RELAYTTS: STRUCTURE-AWARE VERIFIER SERVING FOR EDGE TEST-TIME SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Verifier-guided test-time scaling (TTS) offers a practical route to stronger reasoning on memory-constrained devices, where a larger model may not fit. Its search procedure naturally produces a live frontier of parent groups, each defined by a shared parent prefix. Existing TTS serving systems discard this structure by scoring candidates independently and switching between generator and verifier at model granularity. The result is repeated parent-prefix computation within scoring passes and repeated model movement across them. We present RelayTTS, which makes the live frontier the unit of verifier serving. Parent-complete score packing evaluates shared parent prefixes once across sibling candidates, while a frontier-aware planner trades score-pack capacity against the resident verifier prefix. Layer relay supplies the nonresident blocks only during scoring. Across search procedures and model scales, RelayTTS achieves a $1.91\times$ gain in effective-token throughput and cuts end-to-end latency by 47.8% on average, with a maximum reduction of 88.3%.

1 INTRODUCTION

Test-time scaling improves reasoning by exploring and comparing multiple solution paths at inference time instead of increasing model size. In verifier-guided TTS, a generator expands the retained paths, a process reward model (PRM) ranks their children, and selection determines where to spend the next increment of compute (Snell et al., 2025; Beeching et al., 2024; Lightman et al., 2024; Wang et al., 2023a; Skywork-o1 Team, 2024). This approach is especially useful when device memory rules out a larger model (Park et al., 2025). It also creates an execution pattern unlike ordinary decoding, as token-by-token generation repeatedly gives way to prefill-like verifier scoring. On a constrained device, each handoff consumes time and memory that could otherwise support a broader or deeper search within the same memory budget.

However, current TTS serving systems optimize this loop at model or sequence granularity through phase-aware memory allocation and locality-aware scheduling (Chen et al., 2026; Wang et al., 2026). They do not carry the parent relations of the live frontier into verifier execution. As a result, every child rebuilds its shared parent prefix and, when the two models cannot remain resident together, every scoring round reloads the verifier and later restores the generator (Chen et al., 2026; Yu et al., 2026). Figure 1 shows that these transitions can occupy roughly two-thirds of request time. Keeping part of the verifier resident can remove some transfer, but it also shrinks the score packs that preserve parent groups. The same frontier therefore governs reusable computation and the memory available for verifier residency. Turning this dependence into an efficient execution plan raises two challenges.

The first challenge is how to preserve parent-prefix reuse without fragmenting verifier execution. Search already records which children descend from each retained path. Beam search concentrates expansions around shared survivors, whereas diverse verifier tree search (DVTS) maintains separate subtrees (Beeching et al., 2024). Flattening these candidates rebuilds the same shared parent prefix for every child; issuing one request per parent instead produces small, poorly utilized verifier calls. General prefix-sharing runtimes recover overlap by matching token sequences (Zheng et al., 2024; 2026; Juravsky et al., 2024). Because parentage is known before execution, a scoring interface can keep each group intact while combining several groups into an efficient verifier call.

The second challenge is how to choose verifier residency as the frontier changes. Keeping more verifier layers resident saves future transfers but leaves less memory for each score pack (Chen et al.,

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.

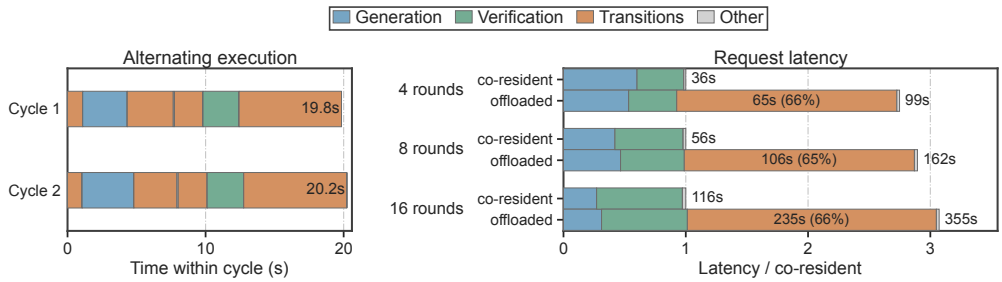

Figure 1: Repeated generation–verification boundaries turn model transitions into a recurring cost. The timeline shows when each transition occurs within a search cycle, and the latency breakdown shows how this cost accumulates as scoring becomes more frequent.

2026; Ukarande et al., 2026; Yu et al., 2026). Several narrow parent groups may fill one pack, while a wide group may no longer fit intact. Splitting that group rebuilds its shared parent prefix and adds another traversal of the nonresident suffix. The best resident verifier prefix thus depends on current group widths and the previous resident boundary as well as model size. Fixed memory partitions cannot follow this change, and policies that choose residency and score-pack capacity separately miss the extra work created at pack boundaries.

We propose RelayTTS, a verifier-serving system that carries the live frontier across the generation–verification boundary. Known parentage forms parent-complete score packs, and their boundaries determine how many verifier blocks remain resident. Layer relay supplies only the nonresident suffix during scoring, avoiding complete model restoration at every boundary. The frontier therefore guides prefix reuse within each pass and verifier movement across rounds. On AIME, AMC, and MATH-500, RelayTTS raises effective-token throughput by $1.91\times$ while lowering end-to-end latency by 47.8% on average and as much as 88.3%.

This paper makes the following contributions.

- We identify a structural mismatch in verifier-guided TTS. Search exposes parent groups before scoring, but serving flattens them even though their composition governs reusable parent-prefix work and feasible verifier residency.
- We develop parent-complete score packing, which evaluates each shared parent prefix once while maintaining dense execution across groups.
- We design a frontier-aware planner that jointly sets score-pack capacity and the resident verifier prefix, together with layer relay for the remaining blocks.
- Experiments with multiple search procedures and model pairs confirm the end-to-end benefit. RelayTTS records a $1.91\times$ effective-token throughput gain; requests complete 47.8% sooner on average and up to 88.3% sooner.

2 Background and Motivation

2.1 Search structure shapes verifier work

Each scoring frontier records the search decisions that produced it. The children of one retained path form a parent group and share the verifier-tokenized parent prefix (Lightman et al., 2024; Wang et al., 2023a; Skywork-o1 Team, 2024). Beam search often places several children under a small set of survivors, whereas DVTS preserves distinct subtrees (Beeching et al., 2024). Two frontiers with the same candidate count can consequently expose very different amounts of parent-prefix computation available for reuse during each verifier pass.

This group structure becomes operationally important when the frontier enters the verifier. Generation carries the KV state of live paths forward, whereas scoring builds temporary state for the shared parent prefix and each new continuation. Scoring children independently rebuilds that prefix; scoring siblings together evaluates it once (Juravsky et al., 2024; Zheng et al., 2026). Candidate count alone therefore describes neither verifier work nor the memory needed to execute it. The parent groups supply the missing context for planning verifier execution.

2.2 Repeated scoring revisits the model boundary

At each round, generation produces a new frontier for the verifier, so savings within one scoring pass address only part of the cost. When the generator and verifier cannot remain resident alongside the live search state, model-level execution loads the verifier before scoring and restores the generator afterward (Chen et al., 2026; Yu et al., 2026). These transitions accumulate with search depth and can dominate even when individual scoring passes are short.

Figure 1 makes this recurrence visible. As scoring becomes more frequent, model transitions remain roughly two-thirds of request time because each verifier pass requires another handoff. Generator decoding is more sensitive to KV-cache capacity than batched verifier scoring (Chen et al., 2026). This asymmetry leaves room for a resident verifier prefix during generation, with the nonresident blocks supplied when scoring begins. How much to retain still depends on the next frontier because verifier weights and scoring state draw from the same memory.

2.3 The frontier changes the best allocation

Partial residency connects model movement to the structure of verifier work. A longer resident verifier prefix removes weight transfer from every score pack. A larger pack keeps more parent groups together, reducing repeated parent-prefix work and scans of the nonresident suffix. Both draw from the same device memory, so the better choice depends on where the current frontier places the next pack boundary. Conventional offloading plans that choose placement at model or layer granularity do not capture this dependence (Sheng et al., 2023; Ukarande et al., 2026; Yu et al., 2026).

