The best of three static caps takes 7–14% longer, and retention changes rollout time by at most 2.4%.

4.5 Scaling

Figure 8 changes one factor at a time. Canopy's speedup grows from 1.25× on one instance to 2.0× on 16 and 32, peaks at 2.5× when the group size is close to the number of instances, and reaches 1.8–2.0× once more than 128 trajectories per GPU exceed KV memory, where a cap of 32 helps at 192–256 but is 17–23% slower than verl at 64–128. The gain falls from 1.9× at a median tool latency of 0.1–1 s to 1.5× at 12 s and none at 30 s, and grows with the shared prefix, from 1.4× at 1k tokens to 2.2× at 32k.

4.6 Robustness of admission

Estimates. A previous-step estimate 2× too large changes rollout time by at most 0.4%; one 4× too large makes admission too cautious (+11%); one 2–4× too small makes Canopy behave as if it had no estimate (+4–27%). Setting λ to 0.8 or 0.9 changes Canopy's time by at most 0.7%; letting the charge exceed capacity ($\lambda = 1.2$) slows the program-level scheduler by 55% on local search. Using estimated instead of exact output lengths changes Canopy's time by -1.7% to $+2.5\%$.

Models. With Qwen3-4B, 14B, 32B, and 30B-A3B, Canopy is 1.55–2.47× faster than verl and 1.14–1.46× faster than the program-level scheduler (Table 8).

4.7 Boundary cases and negative results

In the SWE-style workloads, trajectories mostly reuse their own history: in the short variant, placement alone gives 2.8× because verl's routing puts a sample of every group on each instance; in the long variant (up to 39k tokens), memory runs out even with placement (1.25×), and admission gives 4.29×. On math reasoning, whose input is small next to its output, admission charges each request its

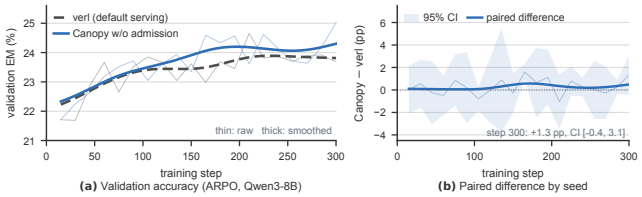


Figure 9: ARPO training of Qwen3-8B in verl for 300 steps with verl's rollout serving and with Canopy's placement and retention (no admission), three seeds each. (a) Validation exact match; (b) mean paired difference with its 95% confidence interval, which contains zero at 19 of 20 evaluations.

1–8k-token output and holds back trajectories that would fit, so Canopy is 13% slower than verl while TreeSticky is 5% faster. Mixing trees of 8, 16, and 32 leaves lowers the gain to 1.68× and 1.41× over verl (1.87× and 1.64× with 16 leaves; Table 9). Whole-group placement leaves the busiest instance 12–19% more requests than the average.

4.8 Effect on learning

Placement and retention do not detectably change learning. We checked them with 300-step ARPO training of Qwen3-8B in verl against verl's default rollout serving, three paired seeds each; the runs discard ARPO's entropy-triggered branches and test neither admission nor speed (Appendix G). At step 300 the mean difference in validation exact match is $+1.3$ points (95% CI $[-0.4, 3.1]$; Figure 9): no detectable effect.

5 Related Work

RL training systems [Sheng et al., 2025, Fu et al., 2025, The slime Team, 2025, Sheng et al., 2026, Gao et al., 2026a, Yu et al., 2026, Gao et al., 2026b, Wang et al., 2026] overlap rollout with training, repack long tails, or place stages on suitable hardware, and tree-structured RL [Ji et al., 2026, Li et al., 2026b, Dong et al., 2026b,a, Zong et al., 2026] creates the groups that Canopy serves; in all of them the inference engines still receive independent requests. Shared-prefix training [Wang et al., 2025, Liu et al., 2025, Zhang et al., 2026], including our PrefixFlow [Li et al., 2026a], computes a group's shared prefix once in the training pass; Canopy does so in rollout, and the two compose. Seer [Qin et al., 2026] deliberately splits each GRPO group across instances to balance load; Canopy keeps groups together and controls memory by admission. Prefix-aware serving [Kwon et al., 2023, Zheng et al., 2024, Srivatsa et al., 2025, The SGLang Team, 2024, Qin et al., 2025] knows what is cached and where, but not whether or for how long it will be read again, and agent-aware KV managers and program-level schedulers [Abhyankar et al., 2024, Li et al., 2025, Bian et al., 2025, Pan et al., 2025, Kang et al., 2026] decide per request, tool call, or program. None of them treats the group as the unit of placement or admission (Appendix H).

6 Conclusion and Limitations

Canopy passes what the RL controller knows about groups, their lifetimes, and the previous step's memory to the serving stack, which keeps each group on one instance and admits trajectories by a group-level memory charge; together these remove nearly all excess prefill, while eviction order and preemption add little. **Limitations.** Our steps are synthetic: length ranges are set by hand from small samples, output lengths are known, and the previous-step estimate comes from a second sample rather than from a drifting policy. Admission is not in verl or the training runs, so we did not measure RL step time with it. Baselines are re-implementations (Appendix C lists the differences). Main results use one node, engine, and GPU type, with synchronous rollout. We did not evaluate multi-agent or compacting agents with this implementation.

Acknowledgments and Disclosure of Funding

We thank the reviewers and the area chair for their feedback. Experiments used GPU clusters provided by Tencent; this work received no other funding. Feiyuan Zhang did this work during an internship at Tencent, and Ziniu Li, Yuhao Jiang, Guanhua Huang, and Yihao Liu are employed by Tencent. The authors declare no other competing interests.

References

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 81–95. PMLR, 2024. URL https://proceedings.mlr.press/v235/abhyankar24a.html.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025.

Zhuohang Bian, Feiyang Wu, Zhuoran Li, Teng Ma, and Youwei Zhuo. TokenCake: A KV-cache-centric serving framework for LLM-based multi-agent applications, 2025. URL https://arxiv.org/abs/2510.18586. Accepted to EuroSys 2027.

Mingyang Chen, Linzhuang Sun, Tianpeng Li, Haoze Sun, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. ReSearch: Learning to reason with search for LLMs via reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38, pages 85287–85307. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/7b3a0d3a0864b66f229c0a9c84f9e950-Paper-Conference.pdf.

Yeqiu Chen, Ziyan Liu, Zhenxin Huang, Runquan Gui, Hong Wang, and Lei Liu. ArborKV: Structure-aware KV cache management for scaling tree-based LLM reasoning. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 15560–15573. PMLR, 2026. URL https://proceedings.mlr.press/v306/chen26cd.html.

Guanting Dong, Licheng Bao, Zhongyuan Wang, Kangzhi Zhao, Xiaoxi Li, Jiajie Jin, Jinghan Yang, Hangyu Mao, Fuzheng Zhang, Kun Gai, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Toward generalized web agent training: A deep dive into entropy-balanced reinforcement learning. In *Proceedings of the ACM Web Conference 2026 (WWW '26)*, pages 2126–2137. ACM, 2026a. doi: 10.1145/3774904.3792301. Preprint titled "Agentic Entropy-Balanced Policy Optimization", arXiv:2510.14545.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Agentic reinforced policy optimization. In *International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=TX4k7BF6a0.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AREAL: A large-scale asynchronous reinforcement learning system for language reasoning. In *Advances in Neural Information Processing Systems*, volume 38, pages 36256–36282. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/33c00862bfa29ac72ecf630a41e19352-Paper-Conference.pdf.

