SIFTKV: VERIFIER STATE REUSE ACROSS SEARCH ROUNDS FOR EFFICIENT TEST-TIME SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Verifier-guided test-time scaling improves reasoning by expanding several candidate paths and retaining the most promising ones for further search. The selected paths return as the starting points of the next round, so the same reasoning histories continue across rounds. However, conventional process reward model (PRM) serving scores every extension as a new, complete request. It repeatedly reconstructs the histories of surviving paths and materializes KV for candidates that selection immediately removes. As the tree grows, an increasing share of verifier work repeats past computation instead of advancing the search. We introduce SiftKV, a PRM serving system that carries verifier state with the selected paths across rounds. Each surviving path resumes from a token-consistent continuation that preserves its causal KV and accumulated step rewards. New suffix state remains provisional until selection, after which only the selected suffixes join a tree that stores shared ancestry once. Verification can then extend the surviving paths without reconstructing their histories. Across reasoning workloads, search policies, and generator–verifier pairs, SiftKV delivers up to 50% lower end-to-end latency and 69% lower verifier time, together with up to 104% higher Precise Goodput.

1 INTRODUCTION

Difficult reasoning problems often benefit from additional inference compute (Snell et al., 2025; Wu et al., 2025). Verifier-guided search makes that budget selective. A generator extends partial solutions, a PRM scores the new reasoning steps, and a search policy keeps the children most worth pursuing (Park et al., 2025; Chen et al., 2026; Khalifa et al., 2026). After reduction, the surviving children become parents for the next expansion, and repeated cycles grow a reasoning tree. Promising paths receive several rounds of computation, while weak branches stop early. Unlike independent sampling, this search retains path identity and ancestry from one round to the next. Compute can then follow the paths that remain promising instead of repeatedly starting new solutions.

However, the verifier does not see this continuing tree. PRM serving receives each extension as a complete request, so a child that survives one round returns with its previously scored history attached again (Kuang et al., 2026). Figure 1 shows this replay despite the path identity retained by search. The verifier also writes KV for every candidate before the next frontier is known. As paths lengthen, replay grows while most candidate KV has no next-round use (Figure 2). Prefix caches recover matching tokens (Gim et al., 2024; Liu et al., 2025), but token overlap cannot reveal which scored candidates the search will keep. Preserving search continuity across PRM requests introduces two challenges for verifier execution when selected paths return in later rounds.

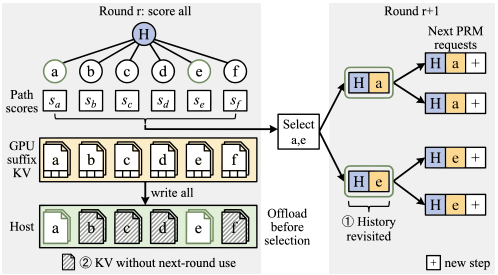


Figure 1: Search continues selected paths, but request-oriented PRM serving rebuilds their histories and writes KV for candidates that selection discards.

The first challenge is to resume a returning path as the same PRM scoring computation. The verifier must recover more than a matching KV prefix, which captures only causal activations. The path

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.

score also depends on rewards from earlier steps and the marker positions at which they were read (Lightman et al., 2024; Wang et al., 2023). Appending a step can also retokenize the parent–child join. If the tokens, KV, and accumulated rewards stop at different positions, the resumed computation no longer represents the path selected by the search. A reusable continuation must align all three at the same token boundary and evaluate only the changed suffix.

The second challenge is to retain state only for paths that selection keeps. The verifier creates suffix state while every candidate is still in contention, but only reduction reveals which continuations will return (Chen et al., 2026). Discarding every suffix forces the selected paths through another verifier pass. Retaining every suffix spends memory and transfer bandwidth on branches that end. The surviving paths also share ancestors, so storing and moving their shared history separately repeats the same state (Juravsky et al., 2024). The verifier must defer state commitment until the reducer identifies the next frontier, then preserve the surviving continuations without replicating the ancestry shared across sibling paths in that frontier.

We present SiftKV, a PRM serving system that makes the selected frontier the handoff between search and verification. For every live path, a continuation binds canonical tokens to the causal KV and accumulated reward state needed by its next extension. Candidate suffixes remain provisional while they compete. Reduction promotes only the survivors into a tree where descendants share their parent segments. Later verifier calls resume from these continuations instead of reconstructing prior reasoning, leaving more of the test-time budget for new work. On AIME, MATH-500, and OlympiadBench Math, reuse removes up to 69% of verifier time. The saved work lowers end-to-end latency by 50% and increases Precise Goodput by 104%.

The main contributions are:

- We identify the selected frontier as the state-lifetime boundary missing from request-oriented PRM serving. It reveals which verifier states will return and which can be released.
- We develop a PRM continuation that resumes selected paths without replay while keeping canonical tokens, causal KV, and accumulated step rewards on the same prefix.
- We design a frontier-guided tree executor that holds suffix state provisionally through reduction, commits only surviving paths, and stores their shared ancestry once.
- Our evaluation measures peak reductions of 50% in end-to-end latency and 69% in verifier time, with Precise Goodput increasing by 104%.

2 Background and Motivation

2.1 Verifier-Guided Scaling Revisits a Search Tree

Verifier-guided scaling alternates between expansion and selection. Best-of-N spends test-time compute on independent completions (Wang et al., 2022), while search-based methods pause between reasoning steps and use intermediate scores to decide where the next expansion should go (Snell et al., 2025; Khalifa et al., 2026). Beam Search ranks candidates globally, DVTS preserves diversity across root partitions, Dynamic Branching changes the number of children, and VG-Search changes the distance between verifier calls (Chen et al., 2025; Hooper et al., 2025; Chen et al., 2026). Despite these differences, every round produces a frontier of selected children, and that frontier becomes the parent set for the following expansion.

This recurrence gives selection a second role. A rejected child's verifier state has reached the end of its lifetime, while a selected child's state becomes context for the next expansion. Selected siblings may also share much of their earlier path. The search controller keeps this identity and ancestry explicit so that it can extend the same paths in the following round.

2.2 Request-Oriented Verification Flattens the Tree

The verifier interface flattens this tree into complete sequences. The PRM scores each new step in the context of the problem and earlier reasoning, reading rewards at designated markers along the path (Lightman et al., 2024; Wang et al., 2023). When an extension arrives, already scored steps and markers reappear beside the new text (Kuang et al., 2026). Siblings also repeat their common parent.

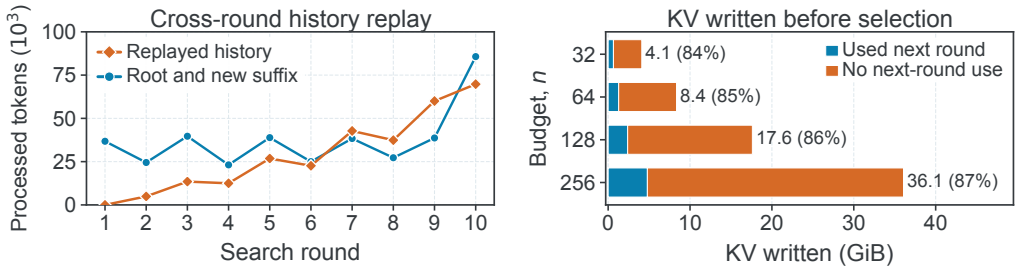


Figure 2: Selection reveals the lifetime of verifier state. Replayed history grows as selected paths return, while most candidate KV written before selection has no next-round use.

The request contains the candidate's tokens, but it carries neither the path identity nor the ancestry links maintained by the search controller.

Flattening the tree wastes work in two ways. With successive returns, request-oriented verification reprocesses a history that approaches the size of the new suffix, while 84–87% of candidate KV never becomes next-round context (Figure 2). The request interface lacks the search state needed to distinguish future context from transient work.

