BRANCHWEAVE: LOCAL WAVEFRONTS WITH DEVICE-WIDE BATCHING FOR VERIFIER-GUIDED TEST-TIME SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time scaling lets compact language models spend additional inference compute on difficult queries, extending reasoning under deployment constraints. Verifier-guided search converts this budget into candidate paths and scores, yet existing runtimes advance subtrees in global depth rounds. A completed subtree waits for unrelated work, while immediate dispatch produces small generator and verifier batches. We introduce BranchWeave, which separates the dependency scope of search from the batching scope of model execution. BranchWeave derives selection domains from reducer dependencies; once a domain receives its selection scores, its local wavefront advances. Stable node identities keep sampling decisions, scores, KV state, and speculative continuations attached to their owners as calls are reordered. Device-wide generator and verifier queues then pool ready calls across domains and depths, forming dense batches without delaying local progress. Together, local release and shared batching convert barrier time into scored reasoning before the response deadline. Across three generator–PRM pairs and five reasoning benchmarks, BranchWeave completes searches up to 5.46× faster and delivers up to 4.94× higher goodput.

1 INTRODUCTION

Serving reasoning models close to users can reduce communication delay and keep context local, while memory limits often favor compact models (Alizadeh et al., 2024; Song et al., 2024; Yu et al., 2024; Chen et al., 2026; Park et al., 2025). Test-time scaling (TTS) improves these models by allocating additional inference compute to difficult queries (Snell et al., 2024; Wu et al., 2024; Muennighoff et al., 2025; Guo et al., 2025). Verifier-guided methods generate candidate paths, score them, and select which continue (Lightman et al., 2024; Chen et al., 2025). Only scored paths completed before the deadline can affect an interactive response. The runtime schedule therefore determines how much of the available compute becomes useful reasoning within the response window. Figure 1 frames the resulting separation: reducer dependencies govern when a domain may advance, while model execution can pool ready calls from many domains.

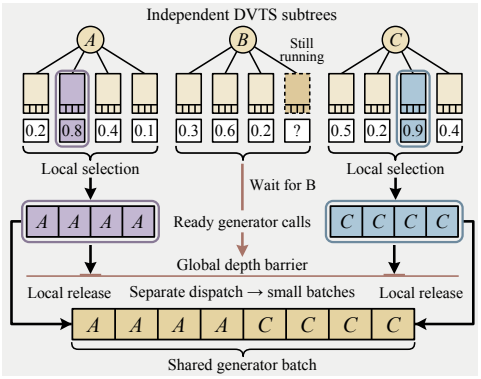


Figure 1: Reducer-local dependencies determine when a domain may advance. Device-wide epochs pool its released generator and verifier calls with ready work from other domains, retaining efficient batches without restoring a global barrier.

However, verifier-guided runtimes commonly tie search progress to global depth rounds. This is required when all candidates compete in one beam, but overly broad for root-parallel methods such as DVTS, whose reducers select within separate subtrees (Beeching et al., 2024; Hooper et al., 2025). Once one subtree has all the scores required for selection, unfinished work elsewhere cannot alter

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.

its choice, yet the current round still blocks it. Serving systems optimize batching, prefix reuse, and KV management (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024; Agrawal et al., 2024; Ye et al., 2025), while FastTTS adds speculation and memory-aware scheduling (Chen et al., 2026), but they can execute only calls exposed by the search controller. Dispatching each ready subtree avoids the barrier at the cost of small generator and verifier batches. Separating these scopes without weakening either reducer decisions or model batches raises three challenges.

The first challenge is how to identify the smallest safe synchronization boundary. Existing controllers use a global depth boundary to preserve frontier comparisons. That boundary matches global beam search but retains dependencies absent from root-parallel selection (Beeching et al., 2024; Hooper et al., 2025). Releasing one completed path is also unsafe because an unfinished sibling may still change the reducer's choice. The safe boundary must follow the selection rule, covering every candidate and mutable decision state read by one reducer while excluding unrelated reducers. A subtree can then advance without changing its comparison, budget, or stopping decision.

The second challenge is how to preserve each path's decisions when local progress changes execution order. Global rounds keep sampling streams, score ties, KV state, and speculative continuations aligned with fixed batch and depth positions (Chen et al., 2026; Leviathan et al., 2023; Miao et al., 2024). Local wavefronts instead allow calls from different subtrees and depths to be rebatched and completed out of order. If randomness or saved state follows position or completion order, the same search may sample different tokens, resolve ties differently, or resume the wrong continuation. Each call and its state must therefore remain bound to a stable logical node identity through release, rebatching, and reduction.

The third challenge is how to batch locally released work without recreating the global barrier. Continuous batching improves efficiency by waiting for compatible requests (Yu et al., 2022; Agrawal et al., 2024; Du et al., 2025). A newly completed subtree often releases only one small sibling group, so immediate dispatch underfills generator decoding and verifier prefill. Waiting for many subtrees to align recovers batch size but again makes the earliest work wait for the slowest subtree. These calls also feed later reducer decisions, unlike independent serving requests. Batching should span subtrees and depths while preserving group membership and result ownership, respecting KV capacity, and bounding how long ready work waits.

Therefore, we introduce BranchWeave to separate search progress from model service. Reducer dependencies define selection domains, each of which advances as a local wavefront. Stable logical identities keep sampling, KV state, and speculative futures attached to nodes as calls move between batches. Shared generator and verifier queues combine calls across domains and depths and return results to their owners. Local release and shared batching turn barrier waits into scored reasoning before the response deadline.

The main contributions are as follows:

- We derive selection domains from reducer read and write dependencies and advance each closed domain as a local wavefront. This gives search progress its smallest safe synchronization boundary and characterizes the barrier time exposed as stragglers shift across successive search depths.
- We make locally released work reorderable through logical node identity. Node-bound sampling, canonical reduction, and ownership of KV and speculative state preserve each path's decisions as its calls move across physical batches and depths.
- We build separate generator and verifier queues that form device-wide epochs from complete sibling groups across domains and depths. Owner-aware routing and bounded admission recover dense batches while independent reducers continue to advance.
- We implement and evaluate BranchWeave on AIME, AMC, and MATH-500. It completes searches up to 5.46× faster and delivers up to 4.94× higher goodput across the evaluated workloads.

2 Background and Motivation

2.1 Selection and Model Execution

Verifier-guided search alternates generation, scoring, and selection: the generator extends paths, a process reward model (PRM) scores them, and a reducer chooses which continue (Lightman et al.,

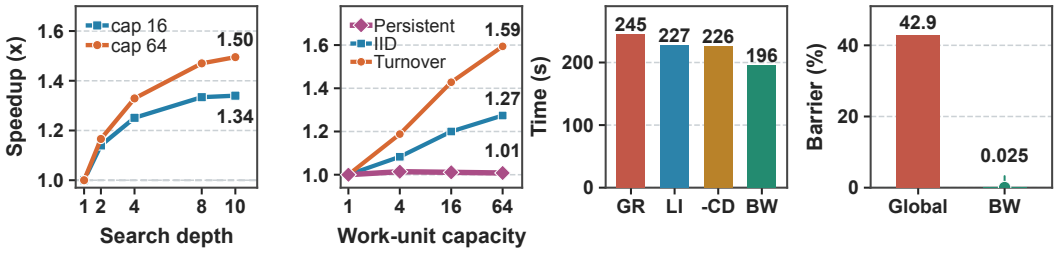


Figure 2: Repeated-barrier and shared-batching evidence. Left: simulated speedup versus depth; cap. denotes work-unit capacity. Center-left: one persistent straggler, independently sampled (IID) stragglers, and depth turnover. Right panels: fixed-work model time and barrier share. GR denotes GlobalRound, LI LocalImmediate, and -CD disables cross-domain batching.

2024; Zhang et al., 2025b; She et al., 2025). The reducer determines the synchronization scope. Global beam search ranks the full frontier, whereas DVTS ranks each root-parallel subtree separately (Beeching et al., 2024); a completed DVTS reducer therefore has no selection dependency on unfinished subtrees.

Model execution benefits from a broader scope. Generator steps share model weights across active sequences, and the PRM scores completed prefixes in prefill batches (Yu et al., 2022; Agrawal et al., 2024; Chen et al., 2026). Requests need compatible shapes and available memory, not a common reducer or depth, so pooling them across subtrees improves device use (Kwon et al., 2023; Zheng et al., 2024; Ye et al., 2025). A depth-synchronous runtime hides ready local work until the global round closes; immediate per-reducer dispatch exposes it but fragments model batches. Readiness should therefore follow the reducer while batching spans the device.