Jiading Gai, Shuai Zhang, Xiang Song, Bernie Wang, and George Karypis. DualKV: Shared-prompt flash attention for efficient RL training with large rollouts and long contexts, 2026. URL https://arxiv.org/abs/2605.15422.

Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollPacker: Taming long-tail rollouts for RL post-training with tail batching. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 849–866, Renton, WA, 2026a. USENIX Association. URL https://www.usenix.org/conference/nsdi26/presentation/gao-wei.

Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, Ju Huang, Siran Yang, Yongbin Li, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollArt: Disaggregated multi-task agentic RL training at scale. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 863–881, Seattle, WA, 2026b. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/gao.

Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History doesn't repeat itself but rollouts rhyme: Accelerating reinforcement learning with RhymeRL. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26)*. ACM, 2026. doi: 10.1145/3779212.3790172. Preprint: arXiv:2508.18588.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online), 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.580. URL https://aclanthology.org/2020.coling-main.580/.

Zhenyu Hou, Ziniu Hu, Yujiang Li, Rui Lu, Jie Tang, and Yuxiao Dong. TreeRL: LLM reinforcement learning with on-policy tree search. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://arxiv.org/abs/2506.11902.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Wenkai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: A ray-based easy-to-use, scalable and high-performance RLHF framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 656–666, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.48. URL https://aclanthology.org/2025.emnlp-demos.48/.

Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. Tree search for LLM agent reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ZpQwAFhU13.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O. Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes, 2024. URL https://arxiv.org/abs/2402.05099.

Hao Kang, Ziyang Li, Weili Xu, Xinyu Yang, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. ThunderAgent: A fast, simple, and program-aware agentic inference system. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 55630–55657. PMLR, 2026. URL https://proceedings.mlr.press/v306/kang26j.html. arXiv:2602.13692.

Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. Fact, fetch, and reason: A unified evaluation of retrieval-augmented generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*, 2025.

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 29th Symposium on Operating Systems Principles*, pages 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live, 2025. URL https://arxiv.org/abs/2511.02230.

Pengbo Li, Feiyuan Zhang, Guangming Sheng, Guangxin He, Di Chai, Ziniu Li, Taiqiang Wu, Han Tian, Wenyu Mao, Binhang Yuan, and Kai Chen. PrefixFlow: Training-time KV caching for schedule-level prefix reuse in LLM RL training. In *Proceedings of the 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC '26)*, 2026a. To appear. arXiv:2606.01143.

Yizhi Li, Qingshui Gu, Zhoufutu Wen, Ziniu Li, Ruibin Yuan, Tianshun Xing, Shuyue Guo, Tianyu Zheng, Xin Zhou, Xingwei Qu, Wangchunshu Zhou, Zheng Zhang, Wei Shen, Wei Xue, Qian Liu, Chenghua Lin, Jian Yang, Ge Zhang, and Wenhao Huang. TreePO: Enhancing policy efficacy and inference efficiency with tree modeling. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 66963–66976. PMLR, 2026b. arXiv:2508.17445.

Zikang Liu, Tongtian Yue, Yepeng Tang, Longteng Guo, Junxian Cai, Qingbin Liu, Xi Chen, and Jing Liu. Prefix grouper: Efficient GRPO training through shared-prefix forward, 2025. URL https://arxiv.org/abs/2506.05433.

llm-d Project. KV-cache wins you can see: From prefix caching in vLLM to distributed scheduling with llm-d. https://llm-d.ai/blog/kvcache-wins-you-can-see, 2025. Blog post, September 24, 2025.

Han Lu, Zichen Liu, Shaopan Xiong, Yancheng He, Wei Gao, Yanan Wu, Weixun Wang, Jiashun Liu, Yang Li, Haizhou Zhao, Ju Huang, Siran Yang, Xiaoyang Li, Yijia Luo, Zihe Liu, Ling Pan, Junchi Yan, Wei Wang, Wenbo Su, Jiamang Wang, Lin Qu, and Bo Zheng. Part II: ROLL flash – accelerating RLVR and agentic training with asynchrony, 2025. URL https://arxiv.org/abs/2510.11345. arXiv preprint arXiv:2510.11345.

Zaifeng Pan, Ajjkumar Dahyalal Patel, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *Advances in Neural Information Processing Systems*, volume 38, pages 126246–126265. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/b7971d31a7d5eb0f1eed2f8f6f368195-Paper-Conference.pdf.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A. Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378/.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous LLM reinforcement learning. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 883–901, Seattle, WA, 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/qin.

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. URL https://arxiv.org/abs/2402.03300. arXiv preprint arXiv:2402.03300.

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. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys '26)*, Edinburgh, Scotland, UK, 2026. ACM. doi: 10.1145/3767295.3803580. URL https://doi.org/10.1145/3767295.3803580.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=meKEKDhdnx.

The SGLang Team. SGLang v0.4: Zero-overhead batch scheduler, cache-aware load balancer, faster structured outputs. https://lmsys.org/blog/2024-12-04-sglang-v0-4/, 2024. LMSYS Org blog, December 4, 2024.

The slime Team. slime: An SGLang-native post-training framework for RL scaling. https://www.lmsys.org/blog/2025-07-09-slime/, 2025. LMSYS Org blog, July 9, 2025. Code: https://github.com/THUDM/slime.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

vLLM Team. vLLM optimization and tuning: Preemption. https://docs.vllm.ai/en/v0.17.1/configuration/optimization/, 2026. Accessed 2026-05-07.

Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Liang Huang, Can Tang, Xiaojiang Zhang, Junyi Peng, Li Wan, Haotian Zhang, and Bin Chen. Tree training: Accelerating agentic LLMs training via shared prefix reuse, 2025. URL https://arxiv.org/abs/2511.00413.

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, and Yu Wang. DynaRL: Flexible and dynamic scheduling of Large-Scale reinforcement learning training. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 847–862, Seattle, WA, 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/wang-yuanqing.

Tianyuan Wu, Lunxi Cao, Yining Wei, Wei Gao, Yuheng Zhao, Dakai An, Shaopan Xiong, Zhiqiang Lv, Ju Huang, Siran Yang, Yinghao Yu, Jiamang Wang, Lin Qu, and Wei Wang. Weave: Efficient co-scheduling for disaggregated RL post-training. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, Seattle, WA, 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/wu-tianyuan.

Ryan Xu, Atlas Zhao, David Bao, and Frank Du. WAR: Workload-aware rollouts for synchronous agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2607.17299.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren,

Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388. arXiv preprint arXiv:2505.09388.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Chao Yu, Yuanqing Wang, Zhen Guo, Hao Lin, Si Xu, Hongzhi Zang, Quanlu Zhang, Yongji Wu, Chunyang Zhu, Junhao Hu, Zixiao Huang, Mingjie Wei, Yuqing Xie, Ke Yang, Bo Dai, Zhexuan Xu, Jiakun Du, Xiangyuan Wang, Xu Fu, Letong Shi, Zhihao Liu, Kang Chen, Weilin Liu, Gang Liu, Boxun Li, Jianlei Yang, Zhi Yang, Guohao Dai, and Yu Wang. RLinf: Flexible and efficient large-scale reinforcement learning via macro-to-micro flow transformation. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 829–846, Seattle, WA, 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/yu-chao.

