TREEROLLOUT: RESOURCE-AWARE TREE SAMPLING FOR EFFICIENT AGENTIC LLM POST-TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Generating rollouts is a major cost in group-based reinforcement learning and rejection sampling for large language models. In agentic tasks, each trajectory alternates between GPU decoding and tool execution. The independence of individual trajectories leads to continuously fluctuating demand for concurrent tool execution and GPU decoding, intermittently leaving GPUs or tool executors idle. Scheduling can improve resource utilization, but spare capacity can remain unused when no trajectory is ready to use it. We introduce TreeRollout, a rollout engine that adapts trajectory generation to current GPU and tool bottlenecks by expanding policy-sampled trajectories into a tree. When tools are saturated, it branches after completed tool responses, reusing tool work to supply ready decoding work. When decoding becomes the bottleneck, the engine spawns new branches for tool calls from selected reasoning prefixes, enabling more frequent tool invocation with reduced reasoning overhead. We evaluate TreeRollout on different benchmarks covering math, search, and code generation. Given the same rollout time, TreeRollout yields significantly more trajectories than the baseline, with speedups of up to 1.82× in GRPO and 1.63× in RFT, and the largest gains appearing under severely imbalanced GPU and tool resources. Beyond raw throughput, these additional rollouts also lift the final trained model's task performance.

1 INTRODUCTION

Many modern post-training algorithms for large language models (LLMs) require the model being trained to generate large numbers of trajectories as training data (Ouyang et al., 2022; Qwen Team, 2025; Kimi Team, 2025). For example, group-based reinforcement learning (RL) methods such as GRPO sample multiple responses per prompt and use their relative rewards to update the policy (Shao et al., 2024; Guo et al., 2025; Yu et al., 2025). Rejection-sampling fine-tuning (RFT) also samples multiple candidates per problem, but retains verified solutions for supervised updates (Yuan et al., 2023). In agentic tasks, generating these trajectories, or *rollouts*, requires alternating between GPU decoding and external tool execution (Gou et al., 2024). Dependencies between these stages, together with mismatches between resource demand and capacity, can leave GPUs or tool executors idle, increasing rollout time and making generation a bottleneck in post-training.

As Figure 1 illustrates, demand for GPU decoding and tool execution shifts over time within a rollout batch. When tool executors such as code sandboxes or search services reach their concurrency limit, pending calls queue, and the corresponding trajectories cannot resume decoding until their tool results return. GPUs can then become underutilized if too few other trajectories are ready to decode. Conversely, when decoding is the bottleneck, tool executors can remain idle while waiting for new calls to be decoded. Synchronous training introduces an additional source of idleness: each rollout batch must wait for its last few trajectories before the policy update can begin. Continuous batching and asynchronous tool execution overlap work across trajectories, but do not guarantee that ready work is available for both resources.

Prior work on reducing rollout cost falls into two categories: execution scheduling and candidate generation. Scheduling systems coordinate model and tool stages, allocate external resources to individual actions, or place trajectories using tool-return feedback (Kang et al., 2026; Zhang et al., 2026b; Xiao et al., 2026; Chen et al., 2026a). Candidate-generation methods such as tree sampling, prefix sharing, and early stopping reduce the cost of producing additional candidates (Ji et al., 2026;

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

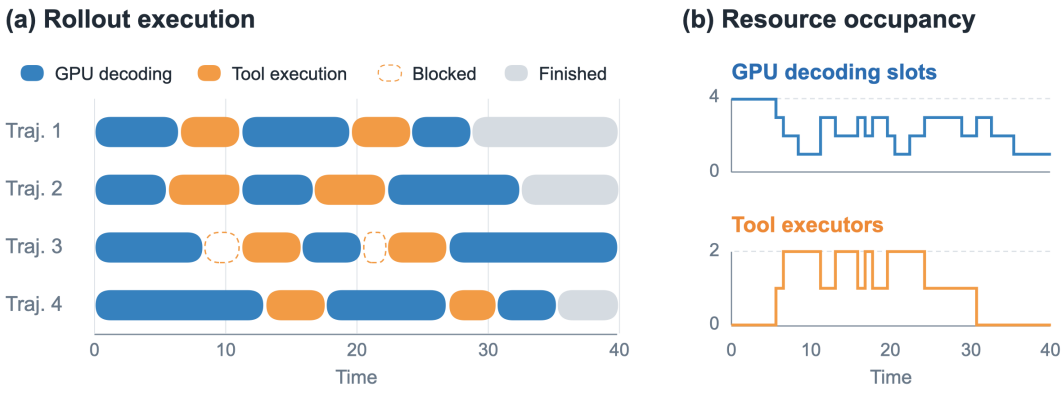


Figure 1: Resource occupancy during a batch of agentic rollouts (schematic). (a) Four trajectories alternate between GPU decoding and tool execution; dashed intervals mark trajectories blocked waiting for a tool executor, and gray intervals run from completion to the batch end. (b) Active decoding slots and tool executors over time, computed from panel (a).

Li et al., 2026; Wang et al., 2026). However, these candidates may still join congested tool queues or require substantial decoding before reaching an idle tool. We therefore consider three factors when generating additional candidates: which resource has spare capacity, how much demand a continuation would add to each resource, and when that demand would arise.

We present **TreeRollout**, a rollout engine that adapts trajectory generation to current GPU and tool bottlenecks by expanding policy-sampled trajectories into trees. For each candidate branch point, the engine estimates the remaining decoding and tool work beyond the shared prefix. It then chooses branch locations dynamically based on these estimates and the current availability of GPU and tool capacity. When a tool pool is saturated and the GPU has spare capacity, branches after completed tool responses reuse the tool results and cached prefixes to supply ready decoding work. When decoding is the bottleneck and tools sit idle, the engine selects reasoning prefixes from which added branches are expected to reach tool execution with little additional decoding. These added branches receive decoding capacity only after eligible primary trajectories have been dispatched. TreeRollout can insert special control tokens into added branches (e.g., `</think>`), guiding them toward tool-call or answer generation according to current resource availability and the remaining rollout budget.

Experiments on math, search, and code generation show that TreeRollout produces additional training candidates and achieves rollout speedups of up to $1.82\times$ in GRPO and $1.63\times$ in RFT relative to linear rollouts, while preserving or improving final task performance.

Our contributions are:

- We show how dependencies between decoding and tool execution can limit throughput in agentic rollouts with independent trajectories.
- We propose **TreeRollout**, which intervenes in trajectory generation through directed branching and proactive token injection, reducing resource idleness and balancing usage under multi-resource constraints.
- Through extensive experiments on math, search, and code generation, we verify that TreeRollout substantially improves rollout throughput and final task performance.

2 Related Work

Post-training with sampled candidates. Group-based RL and RFT both train on candidates generated by the model. GRPO and its variants, including DAPO, compare rewards within a sampled group to update the policy (Shao et al., 2024; Guo et al., 2025; Yu et al., 2025). RFT and methods such as STaR retain verified solutions for supervised updates (Zelikman et al., 2022; Yuan et al., 2023). In agentic tasks, generating these candidates can take a substantial share of training time. TreeRollout accelerates this process across a wide range of settings, improving training throughput.

Tree-based candidate generation. To reduce the sampling cost per trajectory, a line of work generates candidates as trees rather than independent trajectories, sharing prefixes across continuations. Tree-GRPO and TreePO additionally derive training signals from the tree structure, through step-level preferences from outcome rewards and tree-based advantage estimation, respectively (Ji et al., 2026; Li et al., 2026). Other methods guide where to expand with task-level signals, using uncertainty in lookahead rollouts or per-turn task feedback in TSR (Xing et al., 2026; Djuhera et al., 2026). At the execution level, AReaL-DTA reduces the cost of shared prefixes through tree attention (Zhang et al., 2026a). TreeRollout also reuses prefixes to branch, but dynamically plans and guides new branches from the underlying resource level, coupling the sampling distribution with infrastructure conditions for higher throughput.

Decoding and tool scheduling. Infrastructure-side optimization of decoding and tool scheduling reduces idle time while remaining decoupled from the training algorithm. ThunderAgent and Heddle schedule agent workflows across model and tool stages, suspending and migrating trajectories around memory pressure and tool waits (Kang et al., 2026; Zhang et al., 2026b). ARL-Tangram and Libra allocate resources at action or request granularity, using profiled action costs and tool-return states for allocation and placement (Xiao et al., 2026; Chen et al., 2026a). CONCUR and TVCACHE tune concurrency and reuse, adjusting concurrency from cache-congestion feedback and reusing tool results via state-aware snapshots (Chen et al., 2026b; Kumar et al., 2026). TreeRollout instead couples the generation algorithm with underlying resource conditions: while optimizing scheduling, it also actively steers trajectories to fit the current resource state, achieving better performance.