2.2 Changing stragglers compound global waits

At one depth, a global round waits for the slowest subtree; across depths, it pays a sequence of these maxima. A local wavefront instead follows each subtree's causal path. If the same subtree remains slowest, both schedules follow essentially the same critical path. When the straggler changes, the global schedule accumulates delays from several paths even though no individual path contains all of them or incurs the same cumulative delay.

Reasoning traces exhibit this changing-straggler pattern. Step lengths evolve with the generated paths (Kim et al., 2026), and the slowest domain changes across 90.4% of adjacent depths in the recorded traces. The first two panels of Figure 2 isolate the consequence by giving GlobalRound and local wavefronts identical work while varying depth and the identity of the slow domain.

As depth grows from one to ten, local-wavefront speedup rises from 1.00× to 1.34× and 1.50× at the two capacities. A persistent slow domain stays near 1.01× because local release cannot bypass the true critical path, while changing the slow domain across depths raises the gain to 1.59×. The contrast identifies repeated waits, rather than one unusually long branch, as the opportunity for local progress across successive reducer rounds in practice.

2.3 Local release needs shared batching

Local release exposes work sooner, but a closed subtree usually contributes one small sibling group. Immediate service shortens queueing while weakening weight reuse and PRM batch formation (Yu et al., 2022; Agrawal et al., 2024); waiting for many subtrees to align recreates the barrier. Ready work must therefore be pooled without making readiness global.

The two right panels hold generated tokens, scored nodes, and dependencies fixed. GlobalRound takes 245 seconds, with 42.9% spent behind barriers. LocalImmediate and no cross-domain pooling reach 227 and 226 seconds because their batches remain fragmented. Pooling independent domains cuts time to 196 seconds and barrier share to 0.025%. Local progress therefore exposes the opportunity, while cross-domain pooling turns staggered releases into dense model work. This motivates BranchWeave's two scopes: reducer-local progress exposes work, and device-wide batching serves it efficiently under the same device and memory limits.

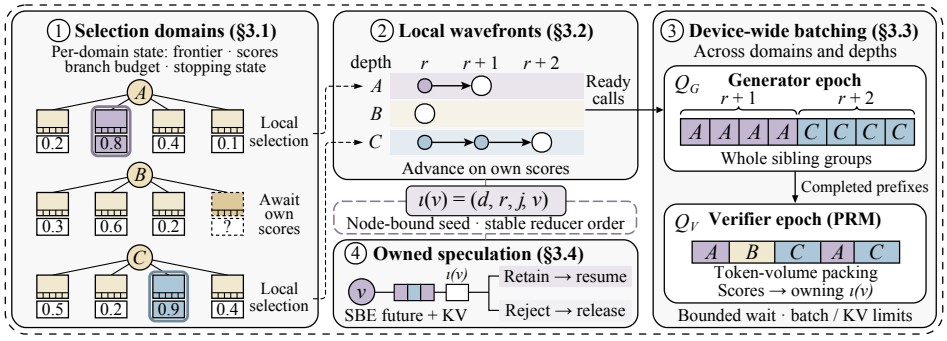


Figure 3: BranchWeave separates reducer-local selection from device-wide batching; logical node identity carries state across reordered execution.

3 METHOD

Figure 3 shows how BranchWeave connects search progress to model service. A selection domain bounds the state required by one reducer; once it closes, a local wavefront can advance independently. Stable logical identity keeps sampling, scores, KV state, and speculation attached to each node as physical execution is reordered. Device-wide queues then batch ready calls across domains and depths and return every result to its owner. Safe release exposes work, identity makes it reorderable, and shared epochs serve it efficiently.

3.1 REDUCER-LOCAL SELECTION DOMAINS

The unit of local progress is the candidate group and mutable state read by one reducer, which we call a selection domain. At depth r, domain d holds frontier F_d^r and transitions by expanding, scoring, and reducing it:

$$C_d^r = \mathrm{Expand}(F_d^r; \Xi_d^r), \tag{1}$$

$$Z_d^r = \mathrm{Score}(C_d^r), \tag{2}$$

$$F_d^{r+1} = \mathrm{Reduce}_d(C_d^r, Z_d^r). \tag{3}$$

where C_d^r is the expanded candidate set, Z_d^r contains its verifier scores, and Ξ_d^r contains the expansion's random choices. The boundary is determined by mutable state, not by model-call layout. For reducer actions a and b, let R_a and W_a denote their read and write sets. They conflict when

$$a \bowtie b \iff W_a \cap (R_b \cup W_b) \neq \varnothing \ \vee \ W_b \cap (R_a \cup W_a) \neq \varnothing. \tag{4}$$

The connected components of this relation are the finest valid domains. A shared ranking, budget, or stopping state joins candidate groups; immutable model weights and query inputs do not. Transitions from distinct domains then commute:

$$T_d(T_e(S)) = T_e(T_d(S)), \qquad d \neq e. \tag{5}$$

where S is the logical search state and T_d advances domain d. DVTS assigns one domain per root-parallel subtree; global beam search keeps one for the full frontier. Children inherit the parent domain, whose reducer releases the next frontier after all candidate scores return.

3.2 LOCAL WAVEFRONTS AND LOGICAL IDENTITY

Once a domain has the scores needed for selection, it releases the next frontier even if other domains remain at an earlier depth. This forms a local wavefront: each domain preserves its causal order, while independent domains need no shared depth barrier. The reducer still compares its original group.

If $\tau_{d,r}$ is the time for domain d to complete its selection inputs at depth r, the dependency paths of global rounds and local progress are

$$T_{\mathrm{round}} = \sum_r \max_d \tau_{d,r}, \qquad T_{\mathrm{local}} = \max_d \sum_r \tau_{d,r}. \tag{6}$$

Global rounds accumulate the slowest domain at every depth, whereas local progress waits for the longest domain path. Their difference is the waiting available to overlap before shared-device service is considered.

Release order cannot identify a search node because legal interleavings change which call completes first. BranchWeave binds node v and its sampling stream to

$$\iota(v) = (d, r, j, \nu), \qquad \xi_v = H(s, \iota(v)), \tag{7}$$

where d is its domain, r its depth, j its child slot, and ν separates repeated expansions; H derives node randomness ξ_v from search seed s. Canonical identity order resolves score ties, so completion order changes neither the node's choices nor its reducer position.

Each queued call carries its logical identity, domain, and model role. When the call finishes, the runtime writes tokens or scores to the named node and updates only that domain's pending set. Reducer eligibility therefore depends on returned logical records, while physical batches may mix calls from different domains and depths.

3.3 Device-wide batching

Let $Q_{s,d}(t)$ contain calls for model $s \in \{G, V\}$ released by domain d. Device-wide queues and an admitted epoch $A_s(t)$ satisfy

$$Q_s(t) = \bigcup_d Q_{s,d}(t), \qquad A_s(t) \subseteq Q_s(t), \quad |A_s(t)| \le B_s, \quad \sum_{a \in A_s(t)} \rho_s(a) \le C_s, \tag{8}$$

where G, V denote the generator and PRM, B_s is the batch limit, $\rho_s(a)$ is the footprint of call a, and C_s is the corresponding capacity. Generator epochs preserve complete sibling groups; verifier epochs pack prefixes by token volume. Pooling spans domains and depths because ownership, rather than depth, routes each result.

The two models form epochs from separate queues. Admission waits only while a fuller batch can repay the delay to ready calls; a waiting bound prevents that choice from becoming another global barrier.

3.4 Speculation follows logical ownership

BranchWeave retains FastTTS's speculative beam extension and prefix-aware execution. A speculative future and its KV blocks are recorded under the identity of the node they extend, even when later epochs finish the work. When the reducer keeps that node, generation resumes from the compatible saved state; when it chooses another candidate, the future is canceled and its blocks are released. Epoch membership may change, but ownership lasts until this reducer decision. The generator, PRM, and search objective remain unchanged.