Jiarui Zhang, Yuchen Yang, Ran Yan, Zhiyu Mei, Liyuan Zhang, Daifeng Li, Wei Fu, Jiaxuan Gao, Shusheng Xu, Yi Wu, and Binhang Yuan. AReaL-DTA: Dynamic tree attention for efficient reinforcement learning of large language models. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. arXiv:2602.00482.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Zefang Zong, Dingwei Chen, Yang Li, Qi Yi, Bo Zhou, Chengming Li, Bo Qian, Peng Chen, and Jie Jiang. AT^{2}PO: Agentic turn-based policy optimization via tree search. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24120–24143. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.acl-long.1106/.

A Implementation Details

The engine side is a set of hooks for vLLM V1 (0.20.1), about 470 lines of Python loaded at server start, with no change to model code or kernels. The descriptor travels in `SamplingParams.extra_args`. The hooks wrap six vLLM functions, described under *Engine hooks* below. Requests without a descriptor follow vLLM's default path.

The controller side runs where the agent loops run. It keeps a registry of the step's groups, trajectories, and branch points, fills in the descriptor, places groups, and runs admission. All timing results come from our rollout driver, which runs the agent loops of a rollout step against the same vLLM instances for every policy (§4.1); admission and the program-level baseline are implemented there. The training runs (§4.8) use an earlier verl integration of Canopy, which places and retains by tree but has no admission (Appendix G).

The controller does constant work per request; admission runs when a request arrives or finishes and at least every 0.5 s. The engine hooks touch only the blocks of the request being freed or preempted. On the math workload, rollout time with the hooks (Canopy w/o admission) and without them (TreeSticky) is 241.5 s in both cases.

Descriptor. The controller puts the descriptor of Equation 1 into vLLM's `SamplingParams.extra_args` as four fields: the group id, the shared length b_u in tokens, a flag for the trajectory's last request, and a flag for the group's last request. The engine rounds b_u down to whole KV blocks. Our driver sets $done_u$ when it issues a request, so the flag is missed when a group's last requests run concurrently, which happens in every tree run; the group's references then expire after 60 s or are dropped by failed allocations.

Engine hooks. The hooks wrap six vLLM V1 functions, in four roles. (1) *Freeing a finished request's blocks.* If the request is the last of its trajectory, its blocks after the first b_u tokens go to the head of the free-block queue, so they are evicted first; if it is also the last of its group, all its blocks do. Other blocks are freed in vLLM's order. With shared-prefix references enabled, the hook also adds one reference to the cached blocks of a live group's first b_u tokens and records the group, so these blocks leave the free queue. (2) *Allocating slots.* If an allocation fails while references are held, the hook drops the least recently used group's references and retries until the allocation succeeds or no reference is left. In the default mode any failed allocation can drop references; in the strict variant only allocations of running requests can. (3) *Preempting a request.* Before vLLM frees a preempted request of a live group, the hook adds a reference to its cached blocks up to $\min(b_u, \text{computed tokens})$; the reference is dropped when the request runs again. (4) *Scheduling and resetting the prefix cache.* At each scheduling step the hook drops a group's references after 60 s without a request from the group, and drops references, oldest first, while the engine has waiting requests but nothing running; resetting the prefix cache, as when policy weights change, drops all references. Figure 10 shows the resulting block states for one group.

Admission. The scheduler implements Equations 2–3. A trajectory's charge is its context, i.e., input plus output of its current request; the shared prefix is charged once per group placed on the instance until the group is done. The previous-step estimate $\hat{w}$ is the peak, over turn index, of the step's total private KV divided by its number of trajectories, computed from a second sample of the workload. Trajectories whose first request is issued as soon as their parent's request finishes (in-turn branches) are reserved at $\hat{w}$ when their group is admitted. The scheduler runs whenever a request arrives or finishes and at least every 0.5 s. It first pauses tool-waiting trajectories on over-committed instances, largest private context first, then starts waiting trajectories in FIFO order on their group's instance (or, for a group's first trajectory, on the instance with the lowest projected charge relative to capacity). The program-level baseline uses the same code without the shared-prefix, group-lifetime, estimate, reservation, group-placement, and finished-first-eviction features: it charges each trajectory its whole context, and a paused trajectory may resume on the instance with the lowest charge, preferring its previous one.

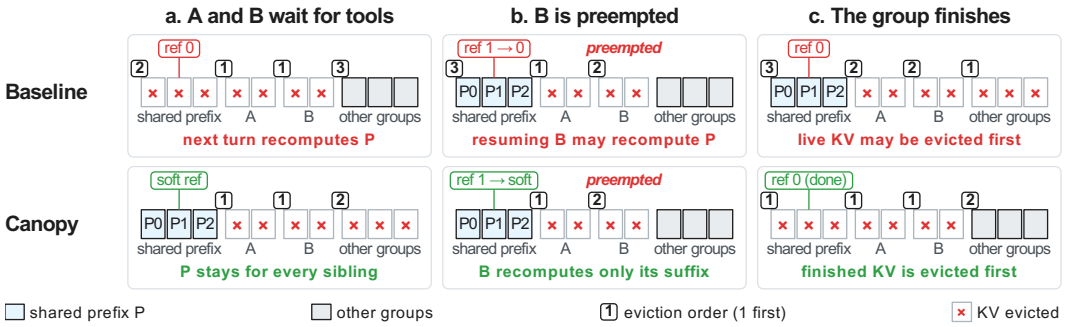


Figure 10: KV blocks of one group on one GPU (schematic). The baseline is a request-level KV cache that evicts unreferenced blocks in LRU order. P0–P2 hold the shared prefix P, which the next turn of every sibling reads; A and B are the private suffixes of two trajectories; gray blocks belong to live trajectories of other groups. Numbers give the eviction order (1 first; unnumbered blocks are in use or held by a reference); the engine needs seven blocks in (a) and (c) and four in (b), and evicts the first ones in this order. (a) While A and B wait for tools, no running request references P, and the baseline's LRU order puts P before newer blocks; Canopy holds P with a soft reference. (b) Memory is short: the engine evicts A's cached blocks and then preempts B. The baseline frees P together with B's suffix, so P is next in line and resuming B may recompute it; Canopy first adds a soft reference to P, so only B's suffix is freed. (c) After the group finishes, its blocks are the most recently freed, so the baseline may evict older blocks of live trajectories first; Canopy evicts the finished group's blocks first. The main results use only (c); (a) and (b) are on in Canopy w/o admission (§4.3).

B Workloads and Method

Parameters. Table 2 gives the parameters per instance. $U[a, b]$ is uniform; $LN(m, \sigma; [a, b])$ is log-normal with median m and shape σ, clipped to $[a, b]$. In the tree workloads, each tree's prompt is shared by four root trajectories; after two turns, each root branches into four children, which gives 16 leaves, and every trajectory runs six turns. In the in-turn variant, the third turn is cut at 20–60% of its output, right after the tool result, and the four children continue from that point without a tool call.

Table 2: Workload parameters per instance. Lengths are in tokens; tool latencies are per call.