Long-tail reduction and token injection. A few long trajectories can delay completion of a synchronous rollout batch. DARTS uses length statistics to stop over-budget candidates (Wang et al., 2026), and BubbleSpec uses idle workers for speculative rollout drafts (Xu et al., 2026). S-GRPO inserts early-exit instructions into ongoing reasoning (Dai et al., 2025), while DRE edits post-answer thinking in successful trajectories (Wei et al., 2026). TreeRollout injects control tokens according to current GPU and tool availability, steering added branches toward tool calls or answers before reasoning runs long. This both fills in long-tail delays at the end and keeps every resource highly utilized throughout the rollout.

3 METHOD

3.1 PRELIMINARIES

GRPO. For a prompt x, the policy $\pi_{\theta_{\text{old}}}$ samples a group of G completions $\{y_i\}_{i=1}^{G}$ with verifiable rewards $\{r_i\}$. Advantages are group-normalized as $\hat{A}_i = (r_i - \bar{r})/(s_r + \delta)$, where $\bar{r}$ and s_r are the group's reward mean and standard deviation and $\delta > 0$ is a numerical stabilizer. Let $\mathcal{M}_i$ contain the positions of model-generated tokens in the serialized trajectory y_i, and define $L_i = |\mathcal{M}_i|$. The clipped surrogate, written for trajectories with $L_i > 0$, is

$$\mathcal{J}(\theta) = \mathbb{E}_{x,\{y_i\}}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{L_i}\sum_{t\in\mathcal{M}_i}\min\Big(\rho_{i,t}\hat{A}_i, \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_i\Big)\right] \tag{1}$$

where $\rho_{i,t} = \pi_\theta(y_{i,t} \mid x, y_{i,<t})/\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})$ is evaluated only for $t \in \mathcal{M}_i$. In fully synchronous GRPO, each rollout round uses a fixed policy snapshot $\pi_{\theta_{\text{old}}}$, and the update starts after candidate collection and scoring finish. Long-horizon rollouts can require thousands of sequential decoding steps per prompt, making generation a substantial cost even when an update spans multiple minibatches or epochs.

Agentic trajectories. An agentic trajectory interleaves model reasoning, tool-call generation, and external tool responses. Variation in reasoning length, call count, and tool latency produces widely different completion times. The phase markers below illustrate transitions supported by a compatible model and tool-call format. In such a format, inserting `</think><answer>` can guide the model toward a final answer, while inserting `<tool_call>` can guide it toward tool-call generation. The model must still generate the answer or a complete call and its arguments.

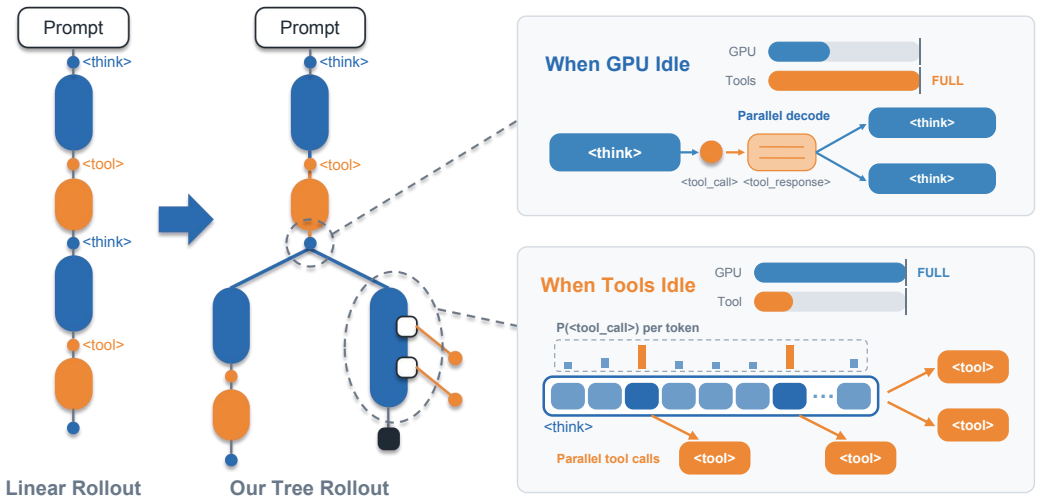


Figure 2: From linear rollout to TreeRollout. Additional candidates reuse cached prefixes and valid tool results. When tools are saturated and the GPU has spare capacity, branching after completed tool responses supplies decoding work. When decoding is the bottleneck and tools have spare capacity, tool-directed branches can reach tool execution with less additional reasoning once they receive decoding capacity under primary-first scheduling.

3.2 Bottleneck-aware branch placement

TreeRollout expands a group of independently sampled trajectories into a tree. The central question of this section is where to place new branches: which existing prefix to extend, and when. The answer depends on which resource is currently the bottleneck, because a branch point determines what the new candidate reuses and how much decoding and tool work it still adds. Algorithm 1 (Appendix A.1) summarizes the execution loop.

Tree initialization and online expansion. For each prompt x, TreeRollout starts G independent policy-sampling requests, forming the primary branches of a prompt-rooted tree $\mathcal{T}(x)$. The primaries alternate between decoding and tool execution while TreeRollout expands the tree from their earliest tool interactions. Each node is a trajectory prefix, and each child extends it with an alternative continuation (Figure 2). Decoding-chunk boundaries and tool returns update the runtime state and trigger reevaluation of tree expansion. Every added continuation is a *filler* branch that preserves its primary and reuses the selected prefix's key–value cache and valid tool results. For stateful tools, reuse requires compatible execution states.

Resource state and continuation estimates. To decide where a branch helps, the engine tracks two kinds of state. First, it monitors the occupancy and queued work of the GPU decoding engine and every tool executor pool, identifying the current bottleneck and resources with spare capacity. Second, it estimates, for every node, the work a continuation started there would still add to each resource. The estimate combines the node's next execution phase, which fixes the immediate resource request, with the mean resource usage of continuations completed below that node, kept separately for decoding and each tool type. Means from the node's subtree and its ancestors are combined with a factor α that attenuates contributions between adjacent nodes (Appendix A.2). Because injected markers can change reasoning length and tool use, these means are heuristics rather than conditional expectations for the controlled continuation. The engine screens candidate branch points against current capacity and remaining expansion budget, favoring those estimated to add the least work to the bottleneck resource.

Branching under tool-executor saturation. When trajectories are waiting on saturated tools and the GPU has spare capacity (Figure 2, top right), TreeRollout places branch points immediately after completed tool responses. A continuation started there shares the response and the cached prefix, so it can begin decoding at once: each such branch turns tool work already paid for into

ready decoding work for the idle GPU. Tool calls issued later by these branches still respect the corresponding executor limits.

Branching under decoding saturation. When decoding work is backlogged and tool executors have spare capacity (Figure 2, bottom right), a new branch is useful only if it reaches a tool call quickly; a branch that first decodes a long reasoning segment would add load to the already scarce resource. The engine therefore seeks earlier reasoning prefixes from which a branch can enter tool-call generation quickly. Let $\mathcal{I}$ contain the eligible positions already observed in a reasoning segment, and let m_{tool} be the tool-call marker from Section 3.1. For the prefix $y_{<\ell}$ before position ℓ, candidate positions are selected as

$$p_\ell = \pi_{\theta_{\text{old}}}(m_{\text{tool}} \mid x, y_{<\ell}), \qquad \mathcal{S}_k = \text{TopK}_{\ell \in \mathcal{I}}\, p_\ell, \qquad k \le |\mathcal{I}| \tag{2}$$

where TopK returns the indices of the k largest probabilities across positions. The engine records p_ℓ while the segment is being generated, so scoring candidates adds no decoding cost. A high p_ℓ indicates the model is likely to begin a call at ℓ, so a branch started there can reach tool execution with little extra reasoning. Naturally reached tool-call boundaries are also eligible. At a chosen prefix, it spawns a filler whose generation starts from the injected marker, and retains the original continuation. The new branch must generate the full call and its arguments before submitting it to the executor. As a filler, it receives decoding capacity only after eligible primaries have been dispatched (Section 3.3); once decoded, prefix reuse and the short path to call generation let it supply work to idle tools with little additional reasoning.