4 Experiments

4.1 Experimental setup

Platform, models, and workloads. All systems use an NVIDIA RTX 4090 D, vLLM 0.9.2 (Kwon et al., 2023), and PyTorch 2.7 under the same memory budget and serving backend; NVML records power. We pair Qwen2.5-Math-Instruct generators (Qwen et al., 2025) with Skywork 1.5B or Math-Shepherd 7B PRMs (Skywork Team, 2024; Wang et al., 2023a), with 7B+1.5B as the primary pair. Workloads comprise AIME 2024–2025, AMC 2024–2025, and MATH-500 (Hendrycks et al., 2021); the main comparisons use DVTS.

Baselines. vLLM is the general serving baseline (Kwon et al., 2023), and FastTTS is the state-of-the-art specialized test-time-scaling baseline (Chen et al., 2026).

Metrics. E2E latency adds generator and PRM time; wall time also includes controller and host work. Goodput measures effective completed tokens per beam per model second, while Precise Goodput accounts for their completion times (Chen et al., 2026). Energy-to-solution integrates device power. Paired ratios use comparison/BranchWeave for latency and energy and BranchWeave/comparison for goodput.

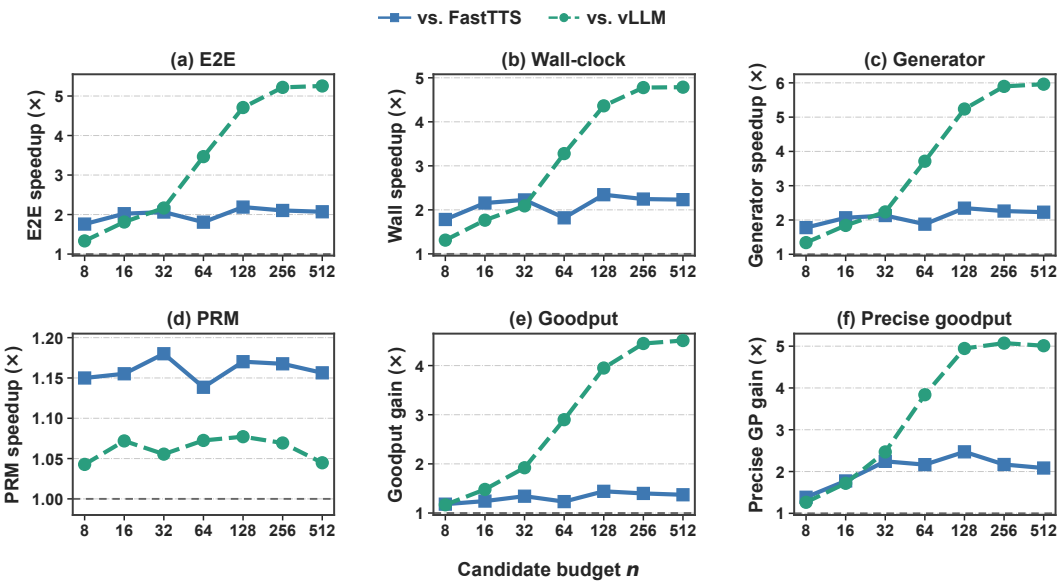


Figure 4: Latency and goodput gains across search budgets, relative to FastTTS and vLLM.

4.2 Scaling with the search budget

Wider searches expose more independent selections, but global rounds turn that parallelism into repeated cross-domain stalls. In Figure 4, the E2E speedup grows from 1.76× to 2.07× over FastTTS as n increases from 8 to 512, while the corresponding gap to vLLM grows from 1.33× to 5.25×.

The stage curves explain this separation. At $n = 512$, generation is 2.23× faster than FastTTS and 5.96× faster than vLLM, whereas PRM speedup is only 1.16× and 1.04×, respectively. A reducer can therefore expose another decoding step while an unrelated sibling group is still being scored; shared epochs absorb those early releases without changing the reducer boundary. Table 1 shows the same pattern in absolute time: on AIME 2025, generator time falls from 254.0 to 108.6 s, while PRM time changes from 11.7 to 10.0 s and PGP rises from 8.13 to 16.89 tok/s.

Wall time and energy track the reduction in generator time across workloads, showing that released work remains efficiently batched without shifting cost into host-side coordination. The goodput gains likewise persist after normalizing by completed search work. At one quarter of FastTTS's

Table 1: Performance by workload. GP and PGP denote Goodput and Precise Goodput. The gain columns use paired ratios.

Dataset	System	E2E (s)	**E2E speedup**	Wall (s)	Gen. (s)	PRM (s)	GP (tok/s)	**GP gain**	PGP (tok/s)	Energy (kJ/q)
AIME 2024	vLLM	483.3	**3.37**×	483.9	473.9	9.4	3.69	**2.80**×	5.51	158.2
	FastTTS	276.8	**2.17**×	298.1	264.0	11.3	7.61	**1.33**×	8.21	99.1
	BranchWeave	125.5	**1.00**×	132.1	114.5	9.2	9.90	**1.00**×	16.70	42.7
AIME 2025	vLLM	453.1	**3.40**×	453.7	442.6	10.5	3.72	**2.85**×	5.23	146.5
	FastTTS	267.4	**2.24**×	278.1	254.0	11.7	7.22	**1.43**×	8.13	91.6
	BranchWeave	120.4	**1.00**×	127.0	108.6	10.0	10.31	**1.00**×	16.89	40.8
AMC 2024	vLLM	359.6	**2.87**×	360.2	351.7	7.9	4.50	**2.41**×	6.34	116.5
	FastTTS	218.1	**2.02**×	234.9	207.6	9.1	7.63	**1.37**×	8.30	77.3
	BranchWeave	103.3	**1.00**×	108.7	94.4	7.4	10.27	**1.00**×	17.48	35.0
AMC 2025	vLLM	329.8	**2.56**×	330.3	321.7	8.1	4.98	**2.31**×	6.95	108.8
	FastTTS	174.4	**1.62**×	197.3	164.5	8.4	9.59	**1.14**×	9.73	64.5
	BranchWeave	100.3	**1.00**×	105.9	91.2	7.5	10.67	**1.00**×	17.67	33.8

Table 2: Stage-level gains and search work by generator+PRM pair. GP and PGP denote Goodput and Precise Goodput; Cand. denotes completed candidates.

G+PRM	Baseline	Latency speedup				Goodput gain		Search work	
		E2E	Wall	Gen.	PRM	GP	PGP	Cand.	Gen. tokens
1.5B+1.5B	FastTTS	1.357×	1.217×	1.539×	0.858×	1.251×	1.192×	1.134×	0.862×
	vLLM	1.287×	1.125×	1.334×	**1.151**×	1.207×	1.114×	1.134×	0.878×
7B+1.5B	FastTTS	1.710×	1.713×	1.787×	1.046×	1.241×	1.740×	0.909×	0.896×
	vLLM	**2.352**×	**2.206**×	**2.467**×	1.141×	**2.027**×	**2.373**×	**1.160**×	**0.799**×
1.5B+7B	FastTTS	0.977×	0.957×	1.562×	0.699×	0.860×	0.890×	1.116×	0.803×
	vLLM	1.130×	1.089×	1.408×	0.983×	0.994×	1.041×	1.116×	0.801×

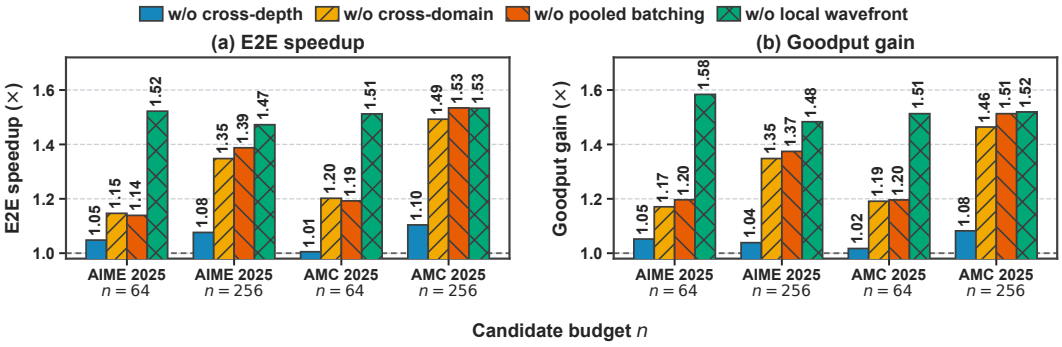


Figure 5: Performance loss after removing each BranchWeave scheduling mechanism.

completion time, BranchWeave has already scored 6.96× as many final-list candidates (Table 9); the fixed-work comparison in Figure 2 isolates the scheduling effect.