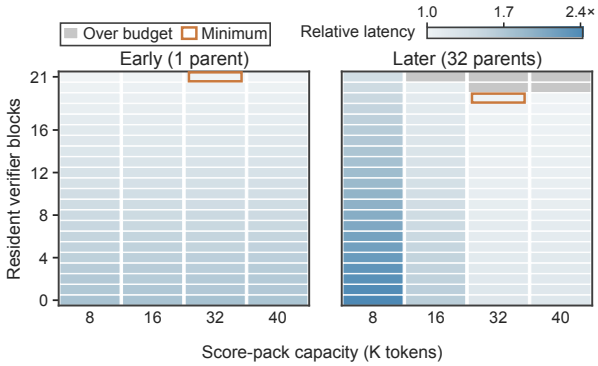

Figure 2: The best allocation shifts with the shape of the arriving frontier. Each panel compares feasible residency and pack choices as parent groups grow; the highlighted choice is the lowest-latency plan.

Figure 2 illustrates the shift with the same model pair throughout. With one parent, the fastest plan retains 21 verifier blocks. With 32 parents, it retains 19 and uses the released capacity for a larger score pack, avoiding another traversal of the relayed layers. The model footprints are unchanged, yet the frontier changes where memory is most useful. As the tree grows, early narrow frontiers may favor residency, while later frontiers need more scoring state to keep another parent group intact. A static split either gives up useful residency early or fragments scoring later. Verifier execution should therefore be organized around parent groups, with residency chosen from the pack boundaries of the live frontier.

3 RelayTTS

RelayTTS keeps the search frontier explicit until verifier scores return to selection. Figure 3 follows one round from the selected parents through packing, planning, and verification with layer relay.

3.1 Parent-complete score packs

At round t, RelayTTS forms score packs from the live parents $\mathcal{P}_t$. Parent p has verifier-tokenized children $\mathcal{G}_p$, exact shared-prefix length d_p, and private suffix length s_i for child i.

For complete groups, the verifier work in the round is:

$$T_t^{\mathrm{pc}} = \sum_{p \in \mathcal{P}_t} \left[d_p + \sum_{i \in \mathcal{G}_p} s_i \right]. \tag{1}$$

Flat scoring processes the shared parent prefix of length d_p for every child. RelayTTS retains its KV state within the group, so each suffix attends to that state and its own preceding tokens, eliminating

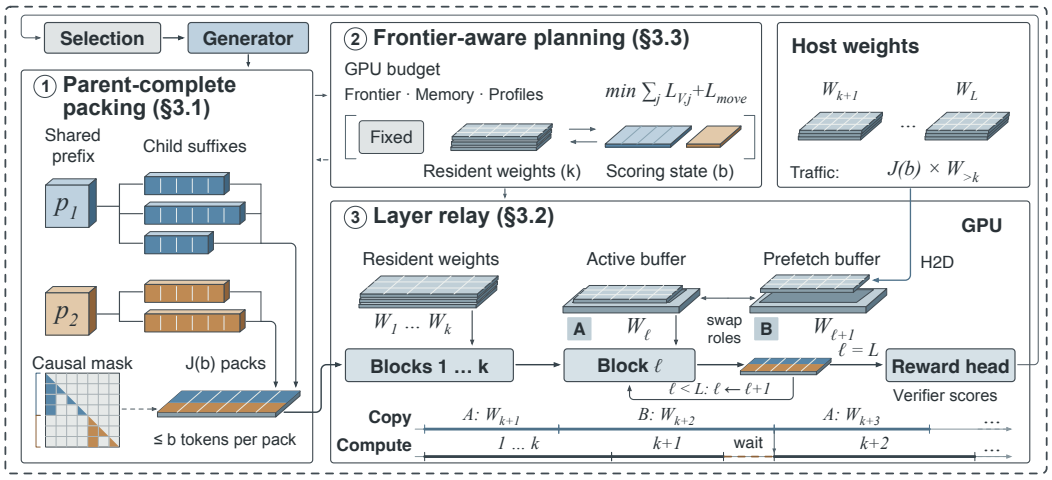

Figure 3: Overview of RelayTTS. Search ancestry forms parent-complete score packs, the planner jointly chooses score-pack capacity and verifier residency, and layer relay supplies the remaining blocks during scoring.

$(|\mathcal{G}_p| - 1)d_p$ repeated positions. Groups remain intact when they fit within the score-pack capacity; several groups can share a pack, while an oversized group is split into adjacent fragments. The benefit grows with both sibling count and shared-prefix depth, so candidate count alone does not determine verifier work.

The choice of packs also determines how often the verifier reads nonresident weights. If capacity b produces $J_t(b)$ score packs and the layers after resident boundary k contain $W_{>k}$ bytes, the resulting weight traffic for the round is

$$Q_t^W(k, b) = J_t(b)W_{>k}. \tag{2}$$

Splitting can repeat both parent-prefix computation and a traversal of the nonresident weights, making group shape and pack count part of the memory plan. A boundary that splits one more parent group can therefore increase both token computation and weight traffic.

3.2 Layer relay for repeated scoring

Parent-complete packing reduces verifier computation, while partial residency avoids a complete model transition at the next scoring round. RelayTTS keeps blocks $1, \dots, k$ on the device and stores the remaining blocks in model order in host memory. As the compute stream evaluates block ℓ, a copy stream loads block $\ell + 1$ into the other of two device buffers. Each score pack traverses the nonresident suffix once in model order.

Scoring begins on the resident verifier prefix while the first nonresident block is prefetched. Computation on one block can overlap the next transfer; only an exposed wait delays scoring. For pack j, let $C_{j,\ell}$ be block-ℓ compute time and $f = k + 1$. With startup wait $\widehat{E}_j(k)$ and concurrent copy–compute duration $\widehat{O}_{j,\ell}$, the estimated score latency is

$$\widehat{L}_{V,j}(k, b) = \sum_{\ell=1}^{k} C_{j,\ell} + \widehat{E}_j(k) + \sum_{\ell=f}^{L-1} \widehat{O}_{j,\ell} + C_{j,L} + C_j^{\text{out}}. \tag{3}$$

The concurrent durations include copy–compute interference, and the final terms cover the last block and reward output. A fully resident plan uses the measured verifier forward pass.

3.3 Planning from the live frontier

The score-pack capacity and resident boundary draw from the same device-memory budget. A capacity b and boundary k are feasible when

$$M_{\text{fixed}} + M_V^{\text{res}}(k) + M_V^{\text{KV}}(b) + M_V^{\text{act}}(b) \le M_{\text{GPU}}, \tag{4}$$

where the fixed footprint contains the generator, verifier endpoints, relay buffers, and runtime reserve. Profiles supply the memory peaks and concurrent copy–compute time at each profiled score-pack capacity considered by the planner.

At runtime, RelayTTS enumerates resident boundaries and profiled score-pack capacities $\mathcal{B}$. Let $\mathcal{F}_t$ denote the pairs that satisfy Equation 4 for the current frontier. Parent-complete packing gives the pack set $\mathcal{J}_t(b)$ for each capacity, and the planner selects

$$(k_t^*, b_t^*) = \arg \min_{(k,b)\in\mathcal{F}_t} \left[\sum_{j\in\mathcal{J}_t(b)} \widehat{L}_{V,j}(k,b) + \widehat{L}_{\text{move}}(k_{t-1}, k) \right]. \tag{5}$$

The objective combines score latency across packs with the cost of moving the resident boundary. Another resident block avoids its future transfer, but the lost score-pack capacity may introduce an extra scan. The planner therefore compares their combined effect on total scoring latency.

The live parents supply an initial plan before generation, and exact parent-prefix and suffix lengths finalize it afterward. Within a request, the resident verifier prefix stays fixed or becomes shorter as the frontier grows; a new request can choose any boundary.

4 Evaluation

4.1 Experimental setup

Platform and models. We use vLLM 0.9.2 and BF16 models on an NVIDIA RTX 4090 (24 GiB) (Kwon et al., 2023). The generators are Qwen2.5-Math-1.5B, Qwen2.5-3B, and Qwen2.5-Math-7B (Yang et al., 2024); the verifiers are the 1.5B Skywork PRM (Skywork-o1 Team, 2024) and 7B Math-Shepherd Mistral PRM (Wang et al., 2023a). Model pairs list the generator first.

Workloads and search. We evaluate 1.5B+7B beam search and DVTS at search budgets of 16–512 candidate paths on AIME 2026 and AMC 12 2025 (Balunovic et al., 2025), and on MATH-500 (Hendrycks et al., 2021). Scale studies span 1.5B+1.5B, 3B+7B, and 7B+7B; structure studies vary the number of scoring rounds and the branching factor.