Online branch-point updates. Each branch advances in bounded token chunks while the engine records tool-call probabilities at individual positions. At a chunk boundary, the engine updates candidate positions and reevaluates branching using the latest resource state and workload estimates. Tool returns create new shared-response boundaries, and completed trajectories update the continuation statistics. New branches enter the ready queue described below.

3.3 Scheduling branch execution

Section 3.2 determines where new branches appear; this section determines when they run. Whenever a decoding slot or a tool executor frees up, the scheduler chooses the next ready branch.

Ready work and resource assignment. At any moment, a ready branch waits for exactly one resource, denoted $r(b)$: the GPU if its next step is decoding, or the corresponding tool pool if it holds a complete call awaiting execution. The scheduler keeps all ready branches in a single logical priority queue. A branch is dispatched when its input dependencies are satisfied and its resource has free capacity; a tool response makes the waiting branch ready for decoding again.

Dispatch priorities. Ready branches are ranked by the priority key

$$\mathcal{K}(b) = \big(c_b,\, V_b,\, \widehat{T}_{b,r(b)}\big), \qquad \widehat{\mathbf{T}}_b = \big(\widehat{T}_{b,r}\big)_{r \in \mathcal{R}} \tag{3}$$

where c_b is the primary/filler class, V_b is the subtree value, and $\widehat{\mathbf{T}}_b$ estimates the branch's remaining time on each resource in $\mathcal{R}$. Keys are compared lexicographically: primaries precede fillers, so added candidates never delay training-batch trajectories; higher subtree values come next, directing spare capacity toward promising subtrees; shorter remaining-time estimates on the needed resource break the remaining ties, freeing the resource sooner. The scheduler dispatches eligible primaries first and gives fillers whatever capacity remains, including empty decoding batch slots. They are not free: they consume key–value memory and execution capacity, and may increase decoding-step latency or hold a non-preemptible tool slot.

Value and remaining-time estimates. The value V_b aggregates completed descendants of the branch's current prefix with a task-appropriate statistic, such as mean reward, verification accuracy, or process reward. A branch with no completed descendants inherits the statistic of its ancestors, and before any trajectory has been scored, exploration proceeds without value-based guidance under primary-first dispatch and resource limits alone. Each scored completion updates the nodes along its path.

The components of $\widehat{\mathbf{T}}_b$ derive from the remaining-work estimates of Section 3.2, combined with execution rates and queueing. These per-resource estimates differ from the predicted completion time of the whole continuation, which governs filler admission in Section 3.4.

Chunked execution and redispatch. After a bounded decoding chunk, a branch returns control to the scheduler and keeps its prefix and key–value state, so no branch monopolizes the GPU. Chunk completions, tool returns, and phase transitions update resource availability, branch readiness, workload estimates, and queue priorities, and the scheduler dispatches newly eligible work at once rather than waiting for other branches. Decoding on one branch can therefore overlap with a tool call on another, while every tool call still respects its pool's concurrency limit.

3.4 Bounded rollout termination

Sections 3.2 and 3.3 determine branch placement and execution. This section bounds filler expansion and termination. Primaries retain their baseline sampling and termination rules.

Expansion budgets and filler admission. Candidate and computation budgets limit filler generation. The predicted completion time of remaining primaries sets a target updated as they progress. Adding fillers does not extend it. When forecasts are available, fillers are admitted only if budget remains and they are expected to finish by the target. These forecasts guide admission without guaranteeing on-time completion.

Filler control and rollout termination. At decoding-chunk boundaries, fillers may continue, receive tool-call or answer markers (Section 3.1), or stop. Decisions use resource availability, expected work after a transition, and remaining time. Controlled continuations follow the same scheduling rules. As primary completion approaches, fillers are guided to finish or stopped if predicted to overrun the target. Primaries retain baseline termination rules even if forecasts are exceeded. Collection ends when all primaries have terminated and all fillers are complete or truncated. Truncation does not release an executor occupied by an in-flight tool request. Complete terminal answers, including those after controlled transitions, enter verification, the candidate set, and resource statistics. Truncated trajectories are excluded from both the candidate set and these statistics.

Candidate groups and training. For GRPO, complete primary and filler trajectories are grouped by prompt, giving a group size N_x that can exceed G. Group-normalized advantages and the sample average in Equation 1 use N_x in place of G. Because trajectories are correlated and adaptively selected, retaining primary sampling rules does not preserve the independent-sampling update distribution. Appendix A.3 gives token masks and trajectory weights. Candidate pruning and downsampling can reduce update work under their own selection and weighting rules (Lin et al., 2025; Xu et al., 2025), with retention levels balancing rollout and update time.

4 Experiments

We compare TreeRollout with linear rollouts on five benchmarks under GRPO and RFT, measuring candidate throughput, training time, and task performance. By varying tool concurrency and latency, we examine how the benefits depend on the balance between GPU and tool capacity. Component ablations assess the incremental benefits of each mechanism, while experiments with different candidate-selection rules assess how the generated candidates contribute to training.

4.1 Experimental setup

Tasks and tools. To test TreeRollout's capabilities, we evaluate it across diverse tasks and training paradigms. We use GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) for mathematical reasoning, CodeContests (Li et al., 2022) and LiveCodeBench (Jain et al., 2024) for code generation, and HotpotQA (Yang et al., 2018) for multi-hop question answering. Mathematics and code trajectories call a `code-exec` tool during generation, while HotpotQA trajectories use a `search` tool over a fixed corpus. Each setting combines GPU decoding with one tool executor pool. Both GRPO and RFT are evaluated across these tasks. Further benchmark and tool details are provided in Appendix B.1.

Table 1: Baseline rollouts versus TreeRollout under the same selection rule and training budget. Task performance is reported as percentages using the listed dataset metrics; Init is the initial checkpoint before training. Rollout time is cumulative generation wall-clock time; throughput is reported in thousand tokens per second, with shared prefixes counted once per generation event. Best results between Linear and Ours are shown in **bold**.

Training	Dataset (metric)	Task performance ↑			Rollout time (h)		Throughput (k tok/s)	
		Init	Linear	Ours	Linear	Ours	Linear	Ours
GRPO	GSM8K (Acc.)	87.4	94.5	**94.8**	53.5	**49.7**	2.67	**4.34**
	MATH (Acc.)	89.8	95.8	**96.6**	30.7	**29.4**	3.15	**5.04**
	CodeContests (pass@1)	73.4	79.6	**79.9**	29.8	**29.3**	2.48	**3.28**
	LiveCodeBench (pass@1)	46.5	53.2	**53.8**	46.7	**41.3**	2.39	**3.14**
	HotpotQA (EM)	44.2	48.8	**49.0**	8.9	**8.6**	3.30	**5.96**
RFT	GSM8K (Acc.)	87.4	**93.3**	**93.3**	38.2	**23.5**	3.70	**5.49**
	MATH (Acc.)	89.8	95.2	**95.6**	17.6	**13.9**	3.77	**5.54**
	CodeContests (pass@1)	73.4	78.6	**79.3**	**14.5**	15.1	3.35	**4.06**
	LiveCodeBench (pass@1)	46.5	49.7	**52.2**	30.9	**29.3**	2.85	**3.48**
	HotpotQA (EM)	44.2	48.4	**48.5**	4.8	**4.7**	5.90	**6.51**

Models and baselines. We evaluate Qwen3-4B (Qwen Team, 2025) and Gemma 4 E2B-it (Gemma Team, 2026) under GRPO and RFT. Qwen3-4B is the default model; results are reported for it unless otherwise stated. For each model, we compare TreeRollout with a linear rollout baseline generating independent trajectories. Both use the same inference backend, with continuous batching and asynchronous tool execution. The baseline disables tree expansion, TreeRollout's branch-priority policy, and control-token injection. Further model details and baseline procedures are provided in Appendix B.2.

Execution and training. All experiments run on four NVIDIA RTX 4090 GPUs with 24 GB of memory each. Both the `code-exec` and `search` executor pools have a default concurrency limit of 10. Section 4.3 varies this capacity to study different resource balances. We train for two epochs and sample rollouts at a temperature of 1.0. Each comparison uses the same selection rule for both rollout engines. Further rollout and training settings, including group sizes, generation limits, batch sizes, and optimization parameters, are provided in Appendix B.3.