4.3 Generator–PRM balance

Table 2 moves the bottleneck between generation and scoring and reports search work as BranchWeave/comparison ratios. With a 7B generator and 1.5B PRM, decoding dominates the model path and exposes the most staggered work. Its 1.787× generator speedup over FastTTS carries through to 1.710× E2E and 1.740× PGP. For the balanced 1.5B+1.5B pair, a 1.539× generator gain becomes a 1.357× E2E gain. When the PRM is 7B, scoring becomes the critical path: each reducer must await its complete sibling group, so less latency can move across the selection boundary. The work columns show that these timing changes are not explained by a larger explored path set.

4.4 Mechanism contribution

GlobalRound restores the depth barrier, while LocalImmediate releases domains without device-wide pooling. Across the four settings in Figure 5, removing the local wavefront raises E2E time to 1.47–1.53× of the full system and reduces Goodput by as much as 1.58×. Removing cross-domain pooling reaches 1.49× E2E, whereas removing only cross-depth admission stays below 1.10×. The ordering shows that local release exposes the main opportunity, shared batching realizes it, and cross-depth admission uses the remaining capacity. The cross-domain loss also grows with the search budget: from 1.15× to 1.35× on AIME 2025 and from 1.20× to 1.49× on AMC 2025.

4.5 Search outcomes under reordered execution

The fixed-choice comparison changes only model-call order. At both reported budgets, local release and GlobalRound produce the same candidate lists and final answers (30/30 in Table 3); physical generation remains at 0.990–0.991× of the control.

Table 3: Search outcomes and PRM scores under reordered execution.

n	Search outcome		PRM trace			Work control	
	Candidate list	Final answer	Exact trace	Mean $\|\Delta\|$ (%)	Max $\|\Delta\|$ (%)	Logical work	Physical generation
16	30/30	30/30	25/30	0.0176	1.002	30/30	0.991×
64	30/30	30/30	26/30	0.0190	0.780	30/30	0.990×

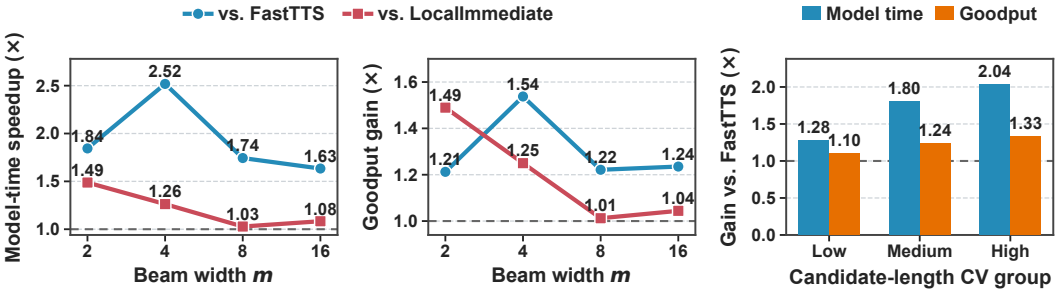


Figure 6: Local beam width and candidate-length coefficient of variation (CV).

Batch shape changes a few PRM traces, with mean score differences of only 0.0176–0.0190%, while the candidate lists and final answers remain stable. Logical identity returns every score to the same reducer position, so reordered model calls do not reorder the alternatives seen by selection. Near-unit physical-generation ratios show that the schedule changes when the work runs rather than substituting additional generation for lower latency.

4.6 Operating conditions

Local progress is most valuable when independent decisions become ready at different times but collectively fill a batch. Across local beam widths, BranchWeave remains faster than FastTTS; at $m = 4$ it reaches a 2.518× model-time and 1.538× Goodput gain (Figure 6).

Uneven candidate lengths stagger readiness and expose intervals that global rounds spend waiting: the high-CV group reaches 2.040× model-time and 1.327× Goodput gain, compared with 1.283× and 1.101× at low CV. Beam width trades release frequency against group size; $m = 4$ balances both. Table 4 varies the admission thresholds.

Goodput remains within 0.995–1.001× of default across these settings, while model time improves by up to 1.093×.

Table 4 shows the effect across admission thresholds. Gen. 20 reduces generator epochs to 0.867× and gives the largest model-time gain, 1.093×, with GP at 1.001× the default; Gen. 12 creates 1.259× as many epochs and retains 0.995× GP. Moving the PRM threshold shifts mean batch volume between 0.871× and 1.038× while model-time gains stay at 1.067–1.073×. Admission changes packing while local releases remain available.

With one reducer, no domain can advance around another, so local and global release expose nearly the same calls. Table 5 accordingly differs by only 1.002× in model time, GP, and PGP, with the generated work unchanged. This near-equality rules out per-request overhead as the source of the

Table 4: Batch-formation sensitivity; absolute value (relative change).

Setting	Model time (s)	GP (tok/s)	PGP (tok/s)	Generator epochs	Mean PRM batch (K tokens)
Default	140.71 (1.000×)	8.43 (1.000×)	15.55 (1.000×)	255.4 (1.000×)	6.64 (1.000×)
Gen. 12	134.76 (1.052×)	8.38 (0.995×)	15.00 (0.959×)	322.5 (**1.259**×)	6.12 (0.924×)
Gen. 20	130.08 (**1.093**×)	8.41 (1.001×)	14.72 (**0.944**×)	224.1 (0.867×)	6.54 (0.988×)
PRM 3072	132.06 (1.073×)	8.42 (1.000×)	15.18 (0.972×)	265.3 (1.032×)	5.75 (**0.871**×)
PRM 5120	132.36 (1.067×)	8.33 (**0.992**×)	14.77 (0.947×)	253.3 (0.985×)	6.88 (1.038×)

Table 5: Single-domain search. GR denotes GlobalRound; other abbreviations follow Table 1.

System	Model time (s)	GP (tok/s)	PGP (tok/s)	Generated tokens (K)	Candidates
GR	19.480 (1.000×)	44.239 (1.000×)	43.859 (1.000×)	3.905	4.000
BW	19.433 (1.002×)	44.344 (1.002×)	43.947 (1.002×)	3.905	4.000

wider gains. They appear when staggered releases from independent domains can be pooled, as in Figure 6, where released groups jointly fill the device.

5 Related Work

Test-time reasoning and verification. Sampling and tree search compare alternative reasoning paths (Wang et al., 2023b; Yao et al., 2023; Beeching et al., 2024); outcome and process supervision rank them (Cobbe et al., 2021; Uesato et al., 2022; Lightman et al., 2024; Wang et al., 2023a; Zhang et al., 2025b; She et al., 2025). Methods vary compute allocation, search policy, and verification granularity (Snell et al., 2024; Wu et al., 2024; Muennighoff et al., 2025; Liu et al., 2025a; Chen et al., 2025; Hooper et al., 2025; Zhang et al., 2025a). These methods allocate compute but generally hold independent subtrees behind a shared depth round; BranchWeave retains their ranking rules while releasing calls at reducer closure.

Serving and model state. LLM serving systems improve efficiency through continuous batching, request multiplexing, structured-program scheduling, optimized attention execution, and paged or virtual KV allocation (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024; Agrawal et al., 2024; Ye et al., 2025; Prabhu et al., 2025; Li et al., 2023). Phase-aware architectures separate or specialize prefill and decoding, while memory-constrained systems place model weights and KV state across limited devices (Zhong et al., 2024; Patel et al., 2024; Du et al., 2025; Qin et al., 2026; Sheng et al., 2023; Alizadeh et al., 2024; Song et al., 2024; Park et al., 2025). A complementary line of work reduces KV overhead through eviction, quantization, compression, and layer-aware retention (Zhang et al., 2023; Liu et al., 2024b; Hooper et al., 2024; Cai et al., 2025). Prefix sharing, cache transfer, and state restoration further avoid repeated computation across requests (Juravsky et al., 2024; Liu et al., 2024a; Yu et al., 2025; Gao et al., 2025; Liu et al., 2025c). These techniques optimize admitted model calls; BranchWeave exposes additional ready calls to them.