Workload	Groups	Size	Prompt	Turns	Tool output	Output per turn	Tool latency (s)
Tree, turn	8	16	$U[300, 600]$	6	$U[600, 1500]$	$U[100, 400]$	$U[0.2, 1.5]$
Tree, in-turn	8	16	$U[300, 600]$	6	$U[600, 1500]$	$U[100, 400]$	$U[0.2, 1.5]$
Local search	32	8	$U[300, 600]$	$U[3, 8]$	$U[600, 1500]$	$U[100, 400]$	$U[0.2, 1.5]$
Web search	16	8	$U[800, 1500]$	$U[3, 8]$	$U[1k, 4k]$	$U[100, 600]$	$LN(8, 0.7; [1, 35])$
Long context	24	8	$U[12k, 20k]$	1	none	$U[128, 512]$	none
SWE, short	8	8	$U[1.5k, 4k]$	$U[6, 14]$	$LN(900, 0.8; [50, 3k])$	$LN(60, 0.8; [10, 400])$	$LN(0.3, 1.0; [0.02, 4.5])$
SWE, long	6	8	$U[1.5k, 4k]$	$U[10, 24]$	$LN(900, 0.8; [50, 3k])$	$LN(60, 0.8; [10, 400])$	$LN(0.3, 1.0; [0.02, 4.5])$
Math	16	8	$U[100, 400]$	1	none	$LN(4k, 0.5; [1k, 8k])$	none

Measured trajectories and the parameters we used. We collected small samples of Qwen3-8B agent trajectories and set the ranges in Table 2 by hand to roughly match them; we did not fit the distributions. Table 3 compares the two. Math output lengths are shorter than those reported by Li et al. [2026a].

Correctness. Canopy changes where and when requests run and which KV blocks stay cached, not what they compute. We checked this with an earlier version of Canopy, without admission, on a seven-request tree at temperature 0: verl's routing, TreeSticky, and Canopy produced identical outputs. We did not repeat the check for admission, which only delays when a trajectory starts or resumes.

Rollout driver. Each engine runs at most 256 sequences and 8192 batched tokens per iteration and uses 90% of GPU memory for weights and KV cache. The driver builds the request graph of a step from a seed, starts one vLLM server per instance (a development build of vLLM 0.20.1, 0.20.1.dev615), and runs one experiment as a sequence of (policy, repetition) runs on the same servers. Before each run it resets every server's prefix cache and checks that the reset succeeded; one unmeasured warm-up run comes first. Each request asks for exactly its output length with

Table 3: Statistics of the collected Qwen3-8B trajectories and of the workloads that we generated (medians unless noted). "Shared" is the context that the request shares with earlier requests; "new" is the rest.

Workload	Sample	Measured	Used
Long context	LongBench v2 [Bai et al., 2025], 32 questions	shared 16.4k (10.2–24.0k), new 1.1k; reuse 93%	prefix 12–20k, output 0.1–0.5k; reuse 87%
Web search	FRAMES [Krishna et al., 2025] + search API, 32 questions (scripted search, read, answer)	shared 6.5k, new 0.7k; reuse 88%; tool 12.7 s	tool output 1–4k; reuse 75%; tool 8 s
Local search, trees	local-retriever QA traces	sibling-shared ≈1.2k, history 1–8k; reuse 72–78%	prompt 0.3–0.6k, tool output 0.6–1.5k; reuse 77–83%
SWE	SWE-bench Verified [Jimenez et al., 2024], 12 tasks (10 collected completely)	history 1–24k, observation ≈1.2k, action ≈27; reuse ≈90%	observation 0.9k, action 60; reuse 88–92%

`ignore_eos`, and the driver checks that the returned length matches. The next turn's input contains the returned tokens. Tool latencies are simulated with timers. Every run processes the same requests and token counts (sampling at temperature 1 makes the generated contents differ); the driver records the prefill tokens computed by the engines, the preemptions, the KV usage every 2 s, and for each request its instance, start, and end. Policies rotate their order across repetitions. Long workloads run each repetition as a separate experiment on the same type of node. Multi-node experiments report correct times but undercount engine statistics, so we report no prefill numbers for them.

C Baselines

Prefix-aware routing uses the SGLang router's default thresholds: if the largest in-flight count exceeds the smallest by more than 32 and by a factor of more than 1.5, it sends the request to the least loaded instance; otherwise it sends it to the instance with the longest cached prefix match if the match covers more than half of the prompt, and to the instance with the smallest cached tree if not. Its approximate prefix tree only grows; the SGLang router also evicts entries. *TTL* holds a reference on a trajectory's blocks for 2 s after each turn and releases it when the trajectory's next turn arrives or the time runs out; Continuum instead sets the time per tool call from a cost model. *Canopy w/o admission* re-implements placement and retention in our driver and engine hooks; the earlier implementation kept only the first eight blocks of a preempted request, while the re-implementation keeps the whole shared prefix. *Program-level (ThunderAgent-style)* charges each trajectory its full context, as ThunderAgent's released code (commit 7ddc861) does: the code's term for prefix-cache savings measured from vLLM stays at zero, because its update was disabled in February 2026. It differs from that code in five ways. It checks every 0.5 s instead of 5 s. It discounts tool-waiting trajectories by halving every 5 s after 1 s, when pausing and when starting; ThunderAgent's 2^{-t} discount is off by default and only affects resuming. It pauses the tool-waiting trajectory with the largest context first; ThunderAgent pauses tool-waiting programs with the smallest context first and then marks running ones for pausing. It retries paused trajectories in FIFO order, preferring the instance they left; ThunderAgent resumes programs with a pending request, then new ones, then tool-waiting ones, and places the largest first on the instance with the most free memory. Like Canopy, it knows each request's output length. Table 10 runs it in the same experiments as a re-implementation of ThunderAgent's released scheduler with its defaults (5 s checks, no discount, the smallest tool-waiting program paused first, running programs marked for pausing, best-fit-decreasing resume): ours is faster on both trees and local search, and the released one on web search and long-context GRPO. The comparisons in §4 use ours; §4.1 also gives Canopy's speedup over the faster of the two. *Static caps* admit whole groups while at most K root trajectories are active on an instance, and free a group's slots when it finishes; we tried 8, 16, and 24 for the trees (each tree has four roots), 64, 128, and 192 for local search, 32, 64, and 96 for web search, and 48, 96, and 128 for long-context QA. Each cap experiment also ran Canopy, and we scale the cap's time by the ratio of Canopy's times to compare it with the main experiment.

We also ran Preble [Srivatsa et al., 2025], SGLang [Zheng et al., 2024], and KVFlow [Pan et al., 2025] on their own back ends with one 188-request trace (Qwen3-8B, two instances, three runs); each completed every request, in 10.8, 12.0, and 12.2 s on average. These systems differ from Canopy's

in engine, scheduler, and memory manager, so their times do not isolate the scheduling policy, and we do not compare them with Canopy. The main experiments instead run re-implementations of the corresponding policies on the same vLLM engine: prefix-aware routing as in the SGLang router, TTL as in Continuum, and the program-level scheduler as in ThunderAgent; §5 and Table 13 compare what Preble and KVFlow know and change.