2.3 Selection supplies the missing boundary

Existing serving techniques reduce the cost of the current request or batch. Prefix caches recover matching token blocks from a memory hierarchy (Gim et al., 2024; Gao et al., 2025; Liu et al., 2025), while batching and faster attention reduce the cost of each verification call (Dao et al., 2022; Dao, 2024; Ye et al., 2025). Their reuse decisions do not incorporate whether a scored candidate will return. The search reducer already makes that decision when it forms the next frontier.

Candidate suffixes have no durable owner while candidates compete. Reduction changes that status: state attached to rejected candidates expires, whereas state attached to the new frontier becomes input to another verifier call. Parent links among those paths expose shared ancestry. The reducer can thus govern verifier-state lifetime at the same point where it advances the search.

3 SiftKV design

SiftKV carries this selection decision into PRM execution. Each selected path continues as a persistent PRM computation, while newly scored suffixes remain provisional until reduction. Figure 3 shows how parent continuations supply context for child scoring and how selected suffixes enter a shared tree. The frontier now directs both future generation and the verifier state kept for it.

3.1 PRM path continuations

For each active path p, SiftKV keeps the verifier state needed to continue that path:

$$S_p = (T_p, V_p, L_p; \nu_p), \tag{1}$$

where T_p is the canonical PRM token sequence, V_p references its causal KV segments, L_p records marker positions and accumulated reward state, and ν_p identifies the verifier, tokenizer, template, and positional convention. A child adds one linked segment; the inherited state remains shared.

The continuation fields move together because they must agree on the same reusable prefix. Reusing KV past a changed token boundary would resume the wrong causal sequence, while retaining reward entries past that boundary would attach scores to markers that no longer occupy the same positions.

When child c extends parent p, the verifier tokenizes the complete canonical input because tokens can change at the join. Let b_c be the first reward-marker position not preserved by the parent ledger, with $b_c = \infty$ when all markers in the matching prefix agree. The reusable boundary is

$$\ell_{\text{tok}} = \text{LCP}(T_p, T_c), \qquad \ell_c = \begin{cases} \min(\ell_{\text{tok}}, b_c), & \nu_p = \nu_c, \\ 0, & \nu_p \neq \nu_c. \end{cases} \tag{2}$$

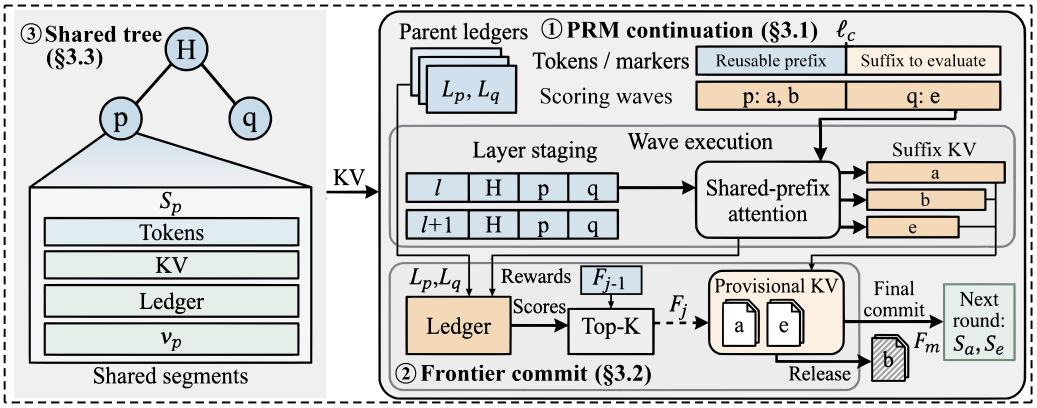


Figure 3: Overview of SiftKV. Parent continuations supply shared KV and accumulated rewards for scoring, while only selected child suffixes join the tree for the next round.

The longest common token prefix captures retokenization at the join, while a moved marker pulls the boundary back to b_c. KV rows and ledger entries before ℓ_c remain, and the verifier evaluates $T_c[\ell_c :]$; causal KV and accumulated rewards resume together.

The verifier evaluates the changed suffix and advances the reward ledger over its new markers. For an associative aggregation state q,

$$q_{j+t} = q_j \odot \phi(\rho_{j+1}) \odot \cdots \odot \phi(\rho_{j+t}), \qquad s = J(q_{j+t}). \tag{3}$$

Here ρ_i is the reward at marker i, ϕ maps it into the aggregation state, and J reads the path score. The parent continuation supplies q_j, so the verifier evaluates only new markers while the reducer still receives a complete-path score. For a mean reward, q_j holds the sum and count of completed steps; extending the path updates both without rereading earlier rewards. A shared token trie evaluates common rows once within a scoring batch; the new suffix state stays with its candidate until selection.

Because the continuation records token and reward coordinates rather than a batch location, a path can move between scoring waves without changing its state. Execution can therefore adapt to available capacity while the search continues to reason over complete paths.

3.2 Frontier-guided state commitment

The candidates of one search round need not fit in a single scoring batch. SiftKV groups siblings and packs them into waves, retaining only the candidates that can still enter the next frontier. Let $C_r^1, \ldots, C_r^m$ be the waves in round r, and let R_K be a stable top-K reducer. After each wave, SiftKV updates the live set:

$$F_0 = \varnothing, \qquad F_j = R_K(F_{j-1} \cup C_r^j). \tag{4}$$

The identity $R_K(R_K(A) \cup B) = R_K(A \cup B)$ makes the updates equivalent to one reduction over all candidates. A candidate that leaves F_j cannot return to the final frontier, so its provisional state can be released after its last device use. Partitioned reducers use the same update within each partition.

Each score enters reduction when its wave completes, while suffix state remains provisional as later waves compete. Membership in F_m commits a continuation to the next round. If $\mathrm{path}(c)$ denotes the segments from the root to candidate c, the persistent state is

$$\mathcal{S}_{r+1} = \{S_c : c \in F_m\}, \qquad \mathcal{T}_{r+1} = \bigcup_{c \in F_m} \mathrm{path}(c). \tag{5}$$

Segments outside $\mathcal{T}_{r+1}$ have no descendant in the next expansion and are reclaimed. Reduction thus turns state produced during scoring into the continuation tree used by the next round.

3.3 Tree-structured state sharing

A flat collection of continuations replicates the shared history of surviving paths. SiftKV stores the tokens and per-layer KV added along each search edge with a parent link, so siblings point to the

same ancestor segments. For a selected frontier F, define $\mathcal{T}(F) = \bigcup_{p \in F} \mathrm{path}(p)$, let $|u|$ denote the token count of segment u, and write $|\,\mathrm{path}(p)|$ for the total token count along a path. The tree occupies

$$M_{\mathrm{tree}}(F) = m_{\mathrm{KV}} \sum_{u \in \mathcal{T}(F)} |u| \le m_{\mathrm{KV}} \sum_{p \in F} |\,\mathrm{path}(p)| = M_{\mathrm{flat}}(F), \tag{6}$$

where m_{KV} is the all-layer KV footprint per token. Shared edges make the inequality strict.

The tree also reduces ancestry movement. A flat layout restores the parent history for every child, whereas SiftKV restores the union requested by a sibling group. The executor stages that union one layer at a time, evaluates suffixes with shared-prefix causal attention, and keeps new KV in a paged arena while the next restore is prefetched.

Parent-path ordering keeps nearby branches together. For parent set P, let $U(P)$ denote the union of their ancestry segments and let $|U(P)|$ count its tokens; let b_p be the number of children of parent p and m_l the per-token KV size at layer l. The host-to-device ancestry volume satisfies

$$V_{\mathrm{tree},l} = m_l |U(P)| \le m_l \sum_{p \in P} b_p |\,\mathrm{path}(p)| = V_{\mathrm{flat},l}. \tag{7}$$

The executor uses FlashInfer's shared-prefix attention path (Ye et al., 2025). Model weights remain resident while one ancestry layer is staged and the arena holds the live suffix set. Each candidate carries its parent-continuation handle through reduction, and the handles in $\mathcal{S}_{r+1}$ accompany the selected texts into the next expansion.