Baselines. vLLM (Kwon et al., 2023) serves the generator and verifier as separate engines. Designed for resource-constrained TTS serving, **FastTTS** (Chen et al., 2026) combines speculative beam extension, prefix-aware generation scheduling, and asymmetric memory allocation. All systems use the same models and search settings.

Metrics. End-to-end latency includes generation, scoring, selection, and planning. Effective-token throughput counts newly generated reasoning tokens per second of request time, excluding tokens copied when paths branch. Prefix reuse is the fraction of verifier input shared among siblings, while Top-1 accuracy measures search quality. Aggregate speedups and throughput ratios use the geometric mean across configurations within each study.

4.2 End-to-end performance

Table 1 shows that RelayTTS's latency advantage widens with search budget across all workloads. Across the principal and model-scale studies, it cuts latency by 47.8% on average and up to 88.3%, corresponding to geometric-mean speedups of 1.92× and 1.95× over FastTTS and vLLM.

The gap is largest under beam search, whose retained paths expose longer shared prefixes and more siblings. Both baselines flatten these groups, and RelayTTS reaches 1.87–2.08× speedup over FastTTS at the largest setting. DVTS forms narrower groups with longer private suffixes, so reusable parent-prefix work predicts the gain more closely than candidate count or nominal search budget.

RelayTTS also achieves the best or tied-best Top-1 accuracy in five of the six workload–search combinations in Table 2. DVTS attains higher Top-1 across these workloads, while its longer private continuations offer less shared verifier work than beam search. Search quality and verifier cost therefore depend on different properties of the surviving paths, so the procedure with greater prefix reuse need not attain higher accuracy.

Table 1: End-to-end latency for the principal generator–verifier pair as the search budget increases. Values are in seconds, and lower latency indicates faster execution.

		Beam search						DVTS					
Workload	System	16	32	64	128	256	512	16	32	64	128	256	512
AIME	vLLM	19.8	31.2	51.0	92.7	169.0	329.6	17.9	23.5	34.7	59.0	110.1	219.2
	FastTTS	18.5	28.4	51.2	92.7	165.2	316.2	**17.6**	23.4	35.5	58.1	95.1	209.1
	RelayTTS	**15.5**	**21.2**	**32.4**	**51.6**	**85.4**	**152.3**	18.4	**21.1**	**29.3**	**44.9**	**68.4**	**181.8**
AMC	vLLM	18.3	27.4	46.0	84.6	153.9	294.3	15.6	19.3	28.0	48.9	93.5	178.0
	FastTTS	17.0	26.6	47.8	84.9	146.5	285.5	15.3	20.2	29.4	49.7	83.7	170.9
	RelayTTS	**13.4**	**19.1**	**32.0**	**46.6**	**77.3**	**152.4**	**15.0**	**17.9**	**25.4**	**40.9**	**64.9**	**147.1**
MATH-500	vLLM	14.6	22.5	38.8	70.2	130.2	250.7	12.6	17.6	22.7	37.0	68.2	130.4
	FastTTS	14.1	20.6	35.3	65.0	114.4	236.6	11.7	16.1	22.4	36.7	63.8	129.4
	RelayTTS	**11.0**	**15.2**	**24.8**	**35.6**	**64.0**	**120.4**	**10.8**	**14.7**	**19.5**	**28.3**	**46.0**	**108.3**

Table 2: Top-1 accuracy (%) for beam search and DVTS on AIME, AMC, and MATH-500, averaged across search budgets for each serving system.

	Beam search			DVTS		
Workload	vLLM	FastTTS	RelayTTS	vLLM	FastTTS	RelayTTS
AIME	1.7	1.7	1.7	7.8	12.8	12.8
AMC	23.0	24.7	23.0	40.3	40.7	43.7
MATH-500	53.4	53.7	55.6	64.4	67.2	67.5

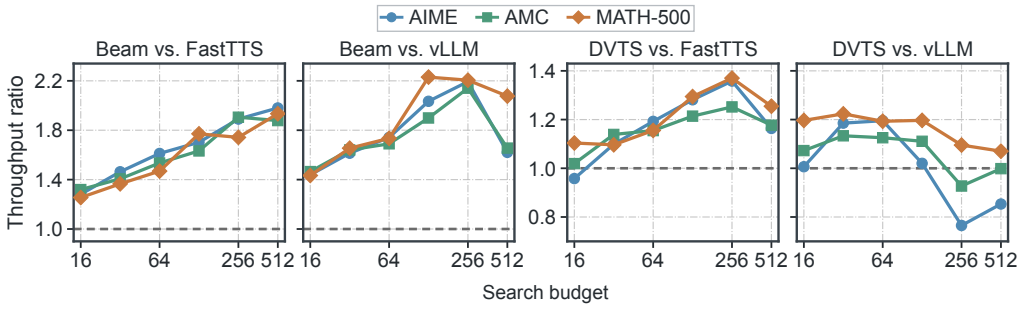

Figure 4: RelayTTS effective-token throughput relative to FastTTS or vLLM across beam-search and DVTS budgets on all three workloads.

Effective-token throughput follows the latency trend. Its gain over FastTTS is 1.60× for beam search and 1.18× for DVTS; across the principal and model-scale studies, the gain is 1.91× over FastTTS and 2.00× over vLLM (Figure 4).

4.3 Scaling across model pairs

Model scale exposes two operating regimes. With the 1.5B+1.5B pair, both models remain resident, so its 1.52× average speedup over FastTTS comes mainly from parent-prefix reuse (Figure 5). With 7B verifiers, partial residency avoids complete model transitions, lifting average speedup above 4.5× and the maximum to 8.52×.

For both 7B-verifier pairs, transitions take a larger share of small-budget requests. As candidate work grows, speedup narrows but remains 2.74–3.67× at budget 128. The 7B+7B pair spends more time generating than 3B+7B, so verifier savings form a smaller share. Co-residency isolates parent-prefix reuse in the smallest pair; larger pairs also depend on scoring frequency and the parent groups exposed at each round of verifier scoring.

4.4 Sensitivity to search structure

The contrast between co-resident pairs and those that require layer relay also depends on how often scoring occurs and how much ancestry each round exposes. Figure 6 varies scoring frequency and branching factor while keeping the models fixed.

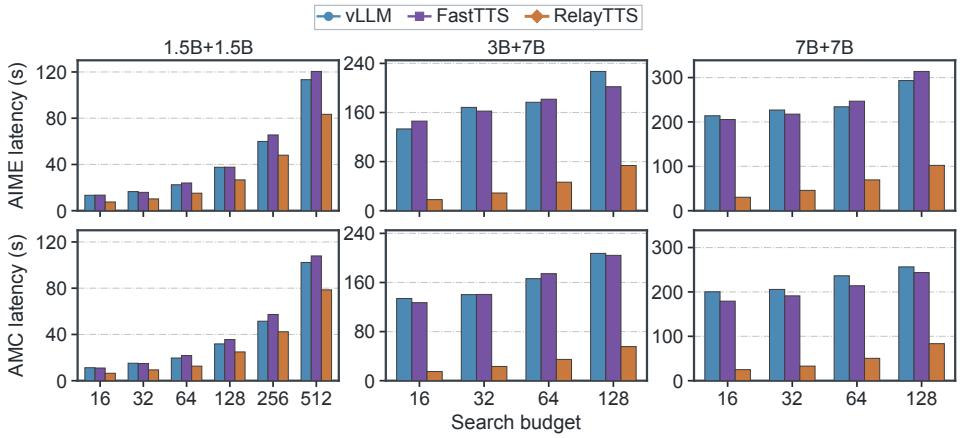

Figure 5: Beam-search latency across model pairs with co-resident execution or layer relay. Larger pairs benefit from both parent-prefix reuse and fewer model transitions.

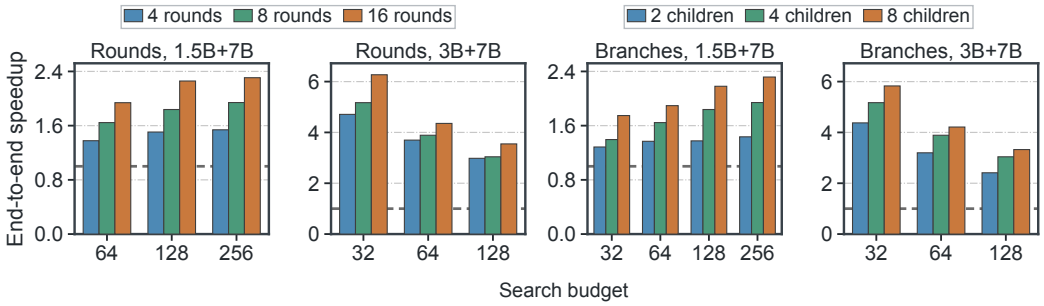

Figure 6: End-to-end speedup over FastTTS as scoring frequency and branching factor vary independently. More rounds expose transition savings, while wider groups expose parent-prefix reuse.