Speculation and dependent work. Speculative decoding reduces autoregressive latency by drafting tokens or branches and verifying them in parallel with the target model (Leviathan et al., 2023; Cai et al., 2024; Miao et al., 2024; Li et al., 2024; 2025). The proposals may come from a separate draft model, multiple prediction heads, tree-structured candidates, or learned intermediate features; FastTTS adapts speculation and prefix reuse to verifier-guided search (Chen et al., 2026). Related runtimes support interruption and resumption, fast model loading, dynamic rebatching, and scheduling for variable-length reasoning traces (Abhyankar et al., 2024; Fu et al., 2024; Liu et al., 2025b; Kim et al., 2026). These systems accelerate proposed or admitted work; BranchWeave changes when independent work becomes ready.

6 Conclusion

Test-time scaling improves compact-model reasoning with more inference compute, but global depth rounds delay completed subtrees and expose small batches. Verifier-guided methods generate, score, and select candidate paths, so only completed work can affect a deadline-bound response. BranchWeave separates reducer-local selection from device-wide execution: local wavefronts advance selection domains, stable node identities preserve reordered decisions, and shared generator and verifier queues batch ready calls across domains and depths. Across three generator–PRM pairs and five reasoning benchmarks, BranchWeave completes searches up to 5.46× faster and delivers up to 4.94× higher goodput while reducing wall time and energy. Aligning synchronization with true reducer dependencies converts barrier delay into scored reasoning before the deadline while retaining efficient shared-device service. This makes additional inference compute more useful on constrained devices operating under tight memory and response-time limits.

AI USE STATEMENT

Generative AI tools assisted with code generation and review, mathematical derivations and proof development, searches for related work, and suggestions to improve the clarity of the manuscript. The authors reviewed and verified all AI-assisted material and take responsibility for the final paper, code, and results.

ETHICS STATEMENT

This work studies LLM serving infrastructure using publicly available model checkpoints and benchmarks. It involves no human participants or private user data.

REPRODUCIBILITY STATEMENT

Code for reproducing the reported results is available in the anonymous repository.

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*, Proceedings of Machine Learning Research, 2024.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 2024.

Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2024.

Edward Beeching, Lewis Tunstall, and Sasha Rush. Scaling test-time compute with open models. Hugging Face, https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute, 2024.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2024.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2025.

Hao Mark Chen, Guanxi Lu, Yasuyuki Okoshi, Zhiwen Mo, Masato Motomura, and Hongxiang Fan. Rethinking optimal verification granularity for compute-efficient test-time scaling. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2025.

Hao Mark Chen, Zhiwen Mo, Guanxi Lu, Shuang Liang, Lingxiao Ma, Wayne Luk, and Hongxiang Fan. FastTTS: Accelerating test-time scaling for edge LLM reasoning. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 2026.

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.

Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. PrefillOnly: An inference engine for prefill-only workloads in large language model applications. In *Proceedings of the ACM*

SIGOPS 31st Symposium on Operating Systems Principles. Association for Computing Machinery, 2025.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 2024.

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in LLM serving with HCache. In *Proceedings of the Twentieth European Conference on Computer Systems*. Association for Computing Machinery, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 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. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2021.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2024.

Coleman Hooper, Sehoon Kim, Suhong Moon, Kerem Dilmen, Monishwaran Maheswaran, Nicholas Lee, Michael W. Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. ETS: Efficient tree search for inference-time scaling. *arXiv preprint arXiv:2502.13575*, 2025.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-Throughput LLM inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying AI agents and test-time scaling from an AI infrastructure perspective. In *2026 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*. Association for Computing Machinery, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2025.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

Runze Liu, Junqi Gao, Jian Zhao, Kaiyan Zhang, Xiu Li, Biqing Qi, Wanli Ouyang, and Bowen Zhou. Can 1B LLM surpass 405B LLM? rethinking compute-optimal test-time scaling. *arXiv preprint arXiv:2502.06703*, 2025a.

Xuting Liu, Daniel Alexander, Siva Kesava Reddy Kakarla, Behnaz Arzani, and Vincent Liu. Dynamic rebatching for efficient early-exit inference with DREX. *arXiv preprint arXiv:2512.15705*, 2025b.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. Association for Computing Machinery, 2024a.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025c.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2024b.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Association for Computing Machinery, 2024.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2025.

Jinwoo Park, Seunggeun Cho, and Dongsu Han. SpecEdge: Scalable edge-assisted serving framework for interactive LLMs. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2025.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vAttention: Dynamic memory management for serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. Association for Computing Machinery, 2025.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-centric disaggregated architecture for LLM serving. *ACM Transactions on Storage*, 2026.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

Shuaijie She, Junxiao Liu, Yifeng Liu, Jiajun Chen, Xin Huang, and Shujian Huang. R-PRM: Reasoning-driven process reward modeling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2025.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2023.

Skywork Team. Skywork-o1-Open-PRM-Qwen-2.5-1.5B. https://huggingface.co/Skywork/Skywork-o1-Open-PRM-Qwen-2.5-1.5B, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2024.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*, 2023a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023b.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, 2025.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022.

Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*. Association for Computing Machinery, 2025.

Zhongzhi Yu, Zheng Wang, Yuhan Li, Ruijie Gao, Xiaoya Zhou, Sreenidhi Reddy Bommu, Yang (Katie) Zhao, and Yingyan (Celine) Lin. Edge-LLM: Enabling efficient large language model adaptation on edge devices via unified compression and adaptive layer voting. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. Association for Computing Machinery, 2024.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, and Chen Ma. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*, 2025a.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*. Association for Computational Linguistics, 2025b.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang "Atlas" Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023.

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*. Curran Associates, Inc., 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 2024.

A Selection Independence and Barrier Exposure

Reducer-local progress relies on disjoint mutable search state and choices bound to logical nodes. This appendix derives the resulting domain partition, shows why legal interleavings preserve each reducer's state, and quantifies the barrier time exposed by local progress. It then connects those properties to device-wide epoch formation and the observed stage-level gains.

A.1 Selection-domain factorization

Let the complete logical search state be

$$S = (S_1, \ldots, S_D, S_{\text{shared}}), \tag{9}$$

where S_d contains the frontier, completed scores, logical depth, budget, and reducer state of domain d. S_{shared} contains immutable model and query inputs. A ready domain transition T_d reads S_d and S_{shared}, writes only S_d, and consumes random variables fixed by the identities of nodes created in that transition.

The factorization follows the reducer's read and write sets. A ranking, normalization, budget, or stopping decision shared by two candidate groups places them in the same domain. When their mutable components are disjoint, their transitions can be exchanged.

This boundary can be expressed through the state accessed by a selection decision. For reducer actions a and b, write R_a and W_a for their mutable read and write sets. They cannot advance independently when

$$W_a \cap (R_b \cup W_b) \neq \varnothing \quad \text{or} \quad W_b \cap (R_a \cup W_a) \neq \varnothing. \tag{10}$$

Taking the transitive closure of this relation groups decisions that share mutable selection state. Immutable model weights and the common query may be read by every action without creating a selection dependency. A global stopping budget, in contrast, connects the reducers that update it. The resulting components describe the smallest independent progress regions under the specified search policy; the model calls that implement them may still share a device batch.

More formally, let $\mathcal{A}$ be the reducer actions and define the undirected selection-conflict graph

$$\mathcal{G}_{\text{sel}} = (\mathcal{A}, E), \qquad \{a, b\} \in E \iff \begin{array}{l} W_a \cap (R_b \cup W_b) \neq \varnothing \text{ or} \\ W_b \cap (R_a \cup W_a) \neq \varnothing. \end{array} \tag{11}$$

The connected components of $\mathcal{G}_{\text{sel}}$ are the finest valid selection-domain partition. Endpoints of every conflict edge must remain in one domain; transitivity therefore places each connected component together. Actions in different components have no conflicting accesses and can be separated. Given the action access sets and their conflict edges, a union-find pass constructs the partition in $O(|\mathcal{A}| + |E|\alpha(|\mathcal{A}|))$ time. This analysis occurs when a search policy is compiled, not on every model call.

For DVTS, candidate slots are partitioned among root-parallel subtrees. Each subtree's ranking and stopping state refers to its own slots, so its reducer actions form a separate component. A beam reducer ranks candidates across the entire frontier; its read set connects those candidate groups into one component. This explains the different progress scopes without changing either search rule.

Independent transitions commute. For ready domains $d \neq e$, let T_d neither read nor write S_e, let T_e neither read nor write S_d, and let both transitions read the same immutable S_{shared}. Then

$$T_d(T_e(S)) = T_e(T_d(S)). \tag{12}$$

T_d changes only S_d, so applying T_e first leaves every input to T_d unchanged. The same argument holds with d and e reversed. Both compositions therefore produce the same updated components S_d and S_e, preserve S_{shared}, and leave every other domain untouched.