Metrics. For task performance, we report answer accuracy for mathematics, pass@1 for code, and exact match (EM) for HotpotQA, expressed as percentages. For efficiency, we report cumulative generation wall-clock time and generation throughput in thousand tokens per second. Rollout time covers candidate generation only, excluding verification, selection, and training updates. Shared prefixes are counted once per generation event rather than once per descendant, so reused computation is not billed multiple times.

4.2 Overall results

Table 1 compares the two rollout engines after two training epochs, using all complete candidates for GRPO and all verified-correct candidates for RFT. It reports task performance, cumulative rollout time, and generation throughput for each engine. Relative to linear rollouts, TreeRollout changes rollout time by a 2–12% reduction and generation throughput by a 31–81% increase under GRPO. Under RFT, it reduces rollout time by 12.5% on average and raises generation throughput by 10–48%. Task scores are comparable to or higher than the baseline in all settings.

These results highlight the strength of TreeRollout: it converts otherwise idle GPU and tool capacity into additional completed candidates, delivering more training data per rollout hour, in most cases in less time. The benefit is consistent across all five benchmarks and both training paradigms, and task performance never degrades in any setting. TreeRollout therefore offers a practical way to accelerate existing GRPO and RFT pipelines without altering their training recipes.

Table 2: LiveCodeBench results across tool capacities: final Pass@1, cumulative rollout time, and generation throughput.

Method	Pass@1 (%)	Rollout time (h)	Throughput (k tok/s)
Constrained: 5 concurrent tool calls, slowed			
Baseline	53.2	80.3	1.32
TreePO	53.3	68.4	1.61
AEPO	53.5	64.1	1.95
TreeRollout	53.7	44.1	2.78
Default: 10 concurrent tool calls			
Baseline	53.2	46.6	2.39
TreePO	53.4	41.9	2.63
AEPO	53.6	44.8	2.19
TreeRollout	53.8	41.3	3.14
High capacity: 100 concurrent tool calls			
Baseline	53.1	44.2	2.56
TreePO	53.3	36.5	3.22
AEPO	54.1	49.0	2.01
TreeRollout	54.1	38.8	3.44

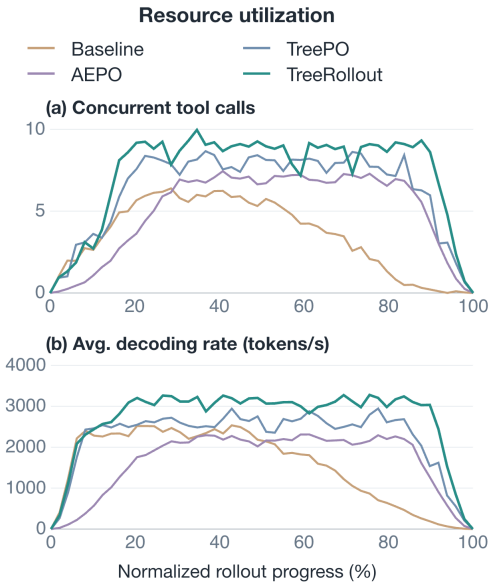


Figure 3: Resource use on LiveCodeBench at the default tool capacity. Elapsed rollout time is normalized to 0–100%.

4.3 Adaptation to tool capacity

The design of TreeRollout rests on a simple premise: spare capacity in whichever resource is not the bottleneck can be converted into additional completions. Whether this premise holds, and how much it is worth, depends on the balance between GPU decoding and tool capacity. We therefore examine how tool capacity affects rollout cost and final test performance under GRPO on LiveCodeBench.

We compare with two samplers that also share prefixes but do not adapt branch locations to runtime resources: TreePO (Li et al., 2026), which redistributes sampling budget from finished to active paths, and AEPO (Dong et al., 2025), which uses entropy to guide branching and sampling budgets. Only sampling changes; candidate selection and GRPO or RFT updates are shared within each comparison. Appendix B.2 gives baseline details.

We test tool concurrency limits of 10 (default), 100, and 5, the last setting also slowing each call to 20 times its normal duration plus 5 s, so that execution moves from tool-limited to decoding-limited conditions while GPU capacity stays fixed. Table 2 reports final-checkpoint Pass@1, cumulative rollout time, and generation throughput. TreeRollout improves over the linear baseline in every setting, with the largest gain under the constrained configuration.

Figure 3 shows concurrent `code-exec` calls and decoding rate over normalized rollout time at the default capacity. TreeRollout leads in throughput at every capacity, and its advantage grows exactly when GPU and tool capacities become unbalanced, showing that the gains come from resource-aware branching rather than tree sampling alone.

4.4 Component ablations

TreeRollout combines tree sampling, resource-aware placement, priority scheduling, phase control, and early stopping. We evaluate the contribution of each component by cumulatively adding them and measuring rollout cost and final model performance under RFT on LiveCodeBench. Starting from linear rollouts, we add tree sampling with prefix reuse and uniform branch-point selection, followed by resource-aware placement, value/time ordering, post-creation phase control, and filler early stopping. Each stage retains all preceding components; the final stage is full TreeRollout.

All stages share the initial model, data, backend, candidate selection, and updates. The initial tree variant samples eligible locations uniformly and orders ready branches by arrival within each re-

Table 3: Cumulative RFT component ablations on LiveCodeBench over two epochs. The last row is full TreeRollout.

Variant	Pass@1 (%)	Rollout (h)	Tokens (k)
Baseline	49.7	30.9	142.2
+ Tree sampling	50.1	28.3	165.0
+ Placement	50.5	26.4	180.8
+ Scheduling	50.9	25.8	184.4
+ Phase control	51.5	31.6	202.7
+ Early stopping	52.2	29.3	200.5

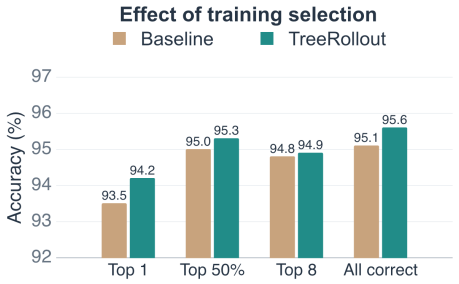


Figure 4: RFT accuracy on MATH under within-prompt selection after correctness filtering. The y-axis starts at 92%.

source and branch class. Adding resource-aware placement changes which locations are selected; adding value/time ordering changes which ready branches execute first.

All tree variants include creation-time markers for tool-directed branches; phase control adds subsequent tool-call and answer markers at decoding-chunk boundaries. The final stage adds early stopping for fillers predicted to overrun the primary completion target. Earlier stages let fillers continue until completion or a hard limit. Truncated branches count toward rollout cost but do not enter the completed training set.

Table 3 reports final Pass@1, cumulative rollout time, and output tokens. Each added component improves task performance over the preceding stage, and the full configuration achieves the best task performance. Every component contributes measurably on top of the previous ones, and the full TreeRollout configuration performs best, confirming that these mechanisms form a coherent design rather than isolated tricks.

4.5 Interaction with training selection

In RL and RFT, different algorithms and tasks call for different rules for selecting training candidates. We therefore examine how TreeRollout performs under different selection rules, comparing final accuracy on MATH with linear rollouts under identical within-prompt rules applied after correctness filtering (Figure 4). Top-1 and Top-8 retain up to one and eight correct candidates per prompt, respectively; Top-50% retains half of the correct pool, and all-correct retains the full pool. Candidates are ranked by the same reward rule for both engines.

TreeRollout shows larger gains with Top-1 selection and all-correct RFT; with 50% retention or per-prompt Top-8, performance is comparable to the baseline or only slightly better. Selection changes both which candidates and how many reach training, so these results alone do not separate candidate quality from training volume. A fixed cap can discard additional correct candidates, while fractional selection can retain different numbers from differently sized pools.

The larger candidate pool built by TreeRollout pays off most at the two ends of selection: when training keeps only the best candidate, and when it keeps every verified candidate. Additional ablation studies are provided in Appendix C.