As scoring becomes more frequent, RelayTTS's speedup over FastTTS grows in every setting and reaches 6.27× for 3B+7B because resident verifier blocks avoid more complete model transitions. As the branching factor grows, speedup reaches 2.32× and 5.83× for the two model pairs because parent-prefix reuse rises from about 48% to 79%.

The sweeps separate repeated model transitions from reuse within a scoring round. Additional rounds make resident verifier blocks valuable more often. A larger branching factor gives each shared parent prefix more continuations, letting complete groups spend more work on new suffixes instead of prefixes already evaluated earlier in the same pass. These are distinct opportunities even at the same search budget. More frequent scoring increases the number of model handoffs, whereas wider branching changes how much computation siblings share at each handoff. This explains why neither model size nor candidate count alone predicts the benefit of structured verifier execution.

4.5 Mechanism Analysis

Figure 7 shows that verifier savings follow ancestry more closely than candidate count. Beam search reuses 68.6% of verifier input and accelerates scoring by 2.87× over FastTTS on average; DVTS reuses 47.4% and reaches 1.77×. Variation across workloads follows path divergence, so equal candidate budgets need not produce equal verifier work.

Removing parent grouping raises verifier time by 9.4–10.5% at budget 64 and by 20.2–22.0% at budget 256 across AIME and AMC because siblings repeat parent work and require more relayed-layer scans. Replacing overlapped relay with synchronous relay changes verifier time by no more than 3.3% at the same budgets. The request-level gains thus come chiefly from persistent residency and parent grouping; overlap within one pass contributes much less.

The ablation also explains why nominal search budget is an incomplete predictor of verifier time. A budget can yield a few wide parent groups under beam search or many narrower groups under DVTS. Longer shared parent prefixes remove more repeated work, while preserving a wide group across a

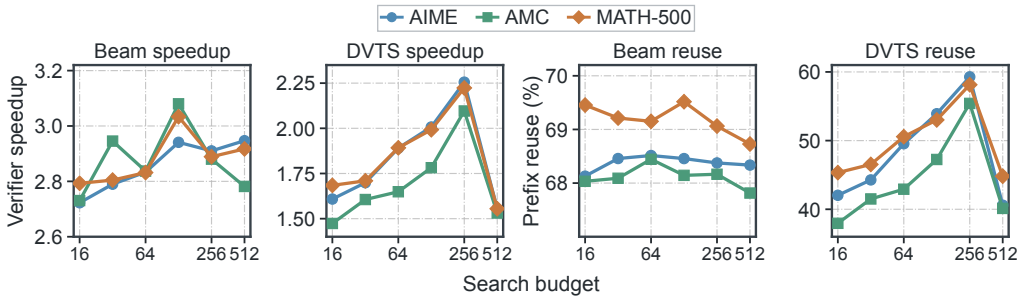

Figure 7: RelayTTS verifier speedup over FastTTS and the fraction of verifier input reused through shared parent prefixes across workloads and search budgets.

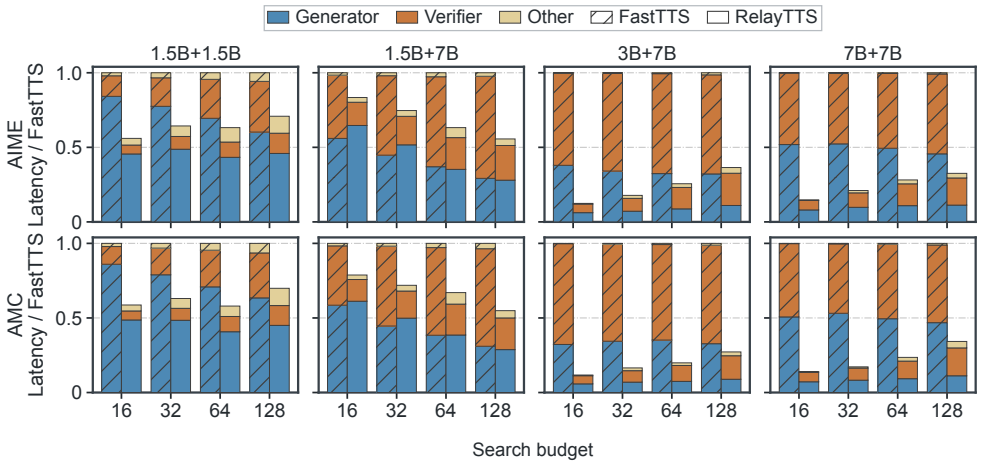

Figure 8: Request latency across model pairs, decomposed by phase and normalized to FastTTS. Parent-prefix reuse reduces verifier time, while residency reduces transition overhead.

pack boundary avoids another scan of the relayed layers. Workloads with the same candidate budget can therefore exhibit different verifier speedups.

Figure 8 shows where the savings occur within a request. With both models resident, parent-prefix reuse shortens verification; with larger verifiers, resident blocks also remove much of the transition time. Generation occupies a larger fraction as model size grows, tempering end-to-end speedup. Grouping therefore changes verifier work in every regime, while residency matters once the verifier no longer fits beside generation. The breakdown also connects verifier acceleration to request latency. Faster scoring has a larger end-to-end effect when verification and transitions dominate; as generation takes a greater share, the same verifier improvement accounts for a smaller fraction of the request.

4.6 Adapting to memory pressure

The same coupling appears when available memory shrinks. Releasing verifier blocks makes room for scoring state; if the larger pack keeps another parent group intact, it avoids both repeated parent-prefix work and another scan of the relayed layers. The most resident plan need not be the fastest.

At 21 and 18 GiB, RelayTTS retains 26 and 19 blocks while staying within 6.8% and 10.3% of the latency at 24 GiB (Figure 9). Under round-level planning, AIME reaches its lowest latency with score-pack capacity capped at 32K, while AMC favors 40K. The choice depends on the parent-group boundaries of the arriving frontier because extra capacity helps only when it keeps another group intact or removes a scoring pass. Online planning can adapt to these shifts instead of fixing one request-wide setting, and extra residency alone does not guarantee lower latency.

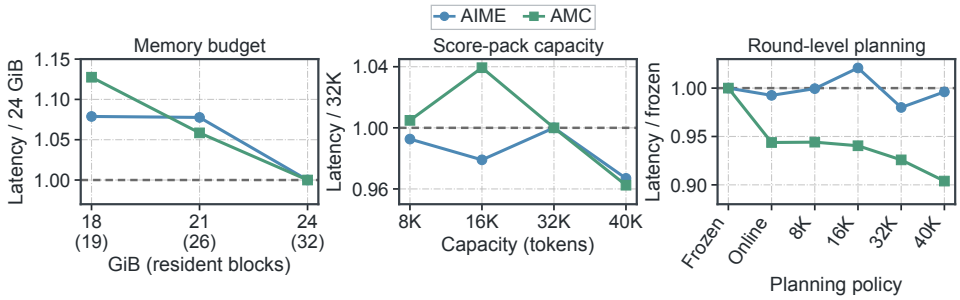

Figure 9: Latency sensitivity to memory budget, score-pack capacity, and round-level planning. The fastest setting shifts as available memory and the arriving frontier change.

5 Related Work

Test-time scaling and verification. Test-time scaling spends inference compute on sampling, search, and verifier feedback (Wang et al., 2023b; Yao et al., 2023a; Cobbe et al., 2021; DeepSeek-AI et al., 2025). Process reward models guide intermediate steps (Wang et al., 2023a; Skywork-o1 Team, 2024), and tools extend generation (Yao et al., 2023b; Schick et al., 2023). FastTTS and locality-aware scheduling lower the serving cost (Chen et al., 2026; Wang et al., 2026). Speculative decoding has a related generate–verify structure, using learned drafts (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Cai et al., 2024a; Li et al., 2024b;c; Liu et al., 2024a; Sadhukhan et al., 2025), verification trees (Miao et al., 2024; Chen et al., 2024), or self-speculation (Elhoushi et al., 2024; Zhang et al., 2024). RelayTTS carries parent relations from reasoning search into verifier execution to guide reuse and residency.