4 Experiments

4.1 Experimental setup

Testbed. We build on vLLM and use FlashInfer for attention execution (Kwon et al., 2023; Ye et al., 2025), with BF16 on an NVIDIA GeForce RTX 4090 D (24 GB).

Workloads and models. We evaluate AIME 2024 and 2025, MATH-500, and OlympiadBench Math (Mathematical Association of America, 2024; Hendrycks et al., 2021; He et al., 2024). We use Qwen2.5-Math-1.5B-Instruct with Math-Shepherd-Mistral-7B-PRM by default, then pair the 1.5B generator with Skywork-o1-Open-PRM-1.5B or Qwen2.5-Math-7B-Instruct with Math-Shepherd to vary the generator–verifier balance (Yang et al., 2024; Wang et al., 2023; Skywork-o1 Team, 2024).

Baselines. We compare SiftKV with **FastTTS** (Chen et al., 2026), **vLLM-TTS** (Kwon et al., 2023), **FlatPRM**, and **LMCache** (Liu et al., 2025). FastTTS combines generator scheduling with prefix-aware optimizations, whereas vLLM-TTS provides standard verifier-guided serving. FlatPRM adds incremental PRM execution and a reward ledger but stores each continuation independently; LMCache provides general-purpose KV reuse and offload. We use **Full Prefill** as the complete-request reference in the verifier data-path study.

Metrics. End-to-end latency includes generation and verification; verifier time isolates PRM execution. We also report Precise Goodput, measured as effective completion tokens per beam divided by beam completion time (Chen et al., 2026).

4.2 End-to-end scaling

Across workloads, the methods are closest before history accumulates and separate as larger budgets carry longer selected paths into later rounds. The widening gap ties verifier cost to the inherited context that later rounds repeatedly revisit. Figure 4 follows this transition across the full budget sweep. Its latency curves combine generation with the cost of revisiting selected history, while Precise Goodput shows how much useful completion work reaches the end of each beam. At small budgets, inherited state is limited and the methods remain close; as later rounds carry more context, replay becomes visible in both metrics. This common sweep makes the cost of carrying verifier state visible before we separate the effects of search policy and model balance.

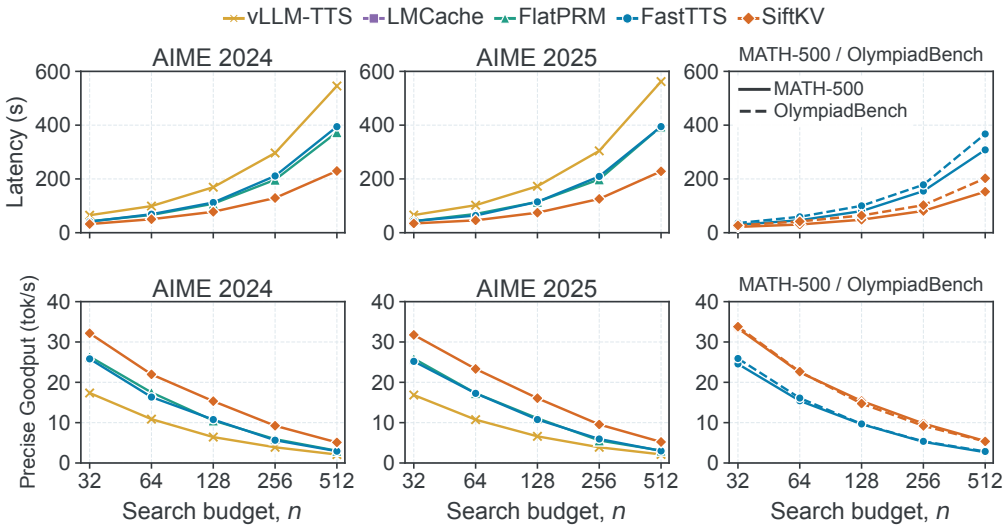


Figure 4: End-to-end scaling across four reasoning workloads. SiftKV sustains lower latency (top) and higher Precise Goodput (bottom) as the search budget grows.

Table 1: Verifier time across the budget sweep (s). Each entry is FastTTS / SiftKV; the last column gives their speedup at the largest budget.

Workload	$n = 32$	$n = 64$	$n = 128$	$n = 256$	$n = 512$	Speedup
AIME 2024	16.5 / 9.7	34.6 / 19.3	64.4 / 33.7	133.5 / 54.1	271.4 / 107.8	**2.52**×
AIME 2025	17.5 / 10.8	33.9 / 18.8	70.4 / 33.4	140.0 / 57.4	275.6 / 109.3	**2.52**×
MATH-500	13.7 / 6.9	25.8 / 12.2	50.7 / 21.7	108.1 / 36.9	225.0 / 70.7	**3.18**×
OlympiadBench	16.2 / 8.9	30.9 / 16.5	61.6 / 28.8	123.0 / 48.7	260.3 / 97.2	**2.68**×

Table 2: PRM-ranked pass@k across search budgets on AIME 2024 and AIME 2025 (%). Each block reports the fraction of problems with a correct solution among the top-k ranked candidates.

	AIME 2024: budget n					AIME 2025: budget n				
k	32	64	128	256	512	32	64	128	256	512
1	10.3	10.3	10.3	10.3	6.9	14.8	11.1	7.4	14.8	14.8
4	10.3	17.2	17.2	13.8	10.3	22.2	11.1	11.1	18.5	14.8
16	13.8	20.7	20.7	20.7	13.8	22.2	11.1	14.8	18.5	18.5
32	13.8	24.1	24.1	20.7	13.8	22.2	11.1	14.8	18.5	18.5
64	–	27.6	24.1	24.1	13.8	–	11.1	14.8	18.5	25.9

At the largest budget in Figure 4, SiftKV reduces end-to-end latency by 50% and raises Precise Goodput by 104% on MATH-500. FastTTS still returns a selected history as a complete request, FlatPRM keeps separate continuations for paths with shared ancestry, and LMCache recovers repeated blocks without knowing which scored candidates will return. SiftKV receives that decision directly from reduction, so a larger search creates more reusable verifier state instead of proportionally more repeated prefill work on returning paths.

Verifier time grows much more slowly under SiftKV (Table 1), reaching a 2.52–3.18× speedup over FastTTS at the largest budget. The largest gain appears on MATH-500, whose longer reasoning paths carry more scored history into later rounds. The same relationship across workloads ties the speedup to the amount of inherited context available for reuse.

Because reuse does not alter candidate ranking, FastTTS and SiftKV have identical PRM-ranked pass@k values in Table 2. Their difference is the verifier work required to produce those rankings.

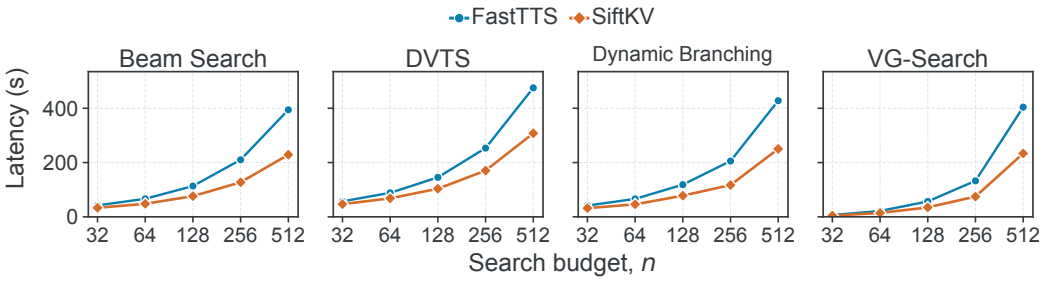


Figure 5: Across four search policies with different expansion and selection rules, SiftKV lowers end-to-end latency throughout the budget sweep.