5 Conclusion

We presented TreeRollout, a rollout engine that uses branch placement and control-token injection to dynamically balance workloads across resources according to their availability during generation. Extensive experiments show that TreeRollout generates more trajectories within the same rollout time while maintaining or improving task performance. We believe that dynamically adapting the generation structure to resource availability during execution offers further opportunities for improving efficiency. Future work will explore more diverse tool environments and more effective methods for intervening in trajectory generation.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, assist with translation, clean and reformat datasets, support qualitative and thematic data analysis, and interpret results. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, or design or provide feedback on research methodology or experiments. Additionally, we used generative AI tools for drafting and editing parts of this paper, and for identifying and summarizing relevant literature. We have reviewed all AI-assisted work: we checked LLM-generated research ideas for potential plagiarism through a manual literature survey, and LLM-generated code was verified and tested for correctness by two authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personal data. The main ethical consideration is the execution of model-generated code: all execution in our experiments runs inside sandboxed subprocesses, while production deployments require stronger isolation. By lowering the rollout cost of post-training, our engine reduces the compute barrier for academic research; we do not foresee uses specific to this work that raise additional ethical concerns beyond those of large language model training in general.

REPRODUCIBILITY STATEMENT

Our engine is implemented on top of the open-source verl/vLLM stack. Algorithmic details are given in Section 3, and all training and rollout hyperparameters are listed in Section 4. All datasets used are public benchmarks (CodeContests, LiveCodeBench, GSM8K, MATH, HotpotQA). We plan to release the code, including the tree agent loop, the scheduler, and the experiment configurations, upon publication.

REFERENCES

Kaiwen Chen, Xin Tan, Jingzong Li, Zhi Zhou, Cen Li, Jiang Liu, Jie Meng, Jiazhi Jiang, and Hong Xu. Libra: Efficient resource management for agentic RL post-training. *arXiv preprint arXiv:2606.03077*, 2026a. URL https://arxiv.org/abs/2606.03077.

Qiaoling Chen, Zhisheng Ye, Tian Tang, et al. CONCUR: High-Throughput Agentic Batch Inference of LLM via Congestion-Based Concurrency Control. In *International Conference on Machine Learning*, 2026b. URL https://icml.cc/virtual/2026/poster/63743.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Muzhi Dai, Chenxu Yang, and Qingyi Si. S-GRPO: Early Exit via Reinforcement Learning in Reasoning Models. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.07686.

Aladin Djuhera, Swanand Ravindra Kadhe, Farhan Ahmed, Syed Zawad, Heiko Ludwig, and Holger Boche. TSR: Trajectory-search rollouts for multi-turn RL of LLM agents. *arXiv preprint arXiv:2602.11767*, 2026. URL https://arxiv.org/abs/2602.11767.

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. Agentic Entropy-Balanced Policy Optimization. *arXiv preprint arXiv:2510.14545*, 2025. URL https://arxiv.org/abs/2510.14545.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026. URL https://arxiv.org/abs/2607.02770.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. Tora: A tool-integrated reasoning agent for mathematical problem solving. In *Proceedings of ICLR*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

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://arxiv.org/abs/2509.21240.

Hao Kang, Ziyang Li, Xinyu Yang, et al. ThunderAgent: A Fast, Simple, and Program-Aware Agentic Inference System. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/62040.

Kimi Team. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

Abhishek Vijaya Kumar, Bhaskar Kataria, Byungsoo Oh, et al. TVCACHE: A Tool-Value Cache for Post-Training LLM Agents. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61148.

Yizhi Li, Qingshui Gu, Zhoufutu Wen, et al. TreePO: Enhancing Policy Efficacy and Inference Efficiency with Tree Modeling. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61677.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, et al. Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814*, 2022.

Zhihang Lin, Mingbao Lin, Yuan Xie, and Rongrong Ji. Cppo: Accelerating the training of group relative policy optimization-based reasoning models. *arXiv preprint arXiv:2503.22342*, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 2022.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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. *arXiv preprint arXiv:2402.03300*, 2024.

Yujie Wang, Siwei Chen, Longzan Luo, et al. DARTS: Distribution-Aware Active Rollout Trajectory Shaping for Accelerating LLM Reinforcement Learning. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61634.

Zihao Wei, Wenjie Shi, Liang Pang, Jingcheng Deng, Shicheng Xu, Shasha Guo, Zenghao Duan, Jiahao Liu, Jingang Wang, Huawei Shen, and Xueqi Cheng. Dynamic Rollout Editing for Reducing Overthinking in RL-Trained Reasoning Models. *arXiv preprint arXiv:2606.17890*, 2026. URL https://arxiv.org/abs/2606.17890.

Bangjun Xiao, Yihao Zhao, Xiangwei Deng, Shihua Yu, Yuxing Xiang, Huaqiu Liu, Qiying Wang, Liang Zhao, Hailin Zhang, Xuanzhe Liu, Xin Jin, and Fuli Luo. ARL-Tangram: Unleash the resource efficiency in agentic reinforcement learning. *arXiv preprint arXiv:2603.13019*, 2026. URL https://arxiv.org/abs/2603.13019.

Shangyu Xing, Siyuan Wang, Chenyuan Yang, et al. Lookahead Tree-Based Rollouts for Enhanced Trajectory-Level Exploration in Reinforcement Learning with Verifiable Rewards. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10011530.

Yixuan Even Xu, Yash Savani, Fei Fang, and J. Zico Kolter. Not all rollouts are useful: Down-sampling rollouts in llm reinforcement learning. *arXiv preprint arXiv:2504.13818*, 2025.

Yuhang Xu, Kaibin Tian, Yang Tian, et al. BubbleSpec: Turning Long-Tail Bubbles into Speculative Rollout Drafts for Synchronous Reinforcement Learning. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/64109.

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. *arXiv preprint arXiv:1809.09600*, 2018.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. In *Proceedings of NeurIPS*, 2022.

Jiarui Zhang, Yuchen Yang, Ran Yan, et al. AReaL-DTA: Dynamic Tree Attention for Efficient Reinforcement Learning of Large Language Models. In *International Conference on Machine Learning*, 2026a. URL https://icml.cc/virtual/2026/poster/62287.

Zili Zhang, Yinmin Zhong, Chengxu Yang, et al. Heddle: A Distributed Orchestration System for Agentic RL Rollout. *arXiv preprint arXiv:2603.28101*, 2026b. URL https://arxiv.org/abs/2603.28101.

A MORE METHOD DETAILS

A.1 ONLINE ROLLOUT ALGORITHM

Algorithm 1 combines the rules of Sections 3.2–3.4. The mutable state S contains the trees $\mathcal{T}$, active primaries $\mathcal{P}$, active fillers $\mathcal{F}$, ready queue $\mathcal{Q}$, complete candidates $\mathcal{C}$, resource occupancy, remaining budgets, and continuation statistics.

Algorithm 1 TreeRollout: event-driven branching and scheduling

Require: Prompts $\mathcal{X}$, fixed policy π, primary count G, capacities K, filler budgets B
Ensure: Complete candidate groups $\{\mathcal{C}_x\}_{x\in\mathcal{X}}$ with verification results

$(\mathcal{T}, \mathcal{P}) \leftarrow \mathrm{StartPrimaries}(\mathcal{X}, G, \pi)$
$\mathcal{F} \leftarrow \emptyset$; $\mathcal{Q} \leftarrow \mathcal{P}$; $\mathcal{C} \leftarrow \emptyset$; $S \leftarrow \mathrm{InitState}(\mathcal{T}, \mathcal{P}, K, B)$; $\tau \leftarrow \bot$
while $\mathcal{P} \cup \mathcal{F} \neq \emptyset$ **do**
  $E \leftarrow \mathrm{PollEvents}(S)$; $S \leftarrow \mathrm{Update}(S, E)$
  $\tau \leftarrow \mathrm{PrimaryTarget}(\mathcal{P}, S; \tau)$
  for all $b \in \mathcal{F} \cap \mathrm{ChunkReady}(E)$ **do**
    $a_b \leftarrow \mathrm{Control}(b, S, \tau) \in \{\mathrm{continue}, \mathrm{tool}, \mathrm{answer}, \mathrm{stop}\}$
    if $a_b = \mathrm{stop}$ **then**
      $\mathrm{Truncate}(b)$; $\mathcal{F} \leftarrow \mathcal{F} \setminus \{b\}$; $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{b\}$
    else
      $\mathrm{ApplyControl}(b, a_b)$
    end if
  end for
  for all $(v, m) \in \mathrm{Propose}(\mathcal{T}, S)$ **do**
    if $\mathcal{P} \neq \emptyset \ \wedge\ \mathrm{Admit}(v, m, S, \tau)$ **then**
      $b \leftarrow \mathrm{Fork}(v, m, \pi)$
      $\mathcal{F} \leftarrow \mathcal{F} \cup \{b\}$; $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{b\}$; $S \leftarrow \mathrm{Register}(S, b)$
    end if
  end for
  while $\mathcal{Q}_{\mathrm{eligible}}(S) \neq \emptyset$ **do**
    $b^* \leftarrow \underset{b\in\mathcal{Q}_{\mathrm{eligible}}(S)}{\arg\min_{\mathrm{lex}}} \left(c_b, -V_b, \widehat{T}_{b,r(b)}\right)$
    $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{b^*\}$; $S \leftarrow \mathrm{Dispatch}(S, b^*, r(b^*))$
  end while
end while
$\mathrm{FinishVerification}(\mathcal{C})$
return $\mathcal{C}_x = \{b \in \mathcal{C} : \mathrm{prompt}(b) = x\}$ for each $x \in \mathcal{X}$