Memory-efficient serving. Memory-efficient serving spans attention kernels and virtual memory (Dao et al., 2022; Dao, 2024; Prabhu et al., 2025); KV-cache reduction and selection (Liu et al., 2024c; Hooper et al., 2024; Zhang et al., 2023; Xiao et al., 2024; Liu et al., 2024b; Li et al., 2024a; Cai et al., 2024b; Tang et al., 2024); batching and fused runtimes (Agrawal et al., 2024; Ye et al., 2025; Zhu et al., 2025); and weight offloading (Song et al., 2024; Xue et al., 2024; Alizadeh et al., 2024; Xu et al., 2025; Borzunov et al., 2023). Phase-aware systems separate prefill from decoding (Zhong et al., 2024; Patel et al., 2024; Du et al., 2025). At each verifier frontier, RelayTTS couples placement with batching because parent groups determine score packs and weight traffic.

Prefix reuse and structured execution. Radix trees, shared-prefix attention, and global batching reuse context across requests (Zheng et al., 2024; Juravsky et al., 2024; Zheng et al., 2026). Locality-aware scheduling exploits path locality during search (Wang et al., 2026), while speculative systems batch verification trees (Miao et al., 2024; Chen et al., 2024; Cai et al., 2024a; Li et al., 2024c). Multi-model serving coordinates placement and migration (Duan et al., 2024; Sun et al., 2024; Fu et al., 2024; Xiang et al., 2025; Yu et al., 2026). These methods use structure within generation, verification, or model scheduling. Search parentage is available before scoring and can shape verifier execution before it begins. RelayTTS uses its group boundaries to reveal repeated prefix work within each pass and relayed weight traffic across scoring rounds, allowing the same structure to guide reuse and residency.

6 Conclusion

RelayTTS treats the search frontier as a serving interface. Parent-complete score packing reuses shared prefixes, while frontier-aware planning and layer relay reduce weight movement across scoring rounds. Together, these mechanisms align verifier execution with the parent structure that reasoning search already produces. Sibling candidates share prefix work, and the resulting pack boundaries determine where limited device memory is most valuable. This connection follows the parent groups exposed at each scoring round. Across search procedures and model pairs, requests finish 47.8% sooner on average and up to 88.3% sooner. Carrying the frontier across the generation–verification boundary turns search structure into a practical systems resource. More of the device budget can then support the reasoning paths being explored and compared.

AI USE STATEMENT

We used generative AI tools to assist with code generation and review, mathematical derivations and proofs, identifying related work, and edits for clarity. The authors reviewed and verified the AI-assisted material and take responsibility for the paper, code, and reported results.

ETHICS STATEMENT

This work studies the serving of LLM reasoning systems using publicly available models 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

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)*, 2024.

Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S 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*, 2024.

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. In *Advances in Neural Information Processing Systems*, 2025.

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

Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2023.

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 *International Conference on Machine Learning*, 2024a.

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*, 2024b.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

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*, 2026.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable and robust speculative decoding. In *Advances in Neural Information Processing Systems*, 2024.

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.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

DeepSeek-AI et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

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*, 2025.

Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple LLM serving. In *International Conference on Machine Learning*, 2024.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, et al. LayerSkip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

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*, 2024.

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*, 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*, 2024.

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.

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*, 2023.

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

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*, 2024b.

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*, 2024c.

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 *International Conference on Learning Representations*, 2024.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. In *International Conference on Machine Learning*, 2024a.

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*, 2024b.

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*, 2024c.

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*, 2024.

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

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, 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*, 2025.

Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. MagicDec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. In *International Conference on Learning Representations*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

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 *International Conference on Machine Learning*, 2023.

Skywork-o1 Team. Skywork-o1 open series. https://huggingface.co/Skywork, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations*, 2025.

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*, 2024.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems*, 2018.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation*, 2024.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Aditya Ukarande, Deep Shekhar, Marc Blackstein, and Ram Rangan. Efficient, vram-constrained xlm inference on clients. In *Proceedings of Machine Learning and Systems*, 2026.

Hsing-Ti Wang, Hung-Tso Shiao, and Chia-Lin Yang. Locality-aware beam scheduling for efficient test-time compute with a consumer-grade GPU. In *Proceedings of Machine Learning and Systems*, 2026.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu 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 *International Conference on Learning Representations*, 2023b.

Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. Aegaeon: Effective gpu pooling for concurrent llm serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024.

Daliang Xu, Wangsong Yin, Hao Zhang, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. EdgeLLM: Fast on-device LLM inference with speculative decoding. *IEEE Transactions on Mobile Computing*, 2025.

Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. PowerInfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2.5-Math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023b.

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.

Shan Yu, Yifan Qiao, Mingyuan Ma, Yangmin Li, Shuo Yang, Xinyuan Tong, Yang Wang, Zhiqiang Xie, Yuwei An, Shiyi Cao, Ke Bao, Deepak Vij, Xiaoning Ding, Yichen Wang, Qingda Lu, Zhong Wang, Gao Gao, Harry Xu, Junyi Shu, Jiarong Xing, and Ying Sheng. Prism: Cost-efficient multi-LLM serving via GPU memory ballooning. In *20th USENIX Symposium on Operating Systems Design and Implementation*, 2026.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

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*, 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*, 2024.

Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. BatchLLM: Optimizing large batched LLM inference with global prefix sharing and throughput-oriented token batching. In *Proceedings of Machine Learning and Systems*, 2026.

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*, 2024.

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation*, 2025.

A Analysis of Tree-Structured Verification

Search ancestry changes verifier cost through two coupled effects. Parent-prefix reuse removes repeated work, while pack fragmentation can reintroduce that work and repeat scans of relayed weights. Score-pack capacity and the resident boundary connect both effects through the memory budget.

A.1 Shared computation within a parent group

For parent group p, let d_p denote the length of its shared parent prefix, $\mathcal{G}_p$ the children, and s_i the private suffix length of child i. Independent scoring evaluates

$$T^{\text{flat}} = \sum_{p \in \mathcal{P}} \left[|\mathcal{G}_p| d_p + \sum_{i \in \mathcal{G}_p} s_i \right]. \tag{6}$$

When each parent group fits in one score pack, parent-complete scoring evaluates

$$T^{\text{pc}} = \sum_{p \in \mathcal{P}} \left[d_p + \sum_{i \in \mathcal{G}_p} s_i \right], \tag{7}$$

the round-independent form of Equation 1.

Subtracting Equation 7 from Equation 6 cancels every suffix term and gives

$$T^{\text{flat}} - T^{\text{pc}} = \sum_{p \in \mathcal{P}} (|\mathcal{G}_p| - 1) d_p \tag{8}$$

Each additional child therefore saves one evaluation of the shared parent prefix.

For a parent with g children of equal suffix length s, the ratio between flat and parent-complete work is

$$\rho(g, d, s) = \frac{g(d+s)}{d+gs}. \tag{9}$$

The ratio approaches g when the shared parent prefix dominates the continuation and approaches one when private suffix work dominates. Short beam-search expansions therefore expose more verifier reuse than the longer continuations produced by DVTS.

A.2 Fragmentation and relayed weight traffic

A score-pack capacity can divide one parent group across several verifier passes. Let $F_p(b)$ be the number of fragments created for parent p under capacity b. Every fragment reconstructs the shared parent prefix before evaluating its suffixes, giving

$$T^{\text{frag}}(b) = \sum_{p \in \mathcal{P}} \left[F_p(b) d_p + \sum_{i \in \mathcal{G}_p} s_i \right]. \tag{10}$$

Relative to a parent-complete pack, fragmentation adds

$$T^{\text{frag}}(b) - T^{\text{pc}} = \sum_{p \in \mathcal{P}} (F_p(b) - 1) d_p. \tag{11}$$

The additional work is the shared parent prefix reconstructed in each new fragment; private suffix work is unchanged.