D Where the Prefill Goes

Figure 11 traces one group through a rollout step under request-level serving and under Canopy. For every run we regenerate the request graph from its seed; the regenerated input length of every request matched the logged length. We then process the requests in the order in which they started on their instances. A request's shared prefix is available on its instance if an earlier request on that instance covered it, and its private history is available if its parent ran on the same instance. The tokens that were not available but had been computed elsewhere are *duplicated on another instance*; their sum plus the minimum gives the placement minimum. The engine-computed prefill minus the placement minimum is *recomputed after eviction or preemption*. vLLM separately counts the prefill computed when a request is first scheduled; the difference to the engine total, which is recomputation after preemption, is at most 2% of the minimum in every main and memory-pressure run. Table 4 gives the breakdown for the main runs.

Table 4: Prefill duplicated on another instance (dup.) and recomputed after eviction or preemption (rec.), as multiples of the minimum (eight instances, median of three runs). The total is $1 + \text{dup.} + \text{rec.}$

Workload	verl		Prefix-aware		TreeSticky		Program-level		Canopy	
	dup.	rec.	dup.	rec.	dup.	rec.	dup.	rec.	dup.	rec.
Tree search (turn)	0.34	1.84	0.29	0.61	0.00	0.64	0.60	0.04	0.00	0.01
Tree search (in-turn)	0.41	1.06	0.26	0.02	0.00	0.01	0.73	0.04	0.00	0.01
Local search (GRPO)	0.08	2.87	0.30	2.60	0.00	2.48	0.79	0.22	0.00	0.01
Web search (GRPO)	0.09	2.65	0.18	2.41	0.00	2.23	1.18	0.84	0.00	0.04
Long context (GRPO)	6.94	0.00	0.00	0.00	0.00	0.00	4.91	0.00	0.00	0.00
SWE agent (short)	0.22	1.49	0.19	0.03	0.00	0.01	0.45	0.07	0.00	0.01
SWE agent (long)	0.12	6.74	0.63	2.16	0.00	3.79	0.98	0.34	0.00	0.05
Math (control)	4.66	0.00	0.68	0.05	0.00	0.07	3.61	0.00	0.00	0.10

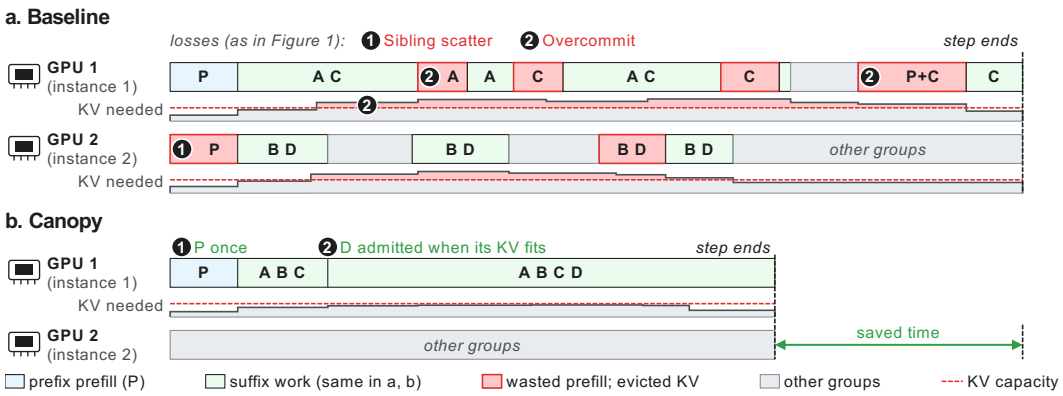


Figure 11: One rollout step of one group (shared prefix P, trajectories A–D) on two GPUs, each running one inference-engine instance; schematic. Each row is one GPU: blue blocks prefill the shared prefix; green blocks are the trajectories' suffix work (decoding and prefilling new tool results), the same in (a) and (b); red blocks compute prefill again; gray blocks serve other groups while this group's trajectories wait for tool responses. Letters in a block name the trajectories whose work the GPU runs in that interval: in (a), A and C share GPU 1 and B and D share GPU 2; in (b), GPU 1 runs A, B, and C, then all four once D is admitted. Below each GPU, the KV memory its live trajectories need (above capacity: evicted). (a) The baseline, request-level routing as in current RL frameworks, shows the two losses of Figure 1; (b) Canopy keeps the group on GPU 1 and admits D when its KV fits.

E Additional Results

Scaling settings. Figure 8 uses one run per point. (a) Tree search (turn), 8 trees per instance; 16 and 32 instances span two and four nodes. (b) Single-turn groups with an 8–12k-token shared prompt, 1,024 trajectories on four instances. (c) Local search, 16–256 trajectories on one GPU. (d) Web search, 128 trajectories on one GPU. (e) Single-turn groups of eight, 96 groups on four instances; the program-level scheduler splits groups and computes about as much prefill as verl. On one GPU placement has no effect, so (c) and (d) show only the program-level scheduler, an untuned static cap of 32 trajectories, and Canopy. Canopy w/o admission is within 1.9% of TreeSticky and is not drawn.

This appendix gives the results behind §4: rollout times of the main runs (Table 5), admission delay and retained KV (Table 6), memory pressure (Figure 12), retention mechanisms with admission on eight instances (Table 7), other models (Table 8), and the follow-up runs below.

Follow-up runs on a second cluster. Tables 9 and 10 come from runs on a second cluster of eight-GPU H20 nodes, with the same software (the same vLLM build and the same CUDA, NCCL, and cuDNN libraries) and the same KV capacity per instance (31,167 blocks). On the five workloads that ran on both clusters, the program-level scheduler and Canopy take within 2.2% of their times in the main runs. With mixed tree sizes, the busiest instance receives 12.5% more requests than the average under TreeSticky and the static cap, and 12.5% (turn) and 19% (in-turn) more under Canopy; with 16-leaf trees, all three place eight trees on every instance.

Long trajectories that branch late. Table 12 gives the runs of §4.4. Each group runs as one trajectory for 4–19 turns and then forks; we vary one factor at a time around 96k tokens, eight branches, and an 85% shared prefix, and set the number of groups so that all shared prefixes fit in KV memory but the step needs 1.3–1.9× of it. Positions beyond Qwen3-8B's 40,960-token window are extrapolated; outputs are discarded and their lengths fixed, so only time and memory matter. Jobs that compare verl, prefix-aware routing, or Canopy with TreeSticky skip the warm-up runs, which changes TreeSticky's time by less than 0.3%. In the main runs our implementation fixed a group's b at its first request, so after a fork each branch was also charged the history it shares with its siblings: at the end of a step, 21% and 43% too much on the two tree workloads of §4.2 and 3.6–5.6× on these forks. The Canopy column charges each request's own b, as §3.1 defines it; we added this after seeing the "first b" column, and we did not re-run the main tree workloads, for which charging the largest b of a group would undercount by 7–19% because their four roots fork separately.

Table 5: Rollout time in seconds on eight instances (median of three runs). Bold: fastest system other than the cap. †Best of three static caps, chosen after the fact and scaled to this experiment.

Workload	verl	PAR	TTL	Sticky	Program	Cap†	w/o adm.	Canopy
Tree search (turn)	315	227	314	221	205	174	221	**169**
Tree search (in-turn)	247	166	240	148	188	155	**148**	151
Local search (GRPO)	798	756	798	730	483	362	730	**365**
Web search (GRPO)	1023	985	1027	928	835	590	918	**551**
Long context (GRPO)	549	155	548	155	432	160	155	**152**

Table 6: Admission delay and retained KV in the main runs (median over three runs). Wait: time a request waited beyond its tool latency (n/a: single-turn, no tool calls). Last start: when the last trajectory started, as a share of rollout time. Peak refs: blocks held by soft references, sum of per-instance peaks over all KV blocks; they are dropped whenever an allocation fails.