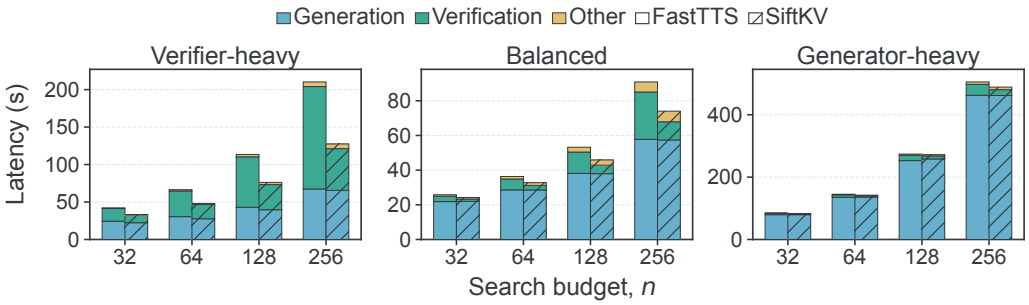


Figure 6: End-to-end latency decomposed into generation, verification, and other work across three generator–verifier pairs. The verifier-heavy pair benefits most because verification accounts for a larger share of total runtime.

4.3 Generality across search policies and generator–verifier pairs

Across Beam Search, DVTS, Dynamic Branching, and VG-Search, SiftKV reduces end-to-end latency by 35–42%, showing that the gain does not depend on one search policy.

The curves in Figure 5 separate as the budget grows even though the policies rank, partition, and revisit candidates differently. Each policy nevertheless forms a selected frontier at the end of every round. Because SiftKV acts at that frontier, the same state-retention rule applies across all four search trees without policy-specific changes to verifier execution.

How much of this reduction reaches end-to-end latency depends on the fraction of each round spent in verification (Figure 6). The verifier-heavy pair reaches 1.27–1.65× speedup, and the balanced pair follows the same trend as later rounds carry more history into verification. With the larger generator, generation dominates wall time and the same verifier reduction has less influence on total latency. The differences among generator–verifier pairs follow the time each round spends rescoring history.

4.4 Mechanism analysis

Search depth and beam width expose different opportunities for reuse: depth accumulates history across rounds, while width creates siblings with shared ancestry. Figure 7 connects these trends to the end-to-end contribution of each mechanism by removing one mechanism at a time.

No single component accounts for the full gain. Removing any one raises end-to-end latency by 57–138%. The largest penalties come from retaining every suffix or recomputing survivors, because both undo the value of selection at a different point: one keeps state with no future, while the other discards state that will immediately be needed again. Removing the reward ledger restores score reconstruction along accepted histories, and removing ancestry sharing reloads the same parent for each child. Even without ancestry sharing, cross-round continuation remains 1.9× faster than Full Prefill, so the two forms of reuse are complementary.

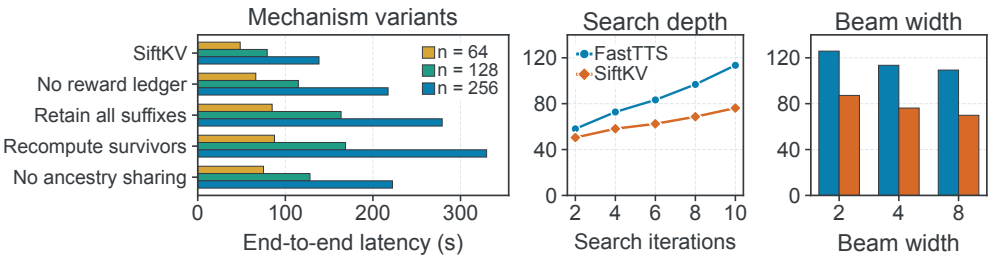


Figure 7: Ablations and search geometry. Each removed mechanism raises latency; deeper searches and wider beams provide more reusable history or sibling ancestry.

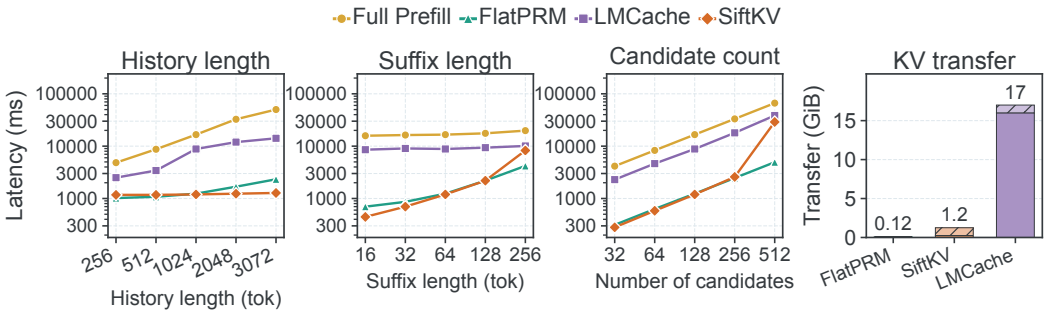


Figure 8: Verifier latency as history length, suffix length, and candidate count vary, with KV transfer at right. Reuse matters most when inherited histories and candidate sets grow.

Table 3: Verifier time as the selected frontier grows (s). Parentheses give the frontier size, and speedup compares SiftKV with Full Prefill.

Selected frontier	Full Prefill	FlatPRM	LMCache	SiftKV	Speedup
1/8 (16 paths)	16.60	1.24	8.98	1.13	**14.7**×
1/4 (32 paths)	16.60	1.24	8.83	1.20	**13.9**×
1/2 (64 paths)	16.58	1.24	9.50	1.44	**11.5**×

Deeper searches widen the gap over FastTTS from 1.15× to 1.49× as histories grow, while wider beams yield 1.44–1.56× speedup by sharing each restored parent across more siblings.

4.5 Verifier state and capacity

Verifier work changes along three dimensions: inherited history, new suffix length, and candidate count. Figure 8 separates these effects and reports their corresponding KV movement.

The history sweep separates the methods most sharply. Full Prefill rises with every inherited token, whereas SiftKV stays nearly flat because those tokens already belong to the continuation. FlatPRM also avoids full replay but represents shared ancestry separately, while LMCache restores repeated blocks through a general cache hierarchy. A larger candidate count further favors shared parents; longer suffixes narrow the gap because every method must evaluate the new tokens. The three sweeps thus follow the two sources of work that SiftKV removes: cross-round replay and repeated ancestry.

KV transfer confirms the same distinction. At the long-history point in Figure 8, SiftKV transfers 1.2 GiB, compared with 17 GiB for LMCache, because it stages the ancestry union rather than separate cached paths. FlatPRM moves only suffix payloads but retains a distinct state object for every active continuation in the frontier.

As the selected frontier grows in Table 3, Full Prefill and FlatPRM barely change because their execution is independent of selection. SiftKV commits more suffix state, yet remains 11.5–14.7× faster than Full Prefill. The small increase in SiftKV time separates retention overhead from the verifier work performed during the current scoring round.

Table 4: Capacity sensitivity of SiftKV. KV arena and attention workspace changes leave latency nearly stable, whereas wider scoring waves avoid repeated ancestry staging.

Resource	Capacity	End-to-end latency (s)		Verifier time (s)	
		Budget 128	Budget 512	Budget 128	Budget 512
KV arena	256 pages	76.2	227.2	33.7	109.0
	384 pages	76.1	226.9	33.7	109.3
	512 pages	76.2	228.9	33.6	108.5
Scoring wave	1,024 tokens	77.4	227.8	35.1	109.3
	2,048 tokens	76.4	227.1	34.1	109.1
	4,096 tokens	76.2	228.9	33.6	108.5
Attention workspace	64 MB	76.2	228.9	33.6	108.5
	128 MB	76.0	226.8	33.7	108.5

Once the live verifier state fits, enlarging the KV arena or attention workspace in Table 4 leaves inherited-history cost nearly unchanged. Scoring-wave capacity matters more because splitting a sibling group stages its ancestry again; end-to-end latency moves less because these capacity changes do not affect generation time in the same round.