Combining Equations 6 and 10 gives the reuse that survives fragmentation:

$$T^{\text{flat}} - T^{\text{frag}}(b) = \sum_{p \in \mathcal{P}} \big(|\mathcal{G}_p| - F_p(b) \big) d_p. \tag{12}$$

Algorithm 1 Parent-complete score packing

Require: Ordered parent groups $\{(p, \mathcal{G}_p)\}$; score-pack capacity b
Ensure: Ordered score packs $\mathcal{J}$

$\mathcal{J} \leftarrow [\,]$; $j \leftarrow [\,]$
for all $(p, \mathcal{G}_p)$ in search order **do**
  $w_p \leftarrow d_p + \sum_{i \in \mathcal{G}_p} s_i$
  if $w_p \le b$ **then**
    if $\mathrm{work}(j) + w_p > b$ **then**
      append j to $\mathcal{J}$; $j \leftarrow [\,]$
    end if
    append the complete group $(p, \mathcal{G}_p)$ to j
  else
    if $j \neq [\,]$ **then**
      append j to $\mathcal{J}$; $j \leftarrow [\,]$
    end if
    partition $\mathcal{G}_p$ into maximal adjacent fragments of work at most b
    append the fragments to $\mathcal{J}$ in candidate order
  end if
end for
if $j \neq [\,]$ **then**
  append j to $\mathcal{J}$
end if
return $\mathcal{J}$

A split group still benefits from sharing while it has fewer fragments than children. Pack boundaries matter twice because they can reconstruct a shared parent prefix and require another traversal of nonresident weights.

The effect of score-pack capacity is especially clear when a parent has g children with the same private-suffix length s. A fragment holding q children evaluates $d + qs$ positions. Provided one child fits, the largest such fragment and the resulting fragment count are

$$q(b) = \left\lfloor \frac{b-d}{s} \right\rfloor, \qquad F(b) = \left\lceil \frac{g}{q(b)} \right\rceil, \qquad b \ge d + s. \tag{13}$$

The group then evaluates $F(b)d + gs$ positions. Increasing b within one interval of Equation 13 may leave the number of fragments unchanged. Once another child fits, however, the fragment count can fall, removing an entire evaluation of the shared parent prefix. If that fragment occupied its own score pack, the same change avoids one scan of the nonresident verifier suffix. The threshold depends on the shared parent prefix and child lengths, not simply on the number of candidates in the frontier.

Fragmentation also affects model traffic. If packing produces $J(b)$ verifier passes and layers after boundary k contain $W_{>k}$ bytes, the round transfers

$$Q^W(k, b) = J(b) W_{>k}. \tag{14}$$

Equal token counts can therefore induce different verifier work and weight traffic. A split parent repeats its shared parent prefix, and every additional pack scans the relayed suffix once more.

Algorithm 1 gives the packing rule. Complete parent groups remain atomic whenever possible and fill packs in search order. A group is divided only when its own work exceeds b, with the resulting fragments kept adjacent.

Ordered-pack optimality. When every parent group fits within b, Algorithm 1 uses the fewest packs among layouts that preserve group order and keep groups intact.

Proof. The first greedy pack ends at the furthest group whose cumulative work fits within b. The first pack of any feasible ordered layout can end no later. Applying the same argument after each boundary shows inductively that the r-th greedy pack reaches at least as far as the r-th pack of any other feasible layout. Hence no ordered parent-complete layout reaches the final group with fewer packs. Because each pack traverses the nonresident suffix once, this also minimizes layer scans for a fixed resident boundary.

A.3 Overlapping relay with scoring

Blocks $1, \dots, k$ are resident, and $f = k + 1$ is the first relayed block. Let $C_{j,\ell}$ and $H_{j,\ell}$ denote the compute and copy time of block ℓ for pack j. The resident verifier prefix provides an initial computation window

$$R_j(k) = \sum_{\ell=1}^{k} C_{j,\ell}. \tag{15}$$

If prefetch begins at $\tau_j(k)$, the first relayed block arrives at $\tau_j(k) + H_{j,f}$.

Write $a_{j,\ell}$ and $z_{j,\ell}$ for copy and compute completion, measured from the start of the resident pass. With a serial copy stream and two alternating device slots,

$$\begin{aligned} a_{j,f} &= \tau_j(k) + H_{j,f}, \\ a_{j,f+1} &= a_{j,f} + H_{j,f+1}, && f+1 \le L, \\ a_{j,\ell} &= \max\{a_{j,\ell-1}, z_{j,\ell-2}\} + H_{j,\ell}, && f+2 \le \ell \le L, \\ z_{j,f-1} &= R_j(k), \\ z_{j,\ell} &= \max\{z_{j,\ell-1}, a_{j,\ell}\} + C_{j,\ell}, && f \le \ell \le L. \end{aligned} \tag{16}$$

The copy of block ℓ follows the previous copy and, once both slots are occupied, waits for block $\ell - 2$ to release a slot. Block ℓ can execute when its weights and the output of the preceding block are ready.

Earliest two-buffer schedule. Subject to model order, a serial copy stream, and two relay slots, Equation 16 gives the earliest completion time of every copied and computed block.

Proof. The first two copies occupy the initially free slots and finish at the first two times in Equation 16. Every later copy is constrained by the preceding copy and by release of the slot used two blocks earlier; the maximum of those completion times is therefore its earliest feasible start. Computation for block ℓ is similarly constrained by block $\ell - 1$ and by arrival of its own weights. Applying these two constraints inductively gives the earliest feasible schedule for the complete suffix.

The exposed wait before block ℓ is

$$e_{j,\ell} = \max\{0, a_{j,\ell} - z_{j,\ell-1}\}. \tag{17}$$

Pack latency is then

$$L_{V,j} = \sum_{\ell=1}^{L} C_{j,\ell} + \sum_{\ell=f}^{L} e_{j,\ell} + C_j^{\text{out}}. \tag{18}$$

The three terms account for block computation, exposed transfer, and reward output. They also give the simple bounds

$$\sum_{\ell=1}^{L} C_{j,\ell} + C_j^{\text{out}} \le L_{V,j} \le \sum_{\ell=1}^{L} C_{j,\ell} + \sum_{\ell=f}^{L} H_{j,\ell} + C_j^{\text{out}}. \tag{19}$$

The lower bound is reached when all copies are hidden; the upper bound is the corresponding serial execution. Equation 3 uses measured concurrent durations between these bounds because copy and compute can slow one another when they share the memory subsystem.

For a fixed pack, residency removes block transfers and extends the computation available before the first relayed block is needed. A larger pack performs more computation per block without changing the bytes copied, allowing more of the next transfer to overlap. The latency profile captures these effects together with copy–compute interference.

A.4 Coupling score-pack capacity and residency

With a fixed score-pack capacity, retaining another verifier block removes one transfer from every pass. The trade-off changes if that block reduces capacity. Suppose boundary k admits capacity b and produces $J = J(b)$ packs. Retaining block $k + 1$ leaves capacity b' and produces $J' = J(b')$ packs. For block size W_{k+1},

$$\begin{aligned} Q^W(k+1, b') - Q^W(k, b) &= J' W_{>k+1} - J(W_{k+1} + W_{>k+1}) \\ &= (J' - J) W_{>k+1} - J W_{k+1}. \end{aligned} \tag{20}$$

Algorithm 2 Joint score-pack and residency planning

Require: Round state, previous boundary k_{t-1}, capacities $\mathcal{B}$
Ensure: Score packs $\mathcal{J}^*$ and resident boundary k^*

```
c* ← ∞
for all b ∈ B do
    J ← ParentCompletePack(P_t, b)
    for all k ∈ K_t do
        if (k, b) satisfies Equation 4 then
            c ← Σ_{j∈J} L̂_{V,j}(k, b) + L̂_move(k_{t−1}, k)
            if c < c* then
                (c*, J*, k*) ← (c, J, k)
            end if
        end if
    end for
end for
return (J*, k*)
```

When another resident block helps. If pack count is unchanged, retaining block $k+1$ always removes JW_{k+1} bytes of round traffic. If the smaller capacity creates more packs, residency reduces traffic only when

$$JW_{k+1} > (J' - J)W_{>k+1}. \tag{21}$$

The right-hand side is the added scan of every layer that remains nonresident. An extra pack may cost more than the transfer saved by retaining one block, and it may also split a parent group and incur Equation 11. Maximizing residency before choosing score-pack capacity can therefore miss the faster plan.

The memory constraint in Equation 4 adds verifier KV and activations to this comparison. The planner evaluates complete score latency, including fragmentation and exposed transfer, rather than transferred bytes alone.

A.5 Planning over a growing tree

Before generation begins, the live parent paths already provide an estimate of the next score pass. If r_p is the planner's token count for live parent p, then

$$\widehat{T}_t^{\text{early}} = \sum_{p \in \mathcal{P}_t} r_p, \tag{22}$$

which reflects the history inherited by the next continuations. This gives an initial residency plan before generation. Once the new continuations are tokenized for the verifier, their shared parent prefixes and private suffixes determine the final packs and boundary. The boundary stays fixed or moves toward fewer resident blocks during a request; a new request may restore a longer resident verifier prefix, with its transfer included in the move cost.