Workload	Canopy				Canopy w/o admission	
	wait p99 (s)	wait max (s)	last start	preempt.	preempt.	peak refs
Tree search (turn)	63.9	72	69%	0	8	11%
Tree search (in-turn)	37.9	53	73%	0	0	18%
Local search (GRPO)	0.1	127	62%	0	55	3%
Web search (GRPO)	0.0	230	62%	0	70	4%
Long context (GRPO)	n/a	n/a	0%	0	0	77%
SWE agent (short)	0.0	0	0%	0	0	4%
SWE agent (long)	0.0	91	64%	0	8	3%

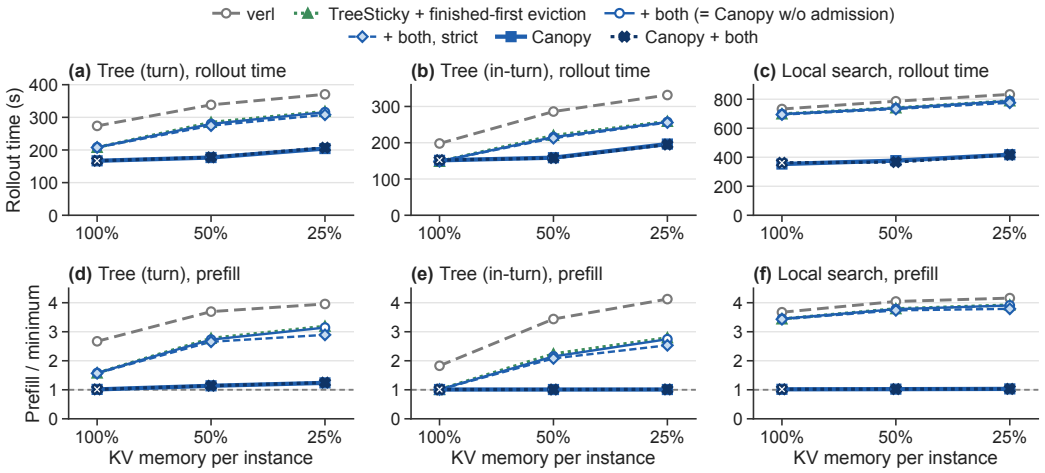


Figure 12: Memory pressure on four instances (Qwen3-8B, H20): rollout time (top) and prefill computed as a multiple of the minimum (bottom) with 100%, 50%, and 25% of each instance's KV memory. At 100%: three runs; Canopy and Canopy + both come from a separate experiment with the same workload (hollow). At 50% and 25%: all systems in one experiment, two runs for the trees and one for local search. "Canopy + both" adds shared-prefix references and prefix-preserving preemption to Canopy.

Table 7: Canopy against Canopy with shared-prefix references and prefix-preserving preemption ("+ both"), eight instances, median rollout time in seconds (three runs, in experiments separate from the main ones). verl and Canopy reproduce the main results within 2.3%.

Workload	verl	Canopy	Canopy + both
Tree search (turn)	308	169	168
Tree search (in-turn)	252	151	151
Local search (GRPO)	799	364	365
Web search (GRPO)	1024	541	546
Long context (GRPO)	546	152	152
SWE agent (long)	1312	304	301
Math (control)	254	286	286

Table 8: Speedup over verl for different models, eight GPUs, one run per cell (Qwen3-8B: the main runs, median of three). Qwen3-32B uses two GPUs per instance (four instances).

Model	Workload	verl (s)	Sticky	Program	w/o adm.	Canopy
Qwen3-4B	Tree search (turn)	200	1.49×	1.36×	1.51×	1.55×
Qwen3-4B	Local search (GRPO)	526	1.08×	1.54×	1.08×	1.89×
Qwen3-8B	Tree search (turn)	315	1.42×	1.54×	1.42×	1.87×
Qwen3-8B	Local search (GRPO)	798	1.09×	1.65×	1.09×	2.19×
Qwen3-14B	Tree search (turn)	604	1.43×	1.73×	1.43×	2.37×
Qwen3-14B	Local search (GRPO)	1383	1.07×	1.70×	1.07×	2.47×
Qwen3-32B (TP 2)	Tree search (turn)	574	1.30×	1.57×	1.31×	2.03×
Qwen3-32B (TP 2)	Local search (GRPO)	1481	1.07×	1.71×	1.07×	2.45×
Qwen3-30B-A3B	Tree search (turn)	250	1.24×	1.38×	1.25×	1.64×
Qwen3-30B-A3B	Local search (GRPO)	547	1.05×	1.40×	1.06×	1.70×

Table 9: Mixed tree sizes: each step mixes trees with 8, 16, and 32 leaves (32, 24, and 12 trees; the same requests, tokens, and minimum prefill as the 64 trees of 16 leaves in the main results). Speedup over verl (median of three runs); in parentheses, the main experiment with 16-leaf trees. Cap 16 is the best static cap of the main tree experiments (four 16-leaf trees per instance).

Workload	verl	TreeSticky	Cap 16	Program-level	Canopy
Tree search (turn)	1.00× (1.00)	1.19× (1.42)	1.54× (1.81)	1.52× (1.54)	1.68× (1.87)
Tree search (in-turn)	1.00× (1.00)	1.47× (1.67)	1.36× (1.59)	1.32× (1.31)	1.41× (1.64)

Table 10: Program-level baselines on eight instances (median of three runs; all three policies in the same experiments). *Ours*: the program-level policy of the main results. *Released*: a re-implementation of the released program-level scheduler of Kang et al. [2026] with its defaults (5 s checks, no discount, smallest tool-waiting program paused first, running programs marked for pausing, best-fit-decreasing resume). The last column scales the released policy's time to verl's by Canopy's ratio to the main experiment.

	Rollout time (s)			Canopy speedup over		Released
Workload	Ours	Released	Canopy	Ours	Released	vs. verl
Tree search (turn)	205	285	169	1.21×	1.69×	1.11×
Tree search (in-turn)	188	245	151	1.25×	1.62×	1.01×
Local search (GRPO)	486	569	357	1.36×	1.59×	1.37×
Web search (GRPO)	838	738	546	1.54×	1.35×	1.37×
Long context (GRPO)	436	423	152	2.86×	2.78×	1.30×

Table 11: Long trajectories that branch late (four instances, synthetic steps): speedup over verl, with prefill as a multiple of the minimum in parentheses. Canopy charges each request its own b (§4.4); Table 12 adds the other policies.

Length	Branches	Shared prefix	verl, s (prefill)	TreeSticky	Canopy
94k	4	79k (85%)	1363 (4.53)	1.80× (1.37)	**1.93**× (1.08)
95k	4	46k (49%)	1568 (5.26)	1.59× (2.23)	**2.33**× (1.05)
95k	4	30k (32%)	1877 (5.88)	1.34× (3.34)	**2.32**× (1.05)
98k	8	93k (95%)	1465 (4.45)	**1.63**× (1.13)	1.62× (1.12)
94k	8	70k (75%)	1681 (5.02)	1.78× (1.60)	**2.07**× (1.03)
122k	8	103k (84%)	1887 (4.64)	1.92× (1.31)	**2.08**× (1.09)

Table 12: Long trajectories that branch late (four instances, Qwen3-8B, H20; synthetic steps). Each group runs as one trajectory for 4–19 turns and then forks into the given number of branches; Prefix is the history shared at the fork and its share of a trajectory. Top: speedup over TreeSticky; bottom: prefill computed as a multiple of the minimum (medians of one to three runs per policy, and of 4–11 runs for TreeSticky, which ran in every experiment as the baseline). Canopy charges each request's own b; "first b" keeps a group's b from its first request, as in the main runs. The 49% and 32% configurations ran only verl, TreeSticky, and Canopy; Table 11 gives them.