5 Related Work

Test-time search and process verification. Test-time scaling uses sampling or search (Wei et al., 2022; Kojima et al., 2022; DeepSeek-AI, 2025; Wang et al., 2022; Snell et al., 2025; Wu et al., 2025). Best-of-N samples independently, while tree search directs expansion with intermediate results (Feng et al., 2023; Yao et al., 2023; Besta et al., 2024), often scored by process reward models (Cobbe et al., 2021; Uesato et al., 2022; Lightman et al., 2024; Khalifa et al., 2026). KV-PRM avoids text re-encoding by scoring from generator KV (Kuang et al., 2026). SiftKV instead maintains a separate PRM's scoring context across rounds and attaches it to the selected frontier.

Inference serving and speculative execution. Speculative execution parallelizes verification with draft tokens, tree proposals, or lookahead (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2024; Cai et al., 2024a; Fu et al., 2024). Serving systems optimize scheduling, batching, memory, and phase placement (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024; Juravsky et al., 2024; Wu et al., 2023; Agrawal et al., 2024; Zhong et al., 2024; Patel et al., 2024; Prabhu et al., 2025). SpecEdge and FastTTS apply these ideas to reasoning; HeraSys reuses computation across workflow nodes (Park et al., 2025; Chen et al., 2026; Li et al., 2026). PRM-guided search revisits selected branches after generation, and SiftKV preserves their verifier state across that interval.

Model and KV state under memory pressure. Quantization and offload reduce or relocate model state (Dettmers et al., 2022; Frantar et al., 2023; Lin et al., 2024b; Kim et al., 2024; Xiao et al., 2023; Zhao et al., 2024; Sheng et al., 2023; Du et al., 2025; Song et al., 2024; Alizadeh et al., 2024). KV systems compress histories, distribute state, restore blocks on demand, or retrieve them from slower memory (Xiao et al., 2024; Zhang et al., 2023; Liu et al., 2023; Li et al., 2024; Cai et al., 2024b; Jin et al., 2025; Liu et al., 2025; Lin et al., 2024a; Qin et al., 2026; Yao et al., 2026; Liu et al., 2026; Ramachandran et al., 2026; Woo et al., 2026). These systems infer reuse from token overlap, token importance, or cache policy. Search reduction supplies a different signal by revealing which histories will return, allowing SiftKV to preserve their step rewards with causal KV.

6 Conclusion

Verifier-guided search carries paths, scores, and ancestry forward, but request-oriented PRM serving discards this continuity between calls. SiftKV gives selected paths persistent verifier continuations and organizes their causal KV and reward state as a shared tree. Reduction decides which provisional suffixes become durable, allowing verification to advance the search without repeatedly rebuilding its past. At the system level, these verifier savings shorten the search and sustain higher Precise Goodput. Selection thus governs both where reasoning continues and which verifier state remains worth carrying into later search rounds. The selected frontier therefore links search progress to the verifier state worth carrying forward.

AI USE STATEMENT

Generative AI tools supported code generation and review, mathematical derivations and proof development, literature search, and suggestions for improving the manuscript's clarity. The authors reviewed and verified all AI-assisted material and accept responsibility for the final manuscript, code, and the results reported in this paper.

ETHICS STATEMENT

This work studies LLM serving infrastructure using publicly available models and benchmarks. It involves neither human participants nor 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 S. 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*, 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*, 2024.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 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*, 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, Guanxi Lu, Yasuyuki Okoshi, Zhiwen Mo, Masato Motomura, and Hongxiang Fan. Rethinking optimal verification granularity for compute-efficient test-time scaling. *Advances in Neural Information Processing Systems*, 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*, 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.

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

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022.

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.

Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. AlphaZero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of LLM inference using lookahead decoding. In *Proceedings of the 41st International Conference on Machine Learning*, 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*, 2025.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, 2024.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. OlympiadBench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 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, 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.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems*, 2025.

Jordan Juravsky, Bradley Brown, Ryan Saul Ehrlich, Daniel Y Fu, Christopher Re, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024.

Muhammad Khalifa, Rishabh Agarwal, Lajanugen Logeswaran, Jaekyeom Kim, Hao Peng, Moontae Lee, Honglak Lee, and Lu Wang. Process reward models that think. *Transactions on Machine Learning Research*, 2026.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. SqueezeLLM: Dense-and-sparse quantization. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Takeshi Kojima, Shixiang Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, 2022.

Peng Kuang, Haibo Jin, Xiaoyu Han, Yanli Wang, Xiaopeng Yuan, Ye Yu, Kaidi Xu, and Haohan Wang. KV-PRM: Efficient process reward modeling via KV-cache transfer for multi-agent test-time scaling. *arXiv preprint arXiv:2607.09153*, 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*, 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*, 2023.

Size Li, Zhiqing Tang, Hongrui Liang, Jianxiong Guo, Jiong Lou, Tian Wang, and Weijia Jia. HeraSys: Collaborative serving of multiple LLM workflows via fine-grained end-to-end optimization. In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2026.

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*, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison 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.

Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. Infinite-LLM: Efficient LLM service for long context with DistAttention and distributed KVCache. *arXiv preprint arXiv:2401.02669*, 2024a.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, 2024b.

Guangda Liu, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Danning Ke, Minyi Guo, and Jieru Zhao. FreeKV: Boosting KV cache retrieval for efficient LLM inference. In *International Conference on Learning Representations*, 2026.

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*, 2025.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems*, 2023.

Mathematical Association of America. American invitational mathematics examination, 2024.

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*, 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, Íñigo 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.

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 Trans. Storage*, 2026.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. ThinKV: Thought-adaptive KV cache compression for efficient reasoning models. In *International Conference on Learning Representations*, 2026.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, 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*, 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 30th ACM 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.

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*, 2023.

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. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Sunghyeon Woo, Jaeeun Kil, Hoseung Kim, Minsub Kim, Joonghoon Kim, Ahreum Seo, Sungjae Lee, Minjung Jo, Jiwon Ryu, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. ICaRus: Identical cache reuse for efficient multi model inference. *arXiv preprint arXiv:2603.13281*, 2026.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

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. In *International Conference on Learning Representations*, 2025.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

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.

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.

Jiayi Yao, Samuel Shen, Kuntai Du, Shaoting Feng, Dongjoo Seo, Rui Zhang, Yuyang Huang, Yuhan Liu, Shan Lu, and Junchen Jiang. Vericache: Turning lossy kv cache into lossless llm inference. *arXiv preprint arXiv:2605.17613*, 2026.

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*, 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*, 2022.

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

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate LLM serving. In *Proceedings of Machine Learning and Systems*, 2024.

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.

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.

A VERIFIER-STATE ANALYSIS

Once a path enters the selected frontier, its verifier state becomes part of the next search round. Continuing that path requires a consistent token and reward boundary, while reduction determines which newly produced state remains live.

A.1 NOTATION AND CONTINUATION BOUNDARY

Let p denote a path that has survived a search round and let c be one of its children. The verifier template maps each path to a canonical token sequence T_p. In the continuation S_p from Equation 1, V_p references the causal KV rows $\mathcal{KV}_p$, and $L_p = (M_p, q_p)$ contains the ordered reward-marker positions M_p and current aggregation state q_p. The convention ν_p identifies the verifier, tokenizer, template, and positional rules under which these objects were produced.

Retokenization can move the join between parent and child. Equation 2 finds the longest common token prefix between the parent and complete child inputs, then pulls the boundary back to the first marker that diverges from the parent ledger. A token merge shortens ℓ_{tok}; a moving marker sets b_c inside that token prefix. KV rows before ℓ_c remain causal history, marker positions before ℓ_c retain their rewards, and the reward ledger resumes from the last unchanged marker. The child continuation links the new segment to its parent rather than flattening both into a new allocation.

A.2 CONTINUATION SEMANTICS

Let f_θ be a decoder-only PRM with L causal layers. For token sequence x, write $h_i^{(l)}(x)$ for the hidden state at position i after layer l, and $K_i^{(l)}(x), V_i^{(l)}(x)$ for its projected key and value.