Algorithm 2 gives the final selection. Let $\mathcal{K}_t$ contain the admissible resident boundaries and $\mathcal{B}$ the profiled score-pack capacities. The move term charges any change from the previous boundary.

At one scoring boundary, parent-complete packing is linear in the parent groups and emitted packs. Prefix sums make each memory query constant time, and scanning all resident boundaries is linear in L. Planning therefore takes

$$O(|\mathcal{B}|(L + P + J)) \tag{23}$$

for P parent groups and at most J emitted packs per capacity. The search covers only profiled capacities and layer boundaries, keeping planning small relative to model execution.

B From Search State to Verifier Execution

B.1 Structured scoring semantics

Search supplies a frontier of candidates with their immediate parents and their order for survivor selection. Parent identity determines the groups in Algorithm 1. After applying the verifier's input template, the shared parent prefix is computed from the actual verifier tokens, so sharing follows the sequence seen by the model rather than the generator's tokenization.

Several parent groups can occupy one score pack. Each child attends to its own group's shared parent prefix and private causal continuation; prefixes from different parents remain separate. If a group spans multiple packs, each fragment evaluates that prefix again, as reflected in Equation 11. Scores are read at the appropriate reward positions and returned to the original candidate order before selection.

B.2 Search procedures and parent-state lifetime

In beam search, the continuations of a surviving path form a parent group. Groups from several survivors can share a pack, and their scores return to global beam selection. DVTS maintains separate subtrees, but siblings within each subtree still share their immediate parent. Groups from different subtrees can be scored together without mixing their parent prefixes or changing local selection.

The shared parent state lasts for the pack that consumes it. At each Transformer block, the verifier computes a group's shared parent prefix once and makes its keys and values available to the child suffixes. That block's state is released after the pack advances. Parent-prefix reuse therefore follows the same layer-major order as relay while memory use remains bounded by the active pack.

B.3 Layer-relayed execution and memory

Verifier weights are held in host memory in model order. Embeddings, final normalization, the reward head, and the selected resident verifier prefix remain on the device. Two buffers hold blocks from the nonresident suffix as they are needed; the verifier applies the same operators to resident and relayed blocks. Weights remain in BF16 throughout.

One copy stream and one compute stream coordinate the buffers. A block executes after its weights arrive, and its buffer can be reused once computation on that block completes. Equation 16 gives the resulting schedule.

When $k > 0$, the first transfer can overlap computation on the resident verifier prefix. Thereafter one buffer supplies the current block while the other receives the next. If no block is resident, scoring waits for the first transfer; a fully resident verifier proceeds without relay. The exposed waits in Equation 18 capture these cases.

As the resident verifier prefix shortens, blocks can be released because their weights remain in host memory. Restoring a longer prefix on a later request has a transfer cost included in $\widehat{L}_{\mathrm{move}}$.

The memory constraint in Equation 4 covers the generator's active state, resident verifier weights, relay buffers, parent and suffix KV state, and activations. Releasing a verifier block gives the next score pack more room without changing the generator's KV allocation.

B.4 Planner inputs

The planner consumes block-level compute, transfer, and peak-memory estimates for the candidate score-pack capacities. These estimates enter the latency and feasibility terms in Equations 5 and 4; the search state supplies the parent groups and parent-prefix lengths that select the appropriate capacity. Plans are chosen at scoring boundaries, while the resident verifier prefix remains fixed or moves toward fewer resident blocks within a request.

C EXECUTION INVARIANTS

The serving path preserves three properties while changing the layout of the computation: parent-complete packing must return the same verifier scores as flat scoring, relay must apply the same sequence of model operators as a resident verifier, and the planner must select only feasible layouts. The following statements make these properties explicit.

C.1 SCORE EQUIVALENCE UNDER PARENT PACKING

Let h_p be the verifier-tokenized parent prefix for parent p, and let u_i be the private suffix of child i. The full input of the child is $x_i = h_p \parallel u_i$. A parent-complete pack stores h_p once and applies a causal mask that allows a child suffix to attend to h_p and to its own preceding suffix tokens, but not to tokens from another child or parent group.

Score equivalence. For any pack that contains complete parent groups, the reward score returned for every child is identical to the score obtained by evaluating the same children independently.

Proof. Consider the hidden state at each Transformer block. At the embedding layer, the state for every token is the embedding of the same verifier token in both executions. Assume the states agree through block $\ell - 1$. The mask in the packed execution exposes exactly the shared parent-prefix and causal suffix positions that are visible to the corresponding child in the independent execution. Thus the attention inputs, keys, values, and attention output agree for every child position. The feed-forward and normalization operators act pointwise on those same states, so the block-ℓ outputs agree as well. Induction over the blocks gives identical hidden states at the reward positions, and the reward head therefore returns the same scores.

Only the verifier tokenization and mask enter the argument; the generator's representation of the paths does not. It also explains why groups from different parents can share a pack without sharing computation: their masks keep the parent prefixes disjoint.

C.2 FRAGMENTATION AND CANDIDATE ORDER

When a parent group exceeds the score-pack capacity, the packer emits adjacent fragments in the original child order. Let $\mathcal{G}_p^{(1)}, \ldots, \mathcal{G}_p^{(F_p)}$ be these fragments and let π_p concatenate their candidate indices.

Fragment invariant. Fragmented execution preserves both the score of every child and the order used by the search selection step.

Proof. Each fragment uses the same shared parent prefix h_p and the same within-group causal mask as a complete group. The preceding score-equivalence argument therefore applies to each fragment independently. The packer records the original candidate index with every suffix; applying π_p to the fragment outputs reconstructs the original order. Since selection consumes this ordered score vector, splitting a group changes only the number of times its prefix is evaluated, not the scores or their association with candidates.

This invariant separates two effects that are often conflated. Fragmentation adds parent-prefix work and may add a scan of the nonresident suffix, but it does not change the search transition itself. The planner can therefore account for fragmentation in its latency objective while preserving the same score vector.

C.3 EQUIVALENCE OF LAYER RELAY

Let $G_\ell(\cdot; W_\ell)$ denote the complete operator for verifier block ℓ, including its normalization and attention state update. A relayed block uses the same operator and weights as the resident execution; only the time at which W_ℓ becomes available changes.

Relay equivalence. If a relayed block is executed only after its weight copy completes, layer relay produces the same verifier state and reward scores as a fully resident execution.

Proof. The input to block 1 is the same in both executions. Suppose the state entering block ℓ agrees. The copy stream supplies an exact copy of W_ℓ before the compute stream invokes G_ℓ, and the preceding block has already completed. Therefore both executions apply the same operator to the same state and produce the same output. The induction continues through the resident verifier prefix and the relayed suffix. The two-buffer recurrence in Equation 16 enforces the readiness condition and prevents a buffer from being overwritten before the block using it completes, so overlap changes only timing.

The same reasoning covers a fully resident verifier and a zero-block resident verifier prefix. In the former, no copy is exposed; in the latter, the first block waits for its weights. Both are endpoints of the same ordered execution.

C.4 Feasibility and planner selection

For a frontier at round t, let $\mathcal{F}_t$ be the finite set of (k, b) pairs satisfying Equation 4, and define

$$C_t(k, b) = \sum_{j \in \mathcal{J}_t(b)} \widehat{L}_{V,j}(k, b) + \widehat{L}_{\text{move}}(k_{t-1}, k). \tag{24}$$

Planner property. Algorithm 2 returns a feasible pair (k_t^*, b_t^*) whose cost is no larger than that of any other pair in $\mathcal{F}_t$.

Proof. The outer loop visits every profiled capacity and the inner loop visits every admissible resident boundary. A pair enters the comparison exactly when it satisfies Equation 4; hence no infeasible layout can become the incumbent. The incumbent is replaced only when the newly evaluated cost is smaller. After the final pair, it is therefore feasible and minimizes C_t over $\mathcal{F}_t$.

The property is local to a scoring boundary. The next frontier changes $\mathcal{J}_t(b)$ and the memory terms, so the same finite comparison can select a different layout without changing the score semantics established above. The move term carries the cost of changing the resident boundary between consecutive rounds, while the packing and relay invariants keep the resulting execution equivalent to the verifier computation represented by the search.

D Boundary State and Execution Semantics

The search and serving layers meet at a scoring boundary. The search layer owns the live paths and their order; the serving layer turns that state into score packs and a memory layout. Making this boundary explicit separates search semantics from the layout choices made by serving.