Events and candidate retention. The sets in Algorithm 1 are views of S and change with each state update. STARTPRIMARIES creates G independent requests per prompt, without waiting for complete answers. POLLEVENTS returns available chunk-completion, tool-return, and scoring events without imposing a batch barrier. UPDATE refreshes prefixes, occupancy, readiness, budgets, and statistics, applies baseline termination to primaries, removes finished branches from the active sets, and adds complete terminal trajectories to $\mathcal{C}$. Truncated trajectories enter neither $\mathcal{C}$ nor completion-based resource statistics. Verification results update values when available, and FINISHVERIFICATION waits for any remaining candidate scores before training.

Control, proposals, and admission. PRIMARYTARGET updates the completion target τ from remaining primaries and retains the last target after they finish; $\bot$ denotes an unavailable forecast. For chunk-ready fillers, CONTROL uses resource availability, expected subsequent work, and remaining time to choose an action (Section 3.4). APPLYCONTROL preserves the prefix and either continues decoding or appends the corresponding tool-call or answer marker. PROPOSE returns eligible prefix–marker pairs (v, m) using Section 3.2: completed tool-response boundaries under tool saturation, and the position top-k rule or natural tool-call boundaries under decoding saturation. It uses continuation estimates to favor lower additional demand on the bottleneck; $m = \emptyset$ denotes a continuation without an injected marker. ADMIT checks resource and budget eligibility and, when

completion forecasts are available, requires $t + \widehat{T}^{\text{finish}}_{v,m} \leq \tau$. Here t is the current time and $\widehat{T}^{\text{finish}}_{v,m}$ estimates the candidate's remaining wall-clock time, distinct from a single-resource estimate. FORK reuses the prefix cache and valid tool results without replacing its primary; REGISTER records the new branch and updates admission state before the next check.

Dispatch and stopping semantics. The set $\mathcal{Q}_{\text{eligible}}(S)$ contains ready branches with satisfied dependencies and free capacity at resource $r(b)$. With $c_b = 0$ for primaries and $c_b = 1$ for fillers, lexicographic minimization implements primary-first dispatch, then larger V_b, then shorter $\widehat{T}_{b,r(b)}$ (Section 3.3). Before value observations are available, the value comparison is omitted as in Section 3.3. DISPATCH reserves capacity and launches one bounded decoding chunk or one complete tool request, so eligibility is recomputed after every dispatch. Truncation does not by itself release an executor occupied by an in-flight tool request. The loop stops only when both active sets are empty; primaries always retain baseline termination rules.

A.2 ESTIMATING RESOURCE REQUIREMENTS FROM COMPLETED TRAJECTORIES

The estimator in Section 3.2 uses completed trajectories to estimate the resource work required after a candidate prefix. Let $\mathcal{R}$ contain the decoding resource and the separate tool executor pools. For an edge e in the trajectory tree, let $c_{e,r} \geq 0$ denote its recorded usage of resource $r \in \mathcal{R}$. Usage is measured consistently within each resource; decoding and tool usage are retained as separate quantities. Queueing is reflected in the runtime waiting state used for bottleneck detection.

Resource statistics at each node. For a node v, let $\mathcal{L}(v)$ be the trajectories that pass through v and end with a terminal answer, and let $n_v = |\mathcal{L}(v)|$. This includes trajectories that produce a complete answer after a controlled phase transition. Trajectories terminated before producing a complete answer are excluded, since their observed suffixes omit unfinished work. For a completed trajectory $\ell \in \mathcal{L}(v)$, its resource usage after v is

$$C_{v,\ell,r} = \sum_{e \in \text{path}(v,\ell)} c_{e,r}. \tag{A.1}$$

When $n_v > 0$, the node maintains the empirical mean

$$\mu_{v,r} = \frac{1}{n_v} \sum_{\ell \in \mathcal{L}(v)} C_{v,\ell,r}. \tag{A.2}$$

Thus, a prefix's recorded cost is excluded from the estimate of work introduced by branching there. When a trajectory completes with a terminal answer, its suffix usage contributes to the statistics of the nodes on its path. These means can be maintained with a count and a resource-wise sum at each node. They describe observed completions; when longer trajectories are more likely to remain unfinished or be stopped, the available observations can favor shorter continuations.

Attenuation between neighboring nodes. Node statistics provide local observations of continuation cost. TreeRollout combines nearby node statistics to estimate the requirements of a current branch, attenuating contributions by α as they pass between adjacent nodes. This allows information from completed continuations to inform branch locations with fewer local observations. The decay follows adjacency in the trajectory tree.

Transferring ancestor estimates. For a current node v, let $\mathcal{N}(v)$ contain v itself when $n_v > 0$, together with every ancestor of v that has completed observations. Let $d(u, v)$ be the number of tree edges from an ancestor u to v, with $d(v, v) = 0$. Using this ancestor neighborhood gives a direct path along which each contribution is attenuated once per edge.

The mean at an ancestor u measures work from an earlier prefix. Let $C_{u\to v,r}$ be the observed resource usage along the current path from u to v. Subtract the work already performed on this path to obtain a residual estimate:

$$q_{u\to v,r} = \mu_{u,r} - C_{u\to v,r}. \tag{A.3}$$

For $u = v$, the path cost is zero and $q_{v\to v,r} = \mu_{v,r}$. This transfer provides an approximation when local observations are sparse and relies on comparability between the current continuation and those

observed from u. It can overestimate or underestimate the remaining work when their resource requirements differ.

For each resource r, let $\mathcal{N}_r(v)$ contain the nodes $u \in \mathcal{N}(v)$ for which $q_{u\to v,r} > 0$ or $\mu_{u,r} = C_{u\to v,r} = 0$. The latter case retains an empirical zero when the completed continuations used none of that resource and the current path has used none either. A zero or negative residual caused by work already performed on the current path is unavailable for that resource and is excluded from the aggregation.

For $0 < \alpha \leq 1$ and nonempty $\mathcal{N}_r(v)$, combine the available estimates as

$$\widehat{w}_{v,r} = \frac{\sum_{u\in\mathcal{N}_r(v)} \alpha^{d(u,v)}\, n_u\, q_{u\to v,r}}{\sum_{u\in\mathcal{N}_r(v)} \alpha^{d(u,v)}\, n_u}. \tag{A.4}$$

The node's estimate is the resource vector $\widehat{\mathbf{w}}_v = (\widehat{w}_{v,r})_{r\in\mathcal{R}}$, with one component for the GPU and for each tool executor pool. Each included node's weight is its observation count attenuated once per edge: the local node carries weight n_v, its parent $\alpha\, n_u$, and an ancestor two edges away $\alpha^2 n_u$. These weights balance observation count and proximity; an ancestor with many observations can outweigh a nearby node with few observations. Smaller α gives relatively more weight to nearby observations, while normalization preserves the units of resource usage. The aggregation operates on node means, which may share completed trajectories; its weights are node contributions rather than counts of independent samples.

If $\mathcal{N}_r(v)$ is empty, the requirement for resource r is unknown. The engine then uses the current occupancy, waiting state, and candidate phase to guide branching with respect to that resource, while retaining available estimates for other resources. This rule also covers the initial period before any trajectory has completed.

Use during rollout. Branch placement combines historical workload estimates with the candidate's next execution phase and current resource conditions. The next phase identifies upcoming resource requests: a tool-directed branch first requires decoding to generate its call and arguments, then execution in the relevant tool pool. Historical estimates provide a heuristic for the subsequent workload, whose distribution can change after a controlled phase transition. Injecting a tool-call or answer marker can change reasoning length, tool choice, and call count relative to the completed suffixes used by the estimator. The ancestor-weighted mean is therefore not assumed to be an unbiased estimate for each controlled continuation. An empirical zero therefore does not override a resource request implied by the chosen phase. Current occupancy and waiting states identify bottlenecks, while phase information and workload estimates guide the choice toward candidates that can use spare capacity with limited additional demand on those bottlenecks. For tool-directed branching, this decision is applied to the candidates supplied by the position top-k rule and naturally reached tool-call boundaries in Section 3.2. Completed trajectories update the node statistics, while chunk completions and tool returns update the prefixes and resource conditions used by the scheduler. These updates occur as individual branches advance within a rollout round.