Prefix invariance. For any suffix z and every prefix position $i < |x|$,

$$h_i^{(l)}(x \parallel z) = h_i^{(l)}(x), \qquad (K_i^{(l)}, V_i^{(l)})(x \parallel z) = (K_i^{(l)}, V_i^{(l)})(x). \tag{8}$$

At the embedding layer, tokens and positions in x are unchanged. At a later layer, position i attends only to positions no greater than i, all of which have identical states by induction. The feed-forward block is position-wise, so the equality continues through the layer; stored rows of x can serve as causal history for queries in z.

Applying the result at the effective continuation boundary gives the reusable interval

$$I(p, c) = \{0, \ldots, \ell_c - 1\}, \tag{9}$$

with $I(p, c) = \varnothing$ when $\ell_c = 0$. Equation 2 gives this case when $\nu_p \neq \nu_c$. The PRM head at every unchanged marker receives the same hidden state, so its previously emitted reward remains part of the score for the continued path.

Score continuation. Suppose the search score is represented by an associative state update. If q_j is the ledger state after marker j, then new rewards extend it as

$$q_{j+t} = q_j \odot \phi(\rho_{j+1}) \odot \cdots \odot \phi(\rho_{j+t}), \qquad s = J(q_{j+t}). \tag{10}$$

Sum, product, minimum, maximum, and mean with an accompanying count fit this form. The verifier reads q_j from the parent, evaluates the markers in the changed interval, and writes one child state. Previous reward values need not be recomputed.

A.3 WORK SAVED BY SHARED PATHS

Consider the candidate set $\mathcal{C}_r$ evaluated in round r. For candidate c, write $t_c = |T_c|$, let ℓ_c be the effective continuation boundary after token and marker refinement, and set $\delta_c = t_c - \ell_c$. Thus δ_c includes a short rollback when retokenization moves the continuation boundary into the end of the parent. A verifier that treats each candidate as a request charges its full sequence:

$$W_{\text{request}} = L \sum_r \sum_{c \in \mathcal{C}_r} t_c. \tag{11}$$

During scoring, SiftKV evaluates each batch with a shared token trie. Let U_r be the sum of unique token rows over the batches in round r, and let π_c indicate that the scorer captured the suffix payload needed to commit survivor c without another transformer pass. This row-wise work is

$$W_{\text{score}} = L \sum_r U_r, \qquad W_{\text{mat}} = L \sum_r \sum_{c \in \mathcal{A}_r} (1 - \pi_c)\delta_c, \tag{12}$$

where $\mathcal{A}_r$ is the selected frontier. W_{mat} accounts for survivor waves that materialize suffix KV after reduction; it is zero when the captured payload is committed directly. At the token-row level, the corresponding SiftKV work is

$$W_{\text{SiftKV}} = W_{\text{score}} + W_{\text{mat}}. \tag{13}$$

The shared execution plan retains the candidate-specific attention context while projecting common token rows once. The resulting row saving is

$$W_{\text{request}} - W_{\text{score}} = L \sum_r \left(\sum_{c \in \mathcal{C}_r} t_c - U_r \right), \tag{14}$$

Sharing prefixes removes the repeated row work inside each scoring batch. Attention context remains candidate-specific, and the continuation boundary decides which suffix payloads are materialized and retained for the selected frontier after reduction.

The effect accumulates with search depth. Suppose a branch begins with h_0 tokens, adds δ tokens per round, and evaluates b children in each of D rounds. For a single scoring batch, the unique-row count in round r is $U_r = h_0 + r\delta + b\delta$, whereas independent requests execute $b(h_0 + r\delta + \delta)$ rows. The repeated-history component grows with depth even when the new suffix remains fixed. If captured survivor payloads are committed without a second pass, the one-round accounting is

$$W_{\text{score}} = L \sum_{r=0}^{D-1} \big(h_0 + r\delta + b\delta\big), \qquad W_{\text{request}} = Lb \sum_{r=0}^{D-1} \big(h_0 + r\delta + \delta\big). \tag{15}$$

Their difference is $L \sum_r (b-1)(h_0 + r\delta)$, before accounting for any survivor materialization pass. The expression isolates the depth effect: each later verification point adds little new text but repeats a history that has grown over preceding rounds.

A.4 Equivalence of layer-streamed execution

For one child c of parent p, let $Q_c^{(l)}$ be the suffix queries at layer l. Independent causal attention uses the logical key and value sequences

$$K_{p \to c}^{(l)} = K_p^{(l)} \parallel K_c^{(l)}, \qquad V_{p \to c}^{(l)} = V_p^{(l)} \parallel V_c^{(l)}. \tag{16}$$

The layer-streamed executor supplies the same sequences through two physical regions: the restored parent segment and the child's arena pages. Shared-prefix attention evaluates

$$H_c^{(l)} = \text{Attn}\left(Q_c^{(l)}, K_{p \to c}^{(l)}, V_{p \to c}^{(l)}; \mathcal{M}_c\right), \tag{17}$$

where $\mathcal{M}_c$ is the causal mask for the child. Siblings with the same requested prefix share the storage of $K_p^{(l)}, V_p^{(l)}$ but retain separate suffix pages and sequence metadata, so Equation 17 has the same operands and token order as an independent execution.

At layer one, both paths begin from the same suffix embeddings and parent KV. If their suffix states agree at layer l, Equation 17, the residual update, and the position-wise feed-forward block produce the same states at layer $l + 1$. Induction through all L layers gives the same reward-head inputs at every new marker. Layer streaming changes where ancestry is stored and how often it is transferred. The logical causal sequence seen by a child is unchanged.

A.5 Wave-wise frontier reduction

Give every candidate a stable ordering key consisting of its score and identifier. For a finite set A, let $R_K(A)$ return its first K candidates under that order. The top-K operator is mergeable:

$$R_K\big(R_K(A) \cup B\big) = R_K(A \cup B). \tag{18}$$

The identity follows because an item removed from the inner reduction already has K higher-ranked elements in A; adding B cannot make it enter the final top-K set.

For scoring waves $C_r^1, \ldots, C_r^m$, repeated application of Equation 18 gives

$$F_j = R_K\left(\bigcup_{i=1}^{j} C_r^i\right), \qquad F_m = R_K(C_r). \tag{19}$$

After wave j, F_j contains every candidate that can still enter the next round. Under the stable ordering, a candidate dropped from this set remains behind at least K retained candidates even after later waves arrive. Its suffix pages may therefore return to the arena after their last device event.

Partitioned reducers. DVTS assigns a fixed quota to each root partition. If partition g retains K_g candidates, SiftKV maintains

$$F_j^{(g)} = R_{K_g}\left(F_{j-1}^{(g)} \cup C_{r,j}^{(g)}\right). \tag{20}$$

Each partition satisfies Equation 19; their union is the final partitioned frontier. Global reducers use the unpartitioned form after producing their candidate sets.

The identifier in the ordering key preserves tied-reward order across wave boundaries.

A.6 Residency and ancestry transfer

Assume a wave contains at most W candidates, the frontier retains K, and a suffix has at most $\Delta_{\max}$ tokens. Using the per-token KV footprint m_{KV} from Equation 6, the logical live-payload footprint under retain-frontier execution is bounded by

$$M_{\mathrm{device}} \le M_{\mathrm{fixed}} + 2M_{\mathrm{stage}} + (W + K)\Delta_{\max} m_{\mathrm{KV}} + M_{\mathrm{meta}}. \tag{21}$$

Double-buffered staging hides the next ancestry restore behind current-layer computation. The all-layer term is a footprint bound for the current wave and the frontier inherited from earlier waves; the streamed executor realizes that state layer by layer. A displaced suffix leaves this term once its final device event has completed.

For payload-backed continuations, the host tree $\mathcal{T}_r$ occupies

$$M_{\mathrm{host}}(r) = m_{\mathrm{KV}} \sum_{v \in \mathcal{T}_r} |\operatorname{seg}(v)| + M_{\mathrm{ledger}}(r). \tag{22}$$

Each edge segment appears once, even when several active paths share it. Reference counts retire a segment after its last descendant leaves the search.