Legal interleavings converge. Suppose each domain preserves its own transition order and every ready transition eventually executes. The per-domain orders define a partial order. Any two fair legal interleavings are linear extensions of that partial order and can be transformed into one another by swapping adjacent incomparable transitions. Such a pair belongs to different domains and commutes by Equation 12. Repeated swaps preserve state, so both interleavings reach the same final logical state.

The exchange argument also applies to intermediate frontiers. After any prefix of a legal interleaving, the state of domain d depends only on the transitions already executed in that domain. Its next reduction can run when its own candidate set is complete, even if another domain has not reached the same depth. For DVTS, each subtree supplies one such state component; a global beam reduction remains a single component.

A.2 Node-bound randomness and stable reduction

Let s be the search seed and $\iota(v)$ the identity in Equation 7. Sampling for node v uses

$$\xi_v = H(s, \iota(v)), \tag{13}$$

where H is a deterministic hash-based seed derivation. Inserting, delaying, or reordering an unrelated model call cannot consume or shift ξ_v.

For a completed candidate set C with scores z_u, let

$$z^* = \max_{u \in C} z_u, \qquad T_\epsilon = \{u \in C : z^* - z_u \le \epsilon_{\text{tie}}\}. \tag{14}$$

The stable reducer selects the candidate in T_ϵ with the smallest canonical logical identity. Other deterministic reducer operations consume candidates in the same canonical order.

The node identity also anchors the path prefix used by generation. If $p(v)$ is the parent of v, the logical prefix of v consists of the prefix of $p(v)$ followed by that parent's selected continuation. Its KV blocks may be placed or reused differently, but the association between prefix, sampling stream, and node identity is unchanged. A speculative future is attached to the same identity until the reducer chooses or discards its owner.

Choice is independent of completion order. For a fixed logical candidate set C, scores $\{z_u : u \in C\}$, search seed s, and tie threshold ϵ_{tie}, neither node seeds nor the reducer output depends on submission or completion order. Equation 13 depends only on s and $\iota(v)$, so the seed of every node is invariant to other calls. Equation 14 depends only on the completed score set. The canonical order has a unique minimum within that set, making the reducer output independent of the order in which its members arrived.

By induction over each domain's depth, a complete reducer input with fixed node choices and scores produces the same next frontier under every legal interleaving. The base frontier is identical; the induction step uses the same child identities, sampling streams, candidate order, and reducer input. Submission order therefore changes only when a model call executes. Physical batch shapes can produce small differences in PRM scores; Table 3 reports the observed search outcomes separately from score-level equality.

A.3 Repeated-barrier exposure

Consider an idealized search with D domains and R logical depths. Let $\tau_{d,r} \ge 0$ be the time domain d needs to produce the complete reducer input at depth r. If every depth closes with a global round, the logical critical path is

$$T_{\text{round}} = \sum_{r=1}^{R} \max_d \tau_{d,r}. \tag{15}$$

If domains advance as soon as their local reducers close, it is

$$T_{\text{wave}} = \max_d \sum_{r=1}^{R} \tau_{d,r}. \tag{16}$$

Barrier exposure is nonnegative. The difference

$$\Delta_{\text{barrier}} = T_{\text{round}} - T_{\text{wave}} \ge 0. \tag{17}$$

For each domain d, $\sum_r \tau_{d,r} \le \sum_r \max_e \tau_{e,r} = T_{\text{round}}$. Taking the maximum over d proves $T_{\text{wave}} \le T_{\text{round}}$.

For each depth, let $M_r = \arg\max_d \tau_{d,r}$ be the set of slowest domains. Equality holds exactly when

$$\bigcap_{r=1}^{R} M_r \neq \varnothing. \tag{18}$$

If a domain belongs to every M_r, its path sums all per-depth maxima. In the other direction, choose a longest domain path d^*. Equality makes the sum of the nonnegative gaps $\max_d \tau_{d,r} - \tau_{d^*,r}$ zero, so d^* must belong to every M_r. Changing stragglers makes a strict gain possible precisely when no domain remains slowest throughout, although the size of that gain still depends on the service-time gaps. These equations describe the synchronization opportunity; device-wide epochs determine how the released work is served.

The gap can be assigned to individual depths. Choose a domain d^* whose total path time attains T_{wave}. Then

$$\Delta_{\text{barrier}} = \sum_{r=1}^{R} \left(\max_d \tau_{d,r} - \tau_{d^*,r} \right). \tag{19}$$

Each term is the time that the longest local path loses to another domain's step at that depth. A domain slowest at every depth makes the gap zero; changing stragglers can make several terms positive. The size of these service-time gaps, rather than the turnover count alone, determines the available overlap.

The idealized ratio has a simple range when $T_{\text{wave}} > 0$. Summing all domain work upper-bounds the round schedule, and no individual domain has more work than the longest domain path. Likewise, every per-depth maximum is bounded by that longest path. Thus

$$1 \leq \frac{T_{\text{round}}}{T_{\text{wave}}} \leq \min(D, R). \tag{20}$$

Equality on the left holds precisely when some domain attains a maximum at every depth, allowing ties. With only one depth, there is no later step to release around a straggler. With several depths, the intersection of the sets of slowest domains determines whether the global barrier adds to the critical path.

The upper bound is attainable. Let $m = \min(D, R)$ and choose m domain–depth pairs with distinct domains and depths. Assign unit service time to those pairs and zero to all others. Global rounds serialize the m unit steps, so $T_{\text{round}} = m$, whereas each domain path contains at most one unit step and $T_{\text{wave}} = 1$. Thus the ratio is m. The construction is an extreme form of straggler turnover; real traces interpolate between this case and the shared-straggler equality condition in Equation 18.

Consider two domains and two depths with service times (a, b) and (b, a), where $a > b > 0$. Both domains perform $a + b$ work, but their slow steps occur at different depths. Equations 15 and 16 give

$$T_{\text{round}} = 2a, \qquad T_{\text{wave}} = a + b, \qquad \frac{T_{\text{round}}}{T_{\text{wave}}} = \frac{2a}{a + b}. \tag{21}$$

The same domain kept slow at both depths would give equality instead. This contrast is the mechanism isolated by the persistent-straggler and turnover curves in Figure 2.

A.4 Local release and shared service

Selection order and service order impose different constraints. Let $Q_{G,d}(t)$ and $Q_{V,d}(t)$ contain generator and PRM calls made ready by domain d at time t. The model queues collect their unions,

$$Q_s(t) = \bigcup_{d=1}^{D} Q_{s,d}(t), \qquad s \in \{G, V\}. \tag{22}$$

Each call retains its logical owner. An epoch can therefore group calls from different domains without adding a dependency between their reducers: the model returns each result to the owner that released it. Conversely, a domain can finish reduction and release new calls without waiting for an epoch to contain every domain at that depth.

Pooling has value when the model serves compatible calls together more efficiently than in separate launches. Let $c_s(A)$ be the measured service time for a feasible batch A on model s. For two ready groups A and B, their potential service saving is

$$\delta_s(A, B) = c_s(A) + c_s(B) - c_s(A \cup B). \tag{23}$$

Algorithm 1 Local release and device-wide epoch formation.

```
Initialize each domain with its frontier, logical-node map, and pending-score set
Set Q_G ← ∅ and Q_V ← ∅
while an unfinished domain or queued call remains do
    Find the domains whose pending-score sets are complete
    for all such domains d, in canonical node order do
        Apply Reduce_d
        if d continues then
            Enqueue its complete sibling group in Q_G
        else
            Emit its result
        end if
    end for
    Choose Q_s by age and service rule; admit a feasible A_s(t)
    Execute Model_s(A_s(t)) and return each record to its logical owner
    Return unfinished generation to Q_G and completed candidates to Q_V
    Update the owner's pending set with returned verifier scores
end while
```

A domain is eligible again only after its next reducer has all required scores.

The saving can come from shared weight traffic, fuller decode steps, or fewer small verifier prefills. Waiting for B also delays A, so an epoch should not pool indefinitely merely because $\delta_s(A, B) > 0$. The scheduler admits compatible ready work within a bounded wait and the available KV capacity. These two constraints preserve the opportunity from local release while allowing the shared device to recover useful batches. The fixed-work replay and batching ablations measure the combined effect beyond the idealized critical path of Equation 17.