A.3 Candidate groups and training weights

Candidate groups. Let $\mathcal{C}_x$ be the retained complete trajectories for prompt x, with $N_x = |\mathcal{C}_x|$. The basic GRPO update uses this group with G replaced by N_x in Equation 1. Fillers originate from selected prefixes and are subject to resource-dependent expansion and completion rules, so the group is correlated and adaptively selected. Adding their rewards changes the group mean and standard deviation, and can therefore change the advantages of the retained primary trajectories as well.

Token and trajectory weights. The set $\mathcal{M}_i$ contains model-generated reasoning, tool-call arguments, answer tokens, and model-generated format markers. Tool responses and externally injected markers, including those inherited in a shared prefix, remain conditioning context and are excluded from $\mathcal{M}_i$. Under the stated objective, each retained trajectory has outer weight $1/N_x$ and its token

losses are normalized by $L_i = |\mathcal{M}_i|$. Thus, tool-response length affects the context and execution cost but does not enlarge the loss denominator. The same distinction between model-generated targets and external context applies when verified complete trajectories are used for RFT.

Shared prefixes. Caching a shared prefix reduces execution work, while the objective remains an average over retained root-to-leaf trajectories. For a shared model-generated prefix token e, let $\mathcal{D}_e$ contain the retained trajectories that include it. Its contributions are accumulated over those trajectories:

$$\sum_{i \in \mathcal{D}_e} \frac{1}{N_x L_i} \min\Big(\rho_e \hat{A}_i,\ \mathrm{clip}(\rho_e, 1-\epsilon, 1+\epsilon)\hat{A}_i\Big). \tag{A.5}$$

Here ρ_e is the token probability ratio conditioned on the shared prefix; each descendant retains its own advantage $\hat{A}_i$. This trajectory average does not introduce an additional normalization by the number of descendants. Optional candidate pruning or down-sampling can change these weights and follows the selected method's own rules.

Scope of the probability ratio. The per-token ratio compares the current policy with the fixed rollout policy on model-generated tokens conditional on the actual history. It does not by itself correct for the selection of a branch location, the injection of a control marker, or the admission and retention of a filler. Equation 1 therefore describes the surrogate applied to the collected group, without implying equivalence to independent policy sampling. Training applies no additional correction for adaptive tree selection beyond these token ratios.

B Experimental details

B.1 Benchmarks and tools

The training and final evaluation problem sets are disjoint. During evaluation, reference answers and hidden-test feedback are withheld from the model throughout generation.

GSM8K. GSM8K contains grade-school mathematical word problems with numerical answers (Cobbe et al., 2021). We use the `main` configuration of `openai/gsm8k`, retaining its official split of 7,473 training and 1,319 test problems. Correctness is determined by matching the numerical answer extracted after the `####` delimiter.

MATH. MATH contains competition-level mathematics problems with answers expressed in mathematical notation (Hendrycks et al., 2021). We train on the 7,500-problem training split of `DigitalLearningGmbH/MATH-lighteval` and evaluate on the 500 problems in `HuggingFaceH4/MATH-500`. Answer equivalence is checked with `math_verify`.

CodeContests. CodeContests is a competitive programming dataset with standard-input/standard-output problems (Li et al., 2022). We retain non-interactive Codeforces problems with difficulty ratings from 800 to 1,000, yielding 1,017 problems, split into 713 for training and 304 for testing. We report pass@1: a generated solution is correct only if it passes all test cases.

LiveCodeBench. LiveCodeBench is a continuously updated competitive programming benchmark (Jain et al., 2024). We use `release_v6` of `livecodebench/code_generation_lite`, which includes both standard-input and function-based problems with starter code. Although the release is intended for evaluation, we create a custom random split of its 1,055 problems into 739 training and 316 test problems using seed 42. The split named `val` in our files serves as the test set. The test set comprises 97 easy, 114 medium, and 105 hard problems.

HotpotQA. HotpotQA is a multi-hop question-answering benchmark (Yang et al., 2018). We restrict the task to bridge questions and randomly sample 2,000 training and 1,000 test questions. The original training and validation splits contain 72,991 and 5,918 bridge questions, respectively.

Code execution. Mathematics and code trajectories use a local FastAPI service exposed as `code_interpreter`, referred to as `code-exec` in the main text. Each call executes the generated Python code as a separate script, returns standard output, and has a 10-second timeout. The service accepts standard input for sample tests and supports driver templates for function-based problems. It applies AST filtering and resource limits of 4 GB memory, 10 seconds of CPU time, and 32 MB file size, followed by process-group cleanup. Calls use JSON arguments inside `<tool_call>` markers in the Hermes format.

Search. HotpotQA trajectories query a local, CPU-based Okapi BM25 retriever over a fixed corpus of 509,308 passages, obtained by pooling and deduplicating distractor contexts. The corpus includes the gold supporting passages for the evaluated questions. The service supports multiple queries and a configurable retrieval count, returning the top five passages per query by default in a ranked text format. Both tools run locally without external model or API calls, and the retriever uses no neural reranker.

B.2 Models and baselines

Qwen3-4B. Qwen3-4B is a dense language model from Alibaba's Qwen family with switchable thinking and non-thinking modes (Qwen Team, 2025). It supports tool use in both modes.

Gemma 4 E2B-it. Gemma 4 E2B-it is an instruction-tuned multimodal model from Google with configurable thinking and native function calling (Gemma Team, 2026). Our experiments use its text interface.

Shared baseline setup. The linear GRPO and RFT baselines share TreeRollout's initial checkpoint, inference backend, continuous batching, asynchronous tool execution, and candidate-selection rule.

GRPO baseline. For each prompt, the baseline samples independent tool-using trajectories from a fixed policy snapshot and scores their completed answers. It computes advantages within each prompt's candidate group and updates the policy with the GRPO objective (Shao et al., 2024). When reward-based pruning is enabled, candidates are selected within prompts or across the batch before advantages are normalized within each retained prompt group. Policy updates begin after the round's candidate collection and scoring are complete.

RFT baseline. The baseline generates candidates through independent sampling and filters out solutions that fail correctness verification (Yuan et al., 2023). Reward-based selection then operates within each prompt, and the retained trajectories form the dataset for supervised fine-tuning. Under all-correct selection, every verified-correct complete trajectory is retained.

TreePO rollout. TreePO samples trajectories in fixed-length token segments and branches at segment boundaries, reusing the key–value cache of shared prefixes (Li et al., 2026). When a path finishes early, its remaining branching budget is redistributed to paths that are still active, keeping the decoding batch full.

AEPO rollout. AEPO uses entropy pre-monitoring to divide the sampling budget between initial trajectories and later branches (Dong et al., 2025). During generation, entropy changes after tool interactions guide branching, while a penalty for consecutive branching limits repeated expansion along a single path.

Tree-based sampling comparison. Section 4.3 reports test Pass@1 for the final checkpoint of each model trained on LiveCodeBench, using the evaluation protocol in Appendix B.3. Rollout time and output tokens are accumulated over training; evaluation generation is excluded from these costs. We use the sampling mechanisms of TreePO and AEPO with the same candidate-selection and GRPO or RFT update procedure as the other samplers in each comparison. Their original advantage estimators and policy-update objectives are not used.

B.3 Execution and training details

Execution environment. All experiments run on four NVIDIA RTX 4090 GPUs with 24 GB of memory each. Training uses verl 0.7.0, with vLLM 0.11.0 and Transformers 4.57.1 for Qwen3-4B. Gemma 4 E2B-it instead uses vLLM 0.19.0 and Transformers 5.5.0. Both the `code-exec` and `search` executor pools have a default concurrency limit of 10.

Rollout settings. Rollout sampling uses a temperature of 1.0. Both GRPO and RFT start with $n = 16$ trajectories per problem, corresponding to $G = 16$ primary trajectories in TreeRollout. Each generation call is limited to 16,384 output tokens, and the total sequence length is capped at 32,768 tokens. Each trajectory permits at most 10 rounds of tool calls. The maximum tool-response length is 4,096 tokens (`max_tool_response_length`).