Reachability and reclamation. A committed segment remains reachable while an active continuation, a provisional child, or an in-flight copy references it or one of its descendants. Let r_u count these live references. The invariant

$$r_u > 0 \quad \Longleftrightarrow \quad u \text{ lies on a path to live verifier state} \tag{23}$$

is maintained as handles are committed and released. When a count reaches zero, the corresponding subtree can be reclaimed immediately.

For parent group P, write b_p for the number of children under parent p and $U(P)$ for the union of requested ancestry segments; $|U(P)|$ is its token count. A flattened layout restores the complete requested parent path from host memory once per child:

$$V_{\mathrm{flat},l} = m_l \sum_{p \in P} b_p \sum_{u \in \mathrm{path}(p)} |u|. \tag{24}$$

The tree layout transfers each segment in the union from host memory once at layer l:

$$V_{\mathrm{tree},l} = m_l \sum_{u \in U(P)} |u|, \qquad V_{\mathrm{flat},l} - V_{\mathrm{tree},l} = m_l \sum_{u \in U(P)} (\eta_u - 1)|u| \ge 0, \tag{25}$$

where η_u counts the children that read segment u. Sibling expansion raises η_u along the parent path; unrelated parents make most multiplicities one.

Layer-local staging overlaps ancestry restoration for layer $l+1$ with suffix computation at layer l. If those operations take d_l and c_l, respectively, a wave satisfies

$$T_{\text{wave}} \le d_1 + \sum_{l=1}^{L-1} \max(c_l, d_{l+1}) + c_L + T_{\text{head}} + T_{\text{commit}}. \tag{26}$$

Only $\max(0, d_{l+1} - c_l)$ remains exposed between adjacent layers. Larger sibling groups reuse the restored ancestry across more suffix queries.

A.7 From verifier reuse to end-to-end latency

Let a FastTTS search spend G, V, and O time in generation, verification, and the remaining search runtime. Suppose SiftKV accelerates verification by $a = V/V'$ and adds runtime cost E for continuation management. The end-to-end speedup is

$$S_{\text{e2e}} = \frac{G+V+O}{G+V/a+O+E} = \frac{1}{1-f_V(1-1/a)+\epsilon}, \tag{27}$$

where $f_V = V/(G+V+O)$ and $\epsilon = E/(G+V+O)$. Increasing history reuse raises a, while the fraction of time originally spent in verification determines how much of that gain reaches the end-to-end runtime of the search.

The expression separates the verifier gain from the fraction of wall time available to improve. A verifier-heavy pair has a large f_V, so a widening verifier gap produces a growing end-to-end separation. With a larger generator, f_V is smaller: verifier time still falls, but the denominator remains dominated by G. Both f_V and a can change as histories lengthen, so the relation applies throughout the entire evaluated range of search budgets.

The continuation path improves total time whenever

$$E < V\left(1 - \frac{1}{a}\right). \tag{28}$$

The right-hand side is the verifier work removed by reuse. This form links the end-to-end benefit to the history preserved by continuation and to the cost of parent grouping and frontier release.

A.8 Ordering and wave boundaries

The layer staging buffer keeps the common prefix when execution moves from one parent to the next. For an ordering $\pi = (p_1, \ldots, p_s)$ of the parents in a wave, let a_j be the lowest common ancestor of p_{j-1} and p_j. The ancestry loaded at layer l is

$$V_l(\pi) = m_l \left(|\operatorname{path}(p_1)| + \sum_{j=2}^{s} |\operatorname{path}(p_j) \setminus \operatorname{path}(a_j)| \right). \tag{29}$$

The union of requested ancestry is a lower bound on the layer-local restore volume. An ordering that keeps each subtree contiguous can approach that bound when the whole union fits in one staging cycle:

$$V_l(\pi) \ge m_l |U(P)|. \tag{30}$$

The planner uses lexicographic parent-path ordering and capacity chunks, which preserves this locality when the parent group fits in a wave.

Wave boundaries may require the same ancestor in several staging cycles. If the parent groups are partitioned into waves $P_1, \ldots, P_m$, the total ancestry volume becomes

$$V_{\text{waves},l} = m_l \sum_{j=1}^{m} |U(P_j)| = m_l \sum_{u \in U(P)} \omega_u |u|, \tag{31}$$

where ω_u is the number of waves containing a descendant that reads segment u. The bounds

$$m_l |U(P)| \le V_{\text{waves},l} \le m_l \sum_{p \in P} |\operatorname{path}(p)| \tag{32}$$

separate the effects of tree sharing and wave capacity. Keeping sibling groups intact makes their parent path contribute to one union, while parent-path packing keeps nearby parents and their early ancestors in the same wave.

The corresponding arena requirement depends on suffix state rather than restored ancestry. If B is the KV block size and wave j contains suffix lengths $\{\delta_c : c \in C_r^j\}$, its provisional page count is

$$\Pi_j = \sum_{c \in C_r^j} \left\lceil \frac{\delta_c}{B} \right\rceil, \qquad P_{\text{live}} \le \max_j \left(\Pi_j + \sum_{c \in F_{j-1}} \left\lceil \frac{\delta_c}{B} \right\rceil \right). \tag{33}$$

Together with Equation 21, this page bound assigns the first term to wave capacity and the second to frontier reduction across successive scoring waves.

B Search-wide accounting

Summing the one-round terms over surviving edges separates candidate computation, shared ancestry movement, and suffix state that ends at reduction.

B.1 Trie bounds for variable path lengths

For variable-length paths, the terms in Equation 12 use U_r as the sum of unique row counts across the scoring batches in round r. Let $N_r = |\mathcal{C}_r|$ and let $\bar{t}_r$ be the mean candidate length. The trie overlap satisfies

$$\max_{c \in \mathcal{C}_r} t_c \le U_r \le \sum_{c \in \mathcal{C}_r} t_c = N_r \bar{t}_r. \tag{34}$$

The bounds describe the two extremes: a common prefix gives the lower end, while disjoint rows give the upper end. As histories lengthen, request-oriented scoring repeats more rows even when the newly generated suffix is unchanged.

Retokenization at a path boundary is included in δ_c. If a merge moves the boundary backward by a_c tokens and the generated step contains d_c tokens, then $\delta_c = d_c + a_c$; the same quantity controls suffix materialization and committed state.

B.2 State committed by reduction

Let $\mathcal{C}_r$ contain the candidates scored in round r, and let $\mathcal{A}_r = R_{K_r}(\mathcal{C}_r)$ be the paths selected for the next round. Candidate c adds δ_c KV rows. Writing every candidate suffix to durable storage would move

$$D^{\text{out}}_{\text{eager}} = m_{\text{KV}} \sum_r \sum_{c \in \mathcal{C}_r} \delta_c \tag{35}$$

bytes from the verifier arena. The selected frontier commits

$$D^{\text{out}}_{\text{frontier}} = m_{\text{KV}} \sum_r \sum_{c \in \mathcal{A}_r} \delta_c. \tag{36}$$

Both paths score every candidate. Their difference concerns the state produced as a by-product of that score:

$$D^{\text{out}}_{\text{eager}} - D^{\text{out}}_{\text{frontier}} = m_{\text{KV}} \sum_r \sum_{c \in \mathcal{C}_r \setminus \mathcal{A}_r} \delta_c. \tag{37}$$

The right-hand side is the suffix state whose path ends at reduction.

Define the token-weighted survivor fraction

$$\rho_r = \frac{\sum_{c \in \mathcal{A}_r} \delta_c}{\sum_{c \in \mathcal{C}_r} \delta_c}. \tag{38}$$

The round writes a ρ_r fraction of the suffix KV produced during scoring. When suffix lengths are equal, $\rho_r = K_r/|\mathcal{C}_r|$; with variable steps, it reflects the actual state volume selected by the reducer.

A larger survivor fraction increases committed continuation state, while request prefill remains unchanged throughout the current round.

The same boundary controls host allocation. Provisional pages belong to the current decision, whereas committed edge segments belong to the next search state. Membership in $\mathcal{A}_r$ directly determines which continuation segments remain allocated for the next round.