Length	Branches	Prefix	verl	Prefix-aware	TreeSticky	+ both	+ both, strict	Canopy	Canopy, first b
Speedup over TreeSticky (median over runs)									
93k	8	79k (85%)	0.55	1.02	1.00	1.00	1.00	1.10	1.00
64k	8	54k (85%)	0.56	1.00	1.00	1.01	1.02	1.11	1.05
122k	8	103k (84%)	0.52	0.68	1.00	1.01	1.01	1.08	1.09
94k	4	79k (85%)	0.56	1.00	1.00	1.00	1.02	1.07	1.05
93k	16	78k (84%)	0.63	0.96	1.00	1.00	1.01	1.16	0.94
94k	8	70k (75%)	0.56	1.00	1.00	1.00	1.00	1.16	1.10
98k	8	93k (95%)	0.61	1.00	1.00	1.00	0.99	0.99	0.94
Prefill computed, multiple of the minimum									
93k	8	79k (85%)	4.78	1.38	1.36	1.35	1.33	1.08	1.00
64k	8	54k (85%)	4.87	1.39	1.32	1.28	1.27	1.02	1.00
122k	8	103k (84%)	4.64	1.46	1.31	1.31	1.30	1.09	1.00
94k	4	79k (85%)	4.53	1.37	1.37	1.37	1.30	1.08	1.00
93k	16	78k (84%)	3.58	1.42	1.39	1.39	1.40	1.02	1.00
94k	8	70k (75%)	5.02	1.59	1.60	1.60	1.60	1.03	1.00
98k	8	93k (95%)	4.45	1.18	1.13	1.13	1.16	1.12	1.00

F Transfer or Recompute

When a request runs on an instance that lacks its group's shared prefix, the instance can recompute the prefix or fetch its KV from the group's instance. Fetching pays for packing the KV, waiting for and using the link, and writing the KV into the destination's cache; recomputing pays for a prefill that competes with other requests in the destination's batches. Both costs depend on queueing and batching, so a choice between them should compare the measured cost that each path adds to the request's critical path rather than a formula. We measured the fetch path with real Qwen3-8B requests between two H20 nodes: the source computes the prefix once, the KV is packed, sent over NCCL, and written on the destination, and the destination then serves the same three sibling requests that the recompute path serves. With 31,008 and 65,536 prefix tokens, fetching took 1.80 s and 1.98 s against 8.61 s and 23.73 s for recomputing (three runs each; KV checksums and all outputs matched). Two implementations inside vLLM were slower than recomputing: vLLM's NIXL connector, which ran over TCP in our setup, on all nine shapes we measured (117 transfers, about 35 GB), and an NCCL path inside vLLM in a paired rerun (32.8 s against 30.7 s). These measurements cover the request path only. We did not implement an automatic choice, and none of the experiments in §4 moves a group's requests between instances.

Table 13 lists, for the systems of §5, what each system decides per, what it knows about future reuse, and which decisions it changes.

Table 13: Related systems by decision unit, knowledge of future reuse, and the decisions they change.

System	Decides per	Knows about future reuse	Changes
vLLM, SGLang prefix caching [Kwon et al., 2023, Zheng et al., 2024]	block	recency	eviction order; SGLang frees only uncached tokens on retraction
SGLang router, llm-d [The SGLang Team, 2024, llm-d Project, 2025]	request	cached prefixes per instance	instance choice
Preble [Srivatsa et al., 2025]	request	cached prefixes, load, cost of evicting other prefixes	instance choice, replication of hot prefixes, load redirection
Mooncake [Qin et al., 2025]	request	where KV is stored	fetch remote KV or recompute
InferCept, Continuum, TokenCake [Abhyankar et al., 2024, Li et al., 2025, Bian et al., 2025]	request in a tool call	expected tool duration	keep, offload, or drop KV while waiting
KVFlow [Pan et al., 2025]	workflow step	agent step graph from the framework	eviction order (suffix first), prefetching
ThunderAgent [Kang et al., 2026]	program	each program's full context	pause and resume, instance choice
Seer [Qin et al., 2026]	GRPO group	lengths of group members	speculative decoding; splits groups across instances with a global KV pool
Canopy	group	membership, shared length, liveness, and end of each group; previous-step estimate $\hat{w}$	instance choice, eviction order, preemption, admission

G Training Setup

The training runs use an earlier verl integration of Canopy, which places each tree on one instance (spilling a request when that instance holds more than 75% of the in-flight requests), holds the blocks of each live tree's latest request, and keeps the first eight blocks of a preempted request. It has no admission, and we have not verified that its preemption hook takes effect inside vLLM's engine processes. The training runs of §4.8 use Qwen3-8B [Yang et al., 2025], verl, and ARPO on the HotpotQA [Yang et al., 2018] training set with a local retriever. Each step has 16 prompts with 4 trajectories each; after every turn, ARPO decides from the turn's entropy whether to branch, with at most two extra branches per trajectory. The branches are generated and then discarded (they do not continue as trajectories and do not enter the batch). The learning rate warms up to 10^{-6}. Each run uses 16 H20 GPUs with two rollout instances, each with tensor parallelism 8. Validation every 15 steps uses one sample per prompt on HotpotQA, 2WikiMultiHopQA [Ho et al., 2020], MuSiQue [Trivedi et al., 2022], and Bamboogle [Press et al., 2023], and reports the macro average of exact match. The three seeds are paired: both conditions of a seed use the same initialization, data order, and settings, and differ only in the scheduling switches. Over the first 29 steps, the mean response length was 1,484–1,507 tokens in all six runs, and the audited first 30 steps had no preemption. The runs used different nodes, so we do not compare their step times.

H Related Work in Detail

RL rollout and training systems. verl [Sheng et al., 2025], OpenRLHF [Hu et al., 2025], AReaL [Fu et al., 2025], slime [The slime Team, 2025], ROLL Flash [Lu et al., 2025], Laminar [Sheng et al., 2026], RollPacker [Gao et al., 2026a], RLinf [Yu et al., 2026], RollArt [Gao et al., 2026b], DynaRL [Wang et al., 2026], and Weave [Wu et al., 2026] shorten RL steps by overlapping rollout with training, repacking long-tail trajectories, placing stages on suitable hardware, or moving resources and jobs between stages and clusters. Their inference engines still receive independent requests, so they compose with Canopy. RhymeRL [He et al., 2026] uses earlier steps for drafts and load balancing. Seer [Qin et al., 2026] uses GRPO groups to predict output lengths and draft speculative tokens, and deliberately splits each group across instances to balance load, keeping KV in a global pool. Canopy instead keeps each group on one instance, computing its shared prefix once without moving KV (Appendix F), and keeps instances within memory by admission rather than splitting.