TreeRollout settings. Decoding proceeds in chunks of 256 tokens, and the resource estimator uses $\alpha = 0.5$. Further chunk-size ablation settings are provided in Appendix C.1. We set $k = 3$ for tool-call position selection and create at most two filler branches from the same prefix position. For each prompt, at most $5G = 80$ fillers may be added to the initial $G = 16$ primaries, giving at most 96 trajectories in total. New filler admission stops when this limit is reached or 50% of the primaries have terminated, whichever occurs first. This admission limit does not itself terminate existing branches, which remain subject to the rules in Section 3.4.

Rewards and evaluation. For training, GSM8K gives 1 for valid formatting and another 1 for a correct answer in that format; MATH gives 1 for valid formatting and 2 for verified correctness. Code rewards are the test-case pass rate plus a bonus of 1 when all test cases pass. HotpotQA uses a binary answer-matching reward and a step reward of $+0.1$ for retrieving a supporting document. Final evaluation uses the standard exact-match (EM) metric for HotpotQA and strict pass@1 for code, requiring all test cases to pass. Evaluation uses the final training checkpoint, a temperature of 1.0, and the same generation and total sequence-length limits as training. We report mean task scores over five generations per evaluation problem.

Training and selection. All training runs use full-parameter fine-tuning for two epochs. For both GRPO and RFT, we use AdamW with a learning rate of 2×10^{-6} and process eight prompts per batch (`train_batch_size`=8). Table 1 uses all complete candidates for GRPO and all verified-correct complete candidates for RFT, without additional reward-based pruning.

Resource and cost measurements. Rollout time measures candidate generation and excludes verification, candidate selection, and training updates. Total time includes candidate generation, verification, selection, and training updates. Elapsed time is measured directly; overlapping stages must not be summed as if they were sequential. Each experimental configuration is trained once.

C Additional ablation studies

C.1 Decoding chunk size

We examine how the decoding chunk size affects rollout efficiency and scheduling responsiveness. The chunk size bounds how long a branch decodes before returning control, setting the granularity at which the scheduler updates priorities and reacts to tool returns. The reference configuration uses a 256-token chunk, as specified in Appendix B.3. We compare alternative chunk sizes of 64, 128, 512, and 1024 tokens on LiveCodeBench with Qwen3-4B, sharing the model, data, tool configuration, and rollout budgets across settings.

Table 4 reports completed-candidate throughput, cumulative rollout time, and output tokens for each chunk size. Relative to the reference, the alternative sizes reduce throughput by 0–25% and increase rollout time by 0.2–19%, with the 512-token setting matching the reference almost exactly. Smaller chunks let the scheduler react earlier to tool returns and priority changes but interrupt decoding more often, while larger chunks amortize dispatch overhead at the cost of later reactions. These measurements test whether the chunk size mainly trades scheduling responsiveness against dispatch overhead, without changing candidate quality.

Table 4: Decoding chunk-size ablation: completed-candidate throughput, cumulative rollout time, and output tokens. The reference configuration uses 256-token chunks (Appendix B.3).

Chunk size (tokens)	Throughput (cand./s)	Rollout time (s)	Output tokens (M)
64	0.044	587	2.54
128	0.053	539	2.51
256 (reference)	0.059	495	2.48
512	0.059	496	2.47
1024	0.057	508	2.45

Table 5: Per-model comparison of linear rollouts and TreeRollout under the same selection rule and training budget. Task performance is reported as percentages using the listed dataset metrics; Init is the initial checkpoint before training. Rollout time is cumulative generation wall-clock time; throughput in thousand tokens per second.

Model	Dataset (metric)	Task performance ↑			Rollout time (h)		Throughput (k tok/s)	
		Init	Linear	Ours	Linear	Ours	Linear	Ours
Qwen3-4B	GSM8K (Acc.)	87.4	94.5	**94.8**	53.5	**49.7**	2.67	**4.34**
	MATH (Acc.)	89.8	95.8	**96.6**	30.7	**29.4**	3.15	**5.04**
	CodeContests (pass@1)	73.4	79.6	**79.9**	29.8	**29.3**	2.48	**3.28**
	LiveCodeBench (pass@1)	46.5	53.2	**53.8**	46.7	**41.3**	2.39	**3.14**
	HotpotQA (EM)	44.2	48.8	**49.0**	8.9	**8.6**	3.30	**5.96**
Gemma 4 E2B-it	GSM8K (Acc.)	90.5	95.6	**96.2**	39.5	**36.8**	4.21	**6.03**
	MATH (Acc.)	78.8	84.8	**86.0**	23.1	**21.2**	4.58	**6.43**
	CodeContests (pass@1)	71.4	75.3	**76.6**	22.5	**21.1**	3.64	**4.66**
	LiveCodeBench (pass@1)	31.6	37.3	**38.6**	34.6	**31.7**	3.52	**4.49**
	HotpotQA (EM)	43.2	46.8	**47.4**	6.9	**6.8**	4.78	**7.27**

C.2 Experiments across models

We examine whether TreeRollout's efficiency and training benefits persist across the model families introduced in Section 4.1. For each model, TreeRollout is compared with its corresponding linear baseline under the same data, tool configuration, candidate selection, and training budget. The comparison covers all five benchmarks under GRPO. We assess the change relative to each model's own baseline, avoiding differences in model capability or backend versions being read as rollout-engine improvements.

Table 5 reports task performance, cumulative rollout time, and generation throughput for Qwen3-4B and Gemma 4 E2B-it. Relative to each baseline, TreeRollout reduces rollout time by 2–12% and raises generation throughput by 31–81% for Qwen3-4B, and by 1–8% and 28–52% for Gemma 4 E2B-it. Task scores change by $+0.2$ to $+0.8$ and $+0.6$ to $+1.3$ points, respectively. Gains that persist across both families would indicate that the improvements stem from the rollout mechanism rather than from model-specific behavior.

C.3 ALL-BRANCH EXPANSION WITH RANDOM SCHEDULING

We compare Linear, All-branch/random, and full TreeRollout on Qwen3-4B with LiveCodeBench under RFT. All-branch/random proposes forks at every eligible tool-response boundary and tool-directed prefix, without selecting positions by resource state or filtering admission by predicted completion time. It retains prefix caching, valid tool-result reuse, the top-k candidate rule, and the same branch limits: at most two fillers per position and 80 per prompt, with admission stopping after half the primaries terminate. Ready branches are dispatched uniformly at random when their required resource is available, removing primary-first and value/time priorities. Creation-time markers, subsequent phase control, and filler early stopping remain unchanged, so this is a joint ablation of selective admission and dispatch, not of all resource feedback. Candidate selection and the two RFT training epochs follow Appendix B.3.

Table 6: Qwen3-4B LiveCodeBench RFT comparison. Rollout time measures generation only for all configurations. Pass@1 evaluates the final trained checkpoint.

Method	Rollout time (h)	Pass@1 (%)
Linear	30.9	49.7
All-branch/random	43.1	51.9
TreeRollout	29.3	52.2

All-branch/random reaches 51.9% Pass@1, compared with 52.2% for full TreeRollout and 49.7% for Linear.

D LIMITATIONS AND DISCUSSION

Scope of resource constraints. Our experiments study rollouts constrained by two resources, GPU decoding and a single tool-executor pool. The rules in Section 3 are stated over a general resource set, but their empirical validation here is limited to this pair. Settings with three or more constrained resources, such as several tool pools with distinct latencies or a separate verification stage, may change both how often bottlenecks switch and how much continuation estimates are worth. Evaluating such regimes is left to future work.

Stateful tools. Our tool model assumes that calls are independent, so a branch can reuse a completed response or issue a new call without coordinating with the executor. Stateful tools break this assumption: branching from a mid-trajectory point requires the executor to fork its state, or to maintain some persistence or rollback mechanism. Holding forkable states costs time and memory, itself a resource that our current accounting does not model. How branch placement should weigh this cost, and how much of it can be amortized across branches that share a prefix, remain open questions.

Effects of truncation and token injection. Early stopping of fillers and repeated control-token injection shift the completed-trajectory distribution away from the policy's own. Although our trained models still improve, the induced distribution differs from the original, and when this gap helps or hurts deserves further study. An injected control token can also be read as a hint, resembling a user who actively interrupts and redirects an agent mid-episode. Treating injection as hint insertion rather than termination control may then lift both trajectory quality and quantity, which we plan to study.