The timing tradeoff is clearest for two groups whose arrivals are separated by Δ. Suppose A is ready at time zero, B becomes ready before a separate service of A would finish, and neither group depends on the other. Serving them separately completes both at $c_s(A) + c_s(B)$; waiting to combine them completes both at $\Delta + c_s(A \cup B)$. In this case, pooling improves the later completion time exactly when

$$\delta_s(A, B) > \Delta. \tag{24}$$

The calculation explains the purpose of a bounded batching window. An additional ready group is useful when its shared-service saving exceeds the delay needed to collect it. Once the earlier group can finish before the later one arrives, holding it provides no such overlap. Generator and PRM epochs have different service functions, so they may favor different group sizes and waiting intervals even though both read from the same domain wavefront.

More generally, the separate and pooled completion times for both groups are $\max\{c_s(A), \Delta\}+c_s(B)$ and $\Delta + c_s(A \cup B)$, respectively. Pooling improves the later completion time when

$$\delta_s(A, B) > \min\{\Delta, c_s(A)\}. \tag{25}$$

If A would finish before B arrives, delaying A also worsens its own completion time. An epoch must therefore weigh shared-service savings against the age of already ready work. This is the reason for bounded waiting alongside capacity-aware admission, rather than an epoch size target alone.

A.5 Wavefront release and epoch formation

The preceding conditions can be expressed as one control loop. A domain closes only after the complete candidate group needed by its reducer has arrived; model service then draws from the union of calls released by all closed domains. For model $s \in \{G, V\}$, an admitted epoch $A_s(t)$ obeys

$$A_s(t) \subseteq Q_s(t), \qquad |A_s(t)| \le B_s, \qquad \sum_{a \in A_s(t)} \rho_s(a) \le C_s, \tag{26}$$

where B_s is the batch limit, $\rho_s(a)$ is a call's relevant resource footprint, and C_s is the corresponding capacity. For generation this footprint includes KV state; for verification it reflects prefill volume. The choice among feasible epochs uses the service saving in Equation 23 together with the age of the oldest ready call. Algorithm 1 makes the ownership and release boundary explicit.

The loop has three invariants. First, a reducer observes either its complete candidate group or no new frontier, so cross-domain batching never changes a selection boundary. Second, every queued

record has one logical owner and is returned to that owner, so physical batch order cannot move state between domains. Third, an epoch may mix depths only after readiness has been established locally; depth remains a service attribute and creates no selection dependency. With D domains and Q queued records, a direct pass over ready state costs $O(D+Q)$ before model execution; ordered queues replace the scan with logarithmic insertion and removal. These invariants are the operational form of the commutation and pooling arguments above.

A.6 Stage balance and useful progress

Local release changes model latency through both generator and PRM service. Let a comparison system spend L_G^0 in generation and L_V^0 in scoring, and let their respective stage speedups be s_G and s_V. Write $p = L_G^0/(L_G^0 + L_V^0)$ for the generator share. The resulting E2E speedup is

$$S_{\text{E2E}} = \frac{L_G^0 + L_V^0}{L_G^0/s_G + L_V^0/s_V} = \frac{1}{p/s_G + (1-p)/s_V}. \tag{27}$$

Let $h = p/s_G + (1-p)/s_V$. The elasticities of E2E speedup to the two stage speedups are

$$\frac{\partial \log S_{\text{E2E}}}{\partial \log s_G} = \frac{p/s_G}{h}, \qquad \frac{\partial \log S_{\text{E2E}}}{\partial \log s_V} = \frac{(1-p)/s_V}{h}. \tag{28}$$

The two terms sum to one and equal the generator and verifier shares of the new model time. This weighted harmonic form explains why a long generator phase offers more room for local wavefronts to affect E2E latency. The width-scaling experiment has a large generator share, and its stage breakdown reflects that leverage. When scoring accounts for more of the model path, the same generator improvement contributes less to the total. The model-pair comparison in Table 2 varies this balance while leaving the progress rule unchanged.

Goodput also depends on useful completed output. For systems A and B on the same question, with effective tokens per beam u_A, u_B and model latencies L_A, L_B, its ratio factors as

$$\frac{G_B}{G_A} = \frac{u_B}{u_A}\frac{L_A}{L_B}. \tag{29}$$

Natural searches can produce different completed paths as stochastic continuations interact with the execution schedule. Their goodput gain therefore combines faster service with the effective progress actually completed. Fixed-work replay holds that progress constant, reducing Equation 29 to the latency ratio in Equation 32. The search-work columns of Table 2 provide context for the natural-search ratios.

B Runtime Behavior and Supporting Experiments

The local wavefront and the model epochs have different notions of readiness. This appendix describes their interaction and then examines the search-order, service-time, and fixed-work behavior behind Figure 2.

B.1 Domain lifecycle

A domain owns its current frontier and the score set required to choose the next one. Once a complete sibling group can be expanded, it enters the shared generator queue. Completed paths enter the verifier queue; their scores return to the owning domain. Only when that domain's candidate set is complete does its reducer release another frontier. Other domains can pass through this cycle independently, including while one waits for a long reasoning step.

The key event is the closure of a local candidate set. A fast path alone does not release a new frontier if its siblings are still being scored, because the reducer must compare the whole group. Once the last required score arrives, however, no other domain's score is an input to that reducer. Its next generation group can enter the shared queue while a slower domain continues to finish an earlier depth. This preserves the sibling comparison that gives DVTS its diversity while removing a wait between unrelated comparisons.

Generator admission keeps sibling groups intact so that reduction sees the candidate multiplicity specified by the search. A depth-rotating queue gives eligible groups access to the device without

imposing a common depth on their selection decisions. The physical order of model calls can vary; the logical owner of each score and continuation does not.

The two queues differ in what makes a useful batch. Generator decoding benefits from several active continuations sharing weight loads across steps; PRM service groups completed prefixes into prefill work. A subtree that has just reduced may therefore contribute generator work to one epoch and return to the verifier queue later, alongside prefixes from domains at other depths. Epoch membership is temporary, whereas domain membership lasts for the search path's lifetime.

B.2 Epoch formation and bounded waiting

Let Q_G and Q_V be the ready generator and verifier queues. Generator items carry sibling-group size and estimated KV footprint; verifier items carry prefill-token count and enqueue time. An epoch draws feasible work from one model's queue within its batch and memory limits. The scheduler favors the active model while it can fill a useful epoch, but an older waiting item or an empty feasible generator queue can trigger verifier service. This balances weight reuse against progress for a small domain.

Model-stationary service avoids switching between generator and verifier for every newly completed path. The waiting bound prevents an already ready domain from remaining behind repeated larger arrivals. Feasibility is checked against the model's active batch and KV footprint before admitting a group; the owner of every admitted call remains available when its result returns. These decisions affect execution order and batch shape, not which candidates the local reducer compares.

The ablation restricts which ready calls may join an epoch. Without cross-domain pooling, an epoch draws from one selection domain. This variant retains the same local reductions and node ownership, allowing the experiment to attribute the change to batch formation.

Cross-domain pooling is especially valuable after local release: a newly ready group may be too small to occupy the generator by itself, while groups released by other domains are already waiting.

B.3 Speculative futures and KV ownership

Speculative beam extension may continue a completed step while its siblings are still running. BranchWeave records that future under the completed node's logical identity and can retain its KV state until selection. A selected node resumes its compatible future or enters ordinary generator admission; a rejected node cancels the future and releases its KV blocks. Thus a continuation created in one device epoch remains with its path even when a later epoch selects the frontier. Canceled speculation remains physical work in the measurements.

The future is provisional until its domain's reducer closes. Its owner may finish earlier than a sibling, but the future cannot become a different subtree's continuation merely because that sibling's model call completes later. When the reducer selects the owner, the saved prefix supplies the continuation; when it selects another node, the tentative state is discarded. If KV pressure requires recomputation, that cost belongs to the execution of the same logical path. This is why logical identity is used for both reducer inputs and cached state, including state outside the final candidate list.

B.4 Local progress under controlled work

Selection under reordered execution. We compare two search structures with the same node-bound token and score choices. The DVTS compiler assigns each root-parallel subtree a disjoint set of candidate slots; global beam search retains one reducer over its full frontier. Replaying DVTS with different legal orders of ready work leaves the generated records and depth-wise selections

Table 6: Simulated GlobalRound/local-wavefront speedup as service-time variability increases.