D.1 Canonical frontier state

Let $\mathcal{P}_t$ be the ordered parent list at round t, and let $\mathcal{G}_t = (\mathcal{G}_p)_{p \in \mathcal{P}_t}$ be the ordered child groups. For each parent, h_p is the verifier-tokenized parent prefix and $d_p = |h_p|$. For child $i \in \mathcal{G}_p$, let u_i be its private suffix and $s_i = |u_i|$. The state exchanged at the boundary is

$$S_t = \big(\mathcal{P}_t, \mathcal{G}_t, (d_p)_{p \in \mathcal{P}_t}, (s_i)_i, k_{t-1}, \pi_t\big), \tag{25}$$

where k_{t-1} is the resident boundary used by the preceding round and π_t is the candidate order consumed by selection. The selected plan derives the corresponding layout and stream schedule from this state.

The verifier input template induces a canonicalization map Γ from search paths to token sequences. For a path x_i with parent p,

$$\Gamma(x_i) = h_p \parallel u_i, \qquad \mathcal{D}_t = \Gamma(\mathcal{P}_t, \mathcal{G}_t) = \{(p, h_p, \{u_i : i \in \mathcal{G}_p\})\}. \tag{26}$$

All shared parent-prefix lengths used by the packer are computed after Γ, so sharing follows the verifier's tokenization. Equal verifier-tokenized parent prefixes remain shareable even if the generator reached them through different internal states.

The order in $\mathcal{P}_t$ and in each $\mathcal{G}_p$ is the order in which the search exposes candidates. It is carried through packing by storing the original candidate index with each suffix. This gives the serving layer a stable key for returning scores, even when several groups share one pack or one group is split across fragments.

State sufficiency. For a fixed verifier, profiled capacity set, and profile table, S_t is sufficient to determine the score-pack layout and the admissible resident boundaries. In particular, two boundary states with the same ordered groups, parent-prefix and suffix lengths, previous boundary, and candidate order produce the same layout and the same planner objective.

Proof. Equation 26 fixes the work of every parent group. Algorithm 1 then scans these groups in their recorded order and uses only their work and the score-pack capacity to emit complete groups or adjacent fragments. The feasibility test in Equation 4 uses the previous boundary, the selected capacity, and the corresponding profile entries. Finally, Equation 5 sums the resulting pack costs and the boundary-move cost. Each operation is deterministic once the fields of S_t are fixed, so the emitted layout and objective coincide.

D.2 Boundary transitions

It is useful to view one search round as a composition of four maps. The generator extension E_t appends candidate suffixes to the live parents; canonicalization and packing form $P_t = \mathrm{Pack}_b \circ \Gamma$; verifier execution V_k scores the packs under resident boundary k; and selection R_t keeps the paths required by the search procedure. The transition can therefore be written as

$$S_{t+1} = R_t\big(V_{k_t^*}(P_t(E_t(S_t)))\big), \qquad (k_t^*, b_t^*) = \mathrm{Plan}(S_t). \tag{27}$$

The planner changes the order in which weights and score packs are processed while preserving the arguments supplied to R_t. This distinction is important for search procedures with different selection rules. Beam search consumes one global ordered score vector; DVTS consumes scores within each subtree before applying its diversity rule. Both receive the same candidate–score association because π_t is part of the boundary state.

Transition invariance. Replacing flat scoring by parent-complete packing and fully resident execution by layer relay leaves the search transition in Equation 27 unchanged.

Proof. For a complete group, score equivalence follows from the causal-mask argument in Section C.1. A fragmented group applies the same argument to each adjacent fragment, and π_t restores the original order by Section C.2. Layer relay applies the same block operator and weights in model order, so Section C.3 gives the same score for every candidate. Hence $V_{k_t^*}(P_t(E_t(S_t)))$ is the same ordered score map as flat, fully resident execution. Applying the same selection map R_t yields the same next frontier.

This formulation also clarifies why planner decisions can be made at scoring boundaries. The plan may change between rounds, but each round exposes the same canonical score map to the search. A change in k_t or b_t therefore changes execution cost while leaving the search transition fixed.

D.3 Memory accounting for an active pack

The feasibility expression in the main text can be decomposed into components that are fixed for a request and components that scale with the selected layout. For a state S_t, boundary k, and score-pack capacity b, write

$$\begin{aligned} M_t(k, b) = M_{\mathrm{fixed}}(S_t) + M_V^{\mathrm{res}}(k) + 2M_{\mathrm{relay}} \\ + M_V^{\mathrm{KV}}(S_t, b) + M_V^{\mathrm{act}}(S_t, b). \end{aligned} \tag{28}$$

The two relay buffers appear explicitly because one holds the current nonresident block while the other receives the next. The fixed term includes the generator state, verifier endpoints, and runtime allocations that coexist with every score pack. Parent and suffix states enter the KV and activation terms through the largest pack emitted for S_t.

For a fixed frontier and resident boundary, both $M_V^{\mathrm{KV}}(S_t, b)$ and $M_V^{\mathrm{act}}(S_t, b)$ are nondecreasing in b: a larger pack contains at least the states of every smaller pack and may contain additional parent groups or suffixes. Similarly, $M_V^{\mathrm{res}}(k)$ is nondecreasing in k because the resident verifier prefix contains one more block. These monotonicities make the feasible set an ordered region even though the latency objective is not monotone.

Feasibility monotonicity. If (k, b) is infeasible, every pair (k', b') with $k' \geq k$ and $b' \geq b$ is infeasible. If (k, b) is feasible, any pair (k', b') with $k' \leq k$ and $b' \leq b$ is feasible, provided the fixed request footprint is unchanged.

Proof. Under the stated ordering, the resident-weight term and the pack-dependent KV and activation terms cannot decrease when either coordinate increases. Equation 28 therefore cannot cross from infeasible to feasible when both coordinates are increased. The reverse ordering gives the second statement.

The latency need not follow this order. Increasing k shortens every relayed suffix, but increasing k can also reduce the largest feasible b, increase the number of packs, and repeat parent or weight work. The planner consequently searches the feasible boundary rather than assuming that the most resident layout is best. Equation 28 supplies the pruning relation; Equations 2 and 3 supply the cost comparison.

D.4 Tie handling and state continuity

Profiled capacities can occasionally give the same predicted cost. A fixed order over capacities and resident boundaries resolves such ties without changing the objective. Let $\prec$ denote this order and define

$$(k_t^*, b_t^*) = \min_{\prec} \arg\min_{(k,b)\in\mathcal{F}_t} C_t(k, b). \tag{29}$$

The tie rule only chooses a representative among equal-cost layouts.

The previous boundary k_{t-1} is retained in the state only to price movement between consecutive scoring rounds. If the next plan keeps the same resident verifier prefix, the move term is zero. If it releases blocks, the release is immediate because their copies remain in host memory; if it restores a longer prefix, the corresponding weights are transferred before the next round. Thus the state transition carries exactly the information needed to account for temporal locality without coupling unrelated requests.

Monotone residency within a request. When the admissible resident boundaries are restricted to the current resident verifier prefix and shorter prefixes, the boundary can only stay fixed or move toward a shorter prefix during one request. Restoring a longer prefix is a new-request transition and is charged by $\widehat{L}_{\text{move}}$.

Proof. At each scoring boundary, the planner considers the current resident verifier prefix and its release candidates. Releasing a block removes its device allocation while leaving its host copy available for later requests. The allowed transition relation is therefore $k_t \leq k_{t-1}$. A fresh request initializes its own k_0 and pays the transfer needed to establish it.

This continuity property keeps the move term local to the request while allowing the planner to react to a changing frontier. It also makes the execution trace interpretable: a resident-boundary change marks a real change in the memory value of the next score pass, and score ordering alone cannot trigger it.

D.5 Cumulative boundary cost

For a request with scoring rounds $1, \ldots, T$, the boundary component of the predicted verifier time is

$$\widehat{L}^{1:T}_{\text{boundary}} = \sum_{t=1}^{T} \widehat{L}_{\text{move}}(k_{t-1}, k_t), \qquad k_0 \text{ given by the request state.} \tag{30}$$

The term is shared by all packs in the next round. If the resident verifier prefix is unchanged across rounds it is zero; otherwise it charges the ordered boundary difference before the first pack that needs the changed blocks. Per-pack costs then account only for parent-prefix reuse, fragmentation, and overlap within that round. This separates persistence across rounds from computation inside a pack while keeping both effects in the planner objective.