Tree-structured RL and shared-prefix training. Tree-GRPO [Ji et al., 2026], TreePO [Li et al., 2026b], TreeRL [Hou et al., 2025], ARPO [Dong et al., 2026b], AEPO [Dong et al., 2026a], and AT^{2}PO [Zong et al., 2026] sample rollout trees, which create the groups that Canopy serves. Tree Training [Wang et al., 2025], Prefix Grouper [Liu et al., 2025], AReaL-DTA [Zhang et al., 2026], and our PrefixFlow [Li et al., 2026a] remove the same redundancy from training by computing a group's shared prefix once; Canopy removes it from rollout generation, and the two compose. Concurrent work also reuses shared prefixes [Chen et al., 2026, Xu et al., 2026, Gai et al., 2026].

Prefix-aware serving. vLLM [Kwon et al., 2023] and SGLang [Zheng et al., 2024] cache and reuse prefixes within an instance, and Hydragen [Juravsky et al., 2024] batches the attention over a shared prefix across the requests that share it. Preble [Srivatsa et al., 2025], the SGLang router [The SGLang Team, 2024], and llm-d [llm-d Project, 2025] route requests by prefix match and load; Preble also weighs eviction-induced recomputation and replicates hot prefixes. Mooncake [Qin et al., 2025] chooses between fetching remote KV and recomputing it. These systems know what is cached and where, but not whether or for how long it will be read again, which the RL controller knows.

Agent-aware KV management and scheduling. InferCept [Abhyankar et al., 2024], Continuum [Li et al., 2025], and TokenCake [Bian et al., 2025] keep, offload, or drop a request's KV while it waits for a tool. KVFlow [Pan et al., 2025] orders eviction by an agent step graph from the workflow framework and evicts suffixes before shared prefixes. ThunderAgent [Kang et al., 2026] receives program identities from the client, pauses programs when KV memory is over-committed, and balances memory across instances. These systems decide per request, tool call, or program, and none treats the group as the unit of placement or admission: none is told where a group's shared prefix ends or when the group is done, and none plans memory from the previous step (Table 13 in Appendix H).

I Changes Since the Reviewed Version

(1) We removed results that our logs do not support: Table 2 and the 2.11× result of the submitted paper (its Canopy entries could not be traced to raw logs, and the 2.11× runs processed different numbers of tokens), the 36.7% figure (from an illustrative script), the partial-preemption counters of the submitted Table 3 (they counted tagging calls, not preemptions that kept a prefix), and the same-backend replays reported during the discussion, whose baselines included the start-up of a load-balancer process in the timed window; in those replays Canopy's admission and engine hooks did not run, so we withdraw their numbers (1.36×/1.34× on the Agent-RL replay, 1.87×/1.90× on the depth-10 tree, the 1.25–3.32× group-size sweep, the long-output points, and the mixed-size runs). We also withdraw two ablations reported during the discussion: spill admission on a 406-request trace (1.19×), on which Canopy was slower than vLLM (21.0 against 14.5 s), and the KV-reuse comparison (3.28×), which measured vLLM's own prefix caching rather than Canopy. The training runs show no detectable effect, not an improvement, and the cost model for transfer or recompute that we described was not implemented. (2) We re-evaluated all systems with the method of §4.1: synthetic rollout steps, identical work for every policy, symmetric timing, and three repetitions. (3) Group-level admission (§3.4) is new in this version and was not evaluated by the reviewers. It is not a revision of the submitted paper's transfer-or-recompute admission (its Equation 3 and the extended form we described during the discussion), which decides whether a request moved to another instance fetches or recomputes its group's shared prefix; we did not implement that decision (items 1 and 4). Group-level admission instead decides when a trajectory starts or resumes on its group's instance, using the same group descriptor. We added it because our re-measurement showed that prefix retention and prefix-preserving preemption change rollout time by at most 3.5% and that gains beyond placement need admission; it adds a further 1.3× on tree search (turn) and 1.7–2.0× on local and web search (§4.2). The training runs (§4.8) do not use it. (4) Transfer-or-recompute is supported only by microbenchmarks and moved to Appendix F; the evaluated system never moves a group's requests between instances. (5) The reviewed mechanisms were re-implemented on vLLM 0.20.1 (Appendix A). (6) The check-mark comparison table was replaced by Table 13, and the description of Preble was corrected. (7) The training runs were extended from 100 steps with two seeds to 300 steps with three paired seeds. (8) For this version we added the mixed-tree-size experiment (§4.7, Table 9) and a re-implementation of ThunderAgent's released scheduler (Table 10); both use the method of §4.1 on a second H20 cluster with the same software (Appendix E).

(9) We also added long trajectories that branch late (§4.4, Table 12). While running them we found that our implementation kept a group's b from its first request, which overcharges branches after a fork; the long-trajectory runs charge each request's own b, and the main runs are unchanged (Appendix E).

J Compute Resources

All experiments use NVIDIA H20 GPUs (96 GB). The rollout-replay experiments in §4 ran as 119 jobs of 1–32 GPUs, most with 4 or 8 GPUs and under two hours each, about 660 GPU-hours in total; 11 of them (about 60 GPU-hours) ran on the second cluster of Appendix E. The six 300-step training runs used 16 GPUs each for one to two days (roughly 2,300–3,900 GPU-hours). Development and failed runs used more.

K Broader Impact

Canopy aims to reduce the rollout compute of RL training for LLM agents. It does not change what models learn, so it inherits the risks and benefits of the agents being trained.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and §1 state the mechanisms, the measured loss (§2), and the speedups on replayed synthetic rollout steps, and name where Canopy does not help (§4.7). Appendix I lists the changes since the reviewed version.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: §6 lists them: synthetic workloads matched to small samples, known output lengths in the main runs, previous-step estimates from a second sample, admission not yet in verl and not used in training, re-implemented baselines, one engine and GPU type, synchronous rollout, and the cases where Canopy does not help.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper has no theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: §4.1 and Appendices A–D and G give the method, workload parameters and their sources, baselines and their differences from the original systems, engine settings, admission parameters, and the training setup.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The engine hooks, the rollout driver, the workload generator, and the configurations of all reported experiments are public with instructions at the URL in §1. Workloads are generated from seeds, so no data needs to be downloaded.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: §4.1, §4.8, and Appendices B and G. Static caps were chosen after the fact and are marked as such.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Main results and the admission ablation report the median and range of three runs; memory-pressure runs use two runs for the trees and one for local search, and scaling, robustness, and model points are single runs, as stated. The training comparison reports the paired difference with a 95% confidence interval over three seeds (Figure 9).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: §4.1 and Appendix J.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work is a scheduling system and involves no human subjects or sensitive data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Appendix K.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We release no models or datasets.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use vLLM and verl (Apache-2.0), Qwen3 models (Apache-2.0), and the QA datasets of the training runs (HotpotQA CC BY-SA 4.0, MuSiQue CC BY 4.0, 2WikiMultiHopQA Apache-2.0, Bamboogle MIT), all cited. LongBench v2, FRAMES with a commercial web search API, and SWE-bench Verified were used only to collect small samples of trajectory lengths for calibration, under their public terms; we release no data from them.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.

- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The code at the URL in §1 includes a README, an Apache-2.0 license, and the configurations of all reported experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: No human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are the workload being served, not a component of the method.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.