Service-time CV	Capacity 4	Capacity 16	Capacity 64
0.00	0.988×	0.986×	1.000×
0.25	1.109×	1.239×	1.331×
0.50	1.116×	1.256×	1.357×
1.00	1.127×	1.284×	1.384×
1.50	1.136×	1.300×	1.388×

Table 7: Trace-calibrated local-wavefront simulation.

Capacity	Speedup	Global-round utilization	Wavefront utilization
1	1.000×	1.000	1.000
4	1.139×	0.829	0.933
16	1.489×	0.434	0.618
64	1.717×	0.244	0.414

Table 8: Fixed-work model execution by search budget. Arrows show GlobalRound to BranchWeave.

Search budget	Model-time speedup	Active generator requests	PRM batch tokens
$n = 64$	1.322×	19.82 → 29.99	5,604 → 5,960
$n = 256$	1.182×	39.75 → 51.14	6,278 → 6,361

unchanged. The result connects the state factorization in Appendix A.1 to the search policies used by the runtime: a subtree may advance after its own selection, while a globally ranked frontier stays together.

Service-time variation and repeated waiting. An event simulation gives global rounds and local wavefronts identical work at each search position. The depth and slow-domain comparisons appear in Figure 2; Table 6 varies the dispersion of that work. As service times become less uniform, more of a global round is spent waiting for the current straggler. A local wavefront places ready work from other domains in that interval.

At uniform service time, the schedules are nearly identical. The gain then increases with variation and available concurrent capacity. To connect the controlled service distributions to natural search, we replay service work derived from generation lengths in recorded traces. The slowest domain turns over between adjacent depths in 90.4% of those traces. Table 7 shows the resulting makespan prediction alongside utilization; these are event-simulation results based on the observed search work.

Fixed-work model execution. A recorded search fixes the generated token IDs, requests, scored nodes, verifier inputs, reducer scores, and graph. Each schedule still performs the generator forwards and PRM prefills; only their release and batch formation change. Recomputation arising from the resulting KV pressure remains in the measured time. Figure 2 shows the aggregate model-time and barrier behavior, while Table 8 shows how the gain and batch activity vary with search budget.

During replay, the generator runs autoregressive forwards and consumes the recorded token at each step. The PRM scores the recorded candidate prefixes; the reducer uses the recorded scores so that a change in batch shape cannot alter the next frontier. The schedules may incur different cache recomputation, which remains part of their measured execution. Thus the comparison fixes the search work and its dependency graph while allowing batch formation and cache residence to follow each schedule.

The smaller budget leaves more unused capacity between global rounds and therefore sees the larger relative gain. At both budgets, local progress raises the number of active generation requests, while pooling keeps verifier batches at least as full. Cross-domain pooling accounts for most of the batching gain.

The replay also clarifies the relation between barrier time and model time. Removing a depth barrier makes more work eligible, but an immediate-release schedule can still form narrow batches. The model-time comparison includes both the newly exposed overlap and the cost of serving the released work. The larger generator batches and stable PRM batch volume show why pooling matters alongside local readiness.

B.5 Scored candidates within a time budget

A completed search is one measure of latency; an intermediate deadline asks how many final-list candidates have already received scores. We count those whose scoring finished before a common cutoff, set as a fraction of FastTTS's completion time for each question. Table 9 shows that Branch-

Table 9: Final-list candidates scored by each time cutoff. Cutoffs are fractions of FastTTS completion time; counts are means over questions. BW denotes BranchWeave.

Time cutoff	FastTTS	BW	BW/FastTTS
25%	31.07	216.03	6.96×
50%	123.80	488.57	3.95×
75%	199.60	576.07	2.89×
100%	512.00	599.10	1.17×

Table 10: Gains over FastTTS across search seeds. Aggregate columns are paired geometric means; the range lists per-seed latency gains. GP denotes Goodput.

Workload	E2E speedup	GP gain	Per-seed E2E range
AIME 2025	1.920×	1.328×	1.745–2.270×
AMC 2025	1.776×	1.275×	1.657–1.852×

Weave has more such candidates early in the search. At the first quarter of the reference time, its mean count is nearly seven times higher; the gap narrows as the slower search completes. The final lists can have different lengths, so the endpoint is not a fixed-work comparison.

The cutoff comparison shows how quickly scored candidates accumulate in the final output list. The fixed-work replay above isolates scheduling under an identical dependency graph; here the naturally evolving search also affects the list's length.

B.6 Variation across search seeds

Sampling changes the paths and their lengths, which in turn changes which domains become ready first. Table 10 summarizes question-paired gains across three search seeds for two workloads. Both latency and goodput gains remain above one for each seed. The wider AIME search varies more across seeds, but retains a latency gain even at the lowest observed ratio.

B.7 Wall time and energy during a search

Model time omits the host-side coordination between calls. To see how local progress carries through the complete search call, Table 11 compares its measured wall time and device energy alongside E2E model time. The fixed-work row uses GlobalRound as the comparison, while the natural search row uses FastTTS. Faster model execution persists in wall time, with energy reductions in both cases.

B.8 Measurement and aggregation

For question i, let $L_{G,i}$ and $L_{V,i}$ be generator and verifier time. The model execution metric in the main text is

$$L_i = L_{G,i} + L_{V,i}. \tag{30}$$

Wall time begins at search entry and ends when the result is returned; it also contains host-side scheduling, reduction, tokenization, and communication. The main scaling and ablation figures report L_i to expose model execution, while Table 1 and the fixed-work study report wall time separately.

If u_i is the average number of effective completed tokens per beam, the reported TTS goodput is

$$G_i = \frac{u_i}{L_i}. \tag{31}$$

Table 11: Per-search latency and energy gains. Ratios are comparison/BW for times and energy, and BW/comparison for useful tokens per joule. GR denotes GlobalRound; BW denotes BranchWeave.

Search	E2E speedup	Wall speedup	Energy reduction	Token/J gain
Fixed work vs. GR	1.250×	1.235×	1.264×	1.264×
Natural vs. FastTTS	1.636×	1.496×	1.560×	1.143×

Copied prefixes do not increase u_i, and canceled speculative tokens are physical work that does not increase completed progress. This keeps the numerator tied to useful search output while the denominator accounts for both models.

Let $\bar{C}_i$ be the mean candidate completion time. Precise Goodput is $u_i/\bar{C}_i$, so each completed path's progress is normalized by its own completion time. If two fixed-work schedules complete the same effective tokens, their per-question goodput ratio is the inverse of their model-latency ratio:

$$\frac{G_i^{\mathrm{BW}}}{G_i^{\mathrm{GR}}} = \frac{L_i^{\mathrm{GR}}}{L_i^{\mathrm{BW}}}. \tag{32}$$

This accounts for the matching aggregate model-latency and goodput gains in the fixed-work replay. Energy-to-solution integrates sampled device power over each query's wall-clock interval. For consecutive power samples (t_k, P_k), the per-query estimate is

$$E_i = \sum_k \frac{P_k + P_{k+1}}{2}(t_{k+1} - t_k). \tag{33}$$

It uses the same query boundaries as wall time; the latency columns use model execution and wall time separately. This distinction matters when comparing a scheduler that changes both device activity and host waiting.

For a positive question-paired ratio r_i, all aggregate gains use the geometric mean

$$\mathrm{GM}(r) = \exp\left(\frac{1}{N}\sum_{i=1}^{N} \log r_i\right). \tag{34}$$

Time ratios use comparison/BranchWeave and goodput ratios use BranchWeave/comparison; both are paired within the same question before aggregation. For Table 1, an absolute entry is instead the geometric mean across budgets of each configuration's mean question value. If $\bar{L}_{d,n,m}$ is that mean model latency for dataset d, budget n, and method m, then

$$\widetilde{L}_{d,m} = \exp\left(\frac{1}{|\mathcal{N}|}\sum_{n\in\mathcal{N}} \log \bar{L}_{d,n,m}\right). \tag{35}$$

The same operation is applied to generator time, PRM time, goodput, and energy separately. Because geometric means are nonlinear, the displayed aggregate generator and PRM entries need not sum to the aggregate E2E entry, even though $L_i = L_{G,i} + L_{V,i}$ for every question. The relative columns use the paired ratio above, not a ratio of the displayed absolute aggregates.