B.3 Persistent tree growth

After round r, let $\mathcal{T}(\mathcal{A}_r)$ be the minimal subtree connecting the root to the active paths. In a payload-backed representation, each edge e stores a segment of δ_e tokens. The tree occupies

$$M_{\text{tree}}(r) = m_{\text{KV}} \sum_{e \in \mathcal{T}(\mathcal{A}_r)} \delta_e. \tag{39}$$

If every active path were flattened independently, edge e would appear once for each active descendant. With multiplicity

$$\mu_e(r) = \left|\{p \in \mathcal{A}_r : e \in \text{path}(p)\}\right|, \tag{40}$$

the corresponding storage is

$$M_{\text{paths}}(r) = m_{\text{KV}} \sum_{e \in \mathcal{T}(\mathcal{A}_r)} \mu_e(r)\delta_e. \tag{41}$$

The storage removed by the tree is

$$M_{\text{paths}}(r) - M_{\text{tree}}(r) = m_{\text{KV}} \sum_{e \in \mathcal{T}(\mathcal{A}_r)} \big(\mu_e(r) - 1\big)\delta_e. \tag{42}$$

Shared ancestry contributes in proportion to both its length and the number of active descendants that continue to share the same segment.

For $K_r = |\mathcal{A}_r|$,

$$1 \le \frac{M_{\text{paths}}(r)}{M_{\text{tree}}(r)} \le K_r. \tag{43}$$

The lower endpoint describes unrelated paths. The upper endpoint is approached when most stored tokens lie on a common trunk. More concretely, if all active paths share H_r tokens, then

$$M_{\text{paths}}(r) - M_{\text{tree}}(r) \ge m_{\text{KV}}(K_r - 1)H_r. \tag{44}$$

The tree changes through selected edges and reclaimed branches. Let $\mathcal{E}_r^+$ be the edge segments committed at round r and $\mathcal{E}_r^-$ the segments that lose their last active descendant. Its storage follows

$$M_{\text{tree}}(r) = M_{\text{tree}}(r-1) + m_{\text{KV}} \sum_{e \in \mathcal{E}_r^+} \delta_e - m_{\text{KV}} \sum_{e \in \mathcal{E}_r^-} \delta_e. \tag{45}$$

Reference counts realize this recurrence locally. A segment enters with a selected child and leaves when no active path reaches it, so the stored tree remains the ancestry needed by the current frontier.

B.4 Ancestry traffic over all waves

Under the coalesced staging path, let $P_{r,j}$ be the parent groups in wave j of round r, let $b_{r,p,j}$ count the children of parent p, and let $|U(P_{r,j})|$ denote the token count in their ancestry union. The flattened and shared ancestry volumes are

$$D_{\text{flat}}^{\text{in}} = \sum_r \sum_j \sum_l m_l \sum_{p \in P_{r,j}} b_{r,p,j} |\,\text{path}(p)|, \tag{46}$$

$$D_{\text{tree}}^{\text{in}} = \sum_r \sum_j \sum_l m_l |U(P_{r,j})|. \tag{47}$$

The first term repeats a parent path for each child; the second restores the requested ancestry union once per wave. Sibling groups reduce inbound movement, while the survivor term in Equation 36 controls what becomes durable after reduction.

B.5 A COST DECOMPOSITION FOR VERIFIER SAVINGS

A single cost model separates compute, ancestry movement, and state commitment. Let α be an effective scoring cost per row, including its attention context, β the exposed time per byte of ancestry movement, and γ the exposed time per byte of durable suffix write. Let E collect frontier maintenance, tree metadata, and any transfer not hidden by layer overlap. A request-oriented execution has the approximate cost

$$T_{\text{request}} \simeq T_{\text{common}} + \alpha W_{\text{request}} + \beta D^{\text{in}}_{\text{flat}} + \gamma D^{\text{out}}_{\text{eager}}, \tag{48}$$

where T_{common} contains reward-head work and search control shared by the two paths. For SiftKV,

$$T_{\text{SiftKV}} \simeq T_{\text{common}} + \alpha(W_{\text{score}} + W_{\text{mat}}) + \beta D^{\text{in}}_{\text{tree}} + \gamma D^{\text{out}}_{\text{frontier}} + E. \tag{49}$$

Subtracting the two expressions yields

$$\begin{aligned} T_{\text{request}} - T_{\text{SiftKV}} \simeq\ & \underbrace{\alpha(W_{\text{request}} - W_{\text{score}})}_{\text{shared candidate rows}} + \underbrace{\beta\big(D^{\text{in}}_{\text{flat}} - D^{\text{in}}_{\text{tree}}\big)}_{\text{shared ancestry}} \\ & + \underbrace{\gamma\big(D^{\text{out}}_{\text{eager}} - D^{\text{out}}_{\text{frontier}}\big)}_{\text{discarded suffix state}} - \underbrace{\alpha W_{\text{mat}} + E}_{\text{survivor materialization and management}}. \end{aligned} \tag{50}$$

The first term is the work removed by shared-trie scoring. The second replaces per-child ancestry restores with tree unions, and the third avoids durable writes for paths that end at reduction. Materialization and bookkeeping follow the state that remains useful.

Search geometry determines the balance among these terms. Depth increases overlap between candidate paths and enlarges the first term; more siblings increase ancestry multiplicity and the second. A smaller survivor fraction enlarges the third term, while a larger survivor set can increase W_{mat}. Equation 27 then maps the resulting verifier saving to end-to-end latency.

For a single parent with history h, b equal-length children, and k selected suffixes of length δ, let u_p be the number of unique token rows in their candidate trie and let m_p be the suffix rows re-executed for selected survivors. The same decomposition reduces to

$$\Delta T_p \simeq \alpha[b(h+\delta) - u_p - m_p] + \beta(b-1)m_{\text{KV}}h + \gamma(b-k)m_{\text{KV}}\delta - E_p. \tag{51}$$

For siblings with disjoint suffixes, $u_p = h + b\delta$; if their captured payloads are retained, $m_p = 0$. The expression exposes the contributions of depth, width, reduction, and survivor materialization for a single parent group in the search tree.

B.6 THE WAVE-CAPACITY TRADE-OFF

Wave capacity controls both temporary residency and repeated ancestry movement. Consider two adjacent waves with parent sets P_a and P_b. If capacity allows them to merge, their layer-l ancestry volume changes from

$$D_{\text{split},l} = m_l\big(|U(P_a)| + |U(P_b)|\big) \quad \text{to} \quad D_{\text{merge},l} = m_l|U(P_a) \cup U(P_b)|. \tag{52}$$

By inclusion–exclusion,

$$D_{\text{split},l} - D_{\text{merge},l} = m_l|U(P_a) \cap U(P_b)|. \tag{53}$$

Merging saves the ancestry shared across the boundary. A boundary between siblings repeats their complete parent path; a boundary between distant branches repeats their earlier common trunk.

There is a corresponding arena cost. Holding the incoming frontier F fixed and writing C_a, C_b for the disjoint candidate suffix sets, split execution requires

$$P_{\text{split}} = P(F) + \max\{P(C_a), P(C_b)\}, \tag{54}$$

whereas the merged wave requires

$$P_{\text{merge}} = P(F) + P(C_a) + P(C_b). \tag{55}$$

The additional peak is $\min\{P(C_a), P(C_b)\}$. Reduction between split waves can lower the incoming frontier further, so Equation 54 is the direct fixed-frontier comparison.

Equations 53 and 55 state the packing trade-off: larger waves remove repeated ancestry at the price of more provisional pages and a larger attention plan. Keeping siblings together captures the largest overlap first. Once those groups fit, additional capacity affects execution weakly.

B.7 REWARD-LEDGER GROWTH

The reward ledger follows the continuation edges and stores marker-level state. Let r_e be the number of reward markers introduced on edge e, m_r the bytes stored per marker record, and m_q the fixed aggregation state attached to a committed node. For a logical ledger representation,

$$M_{\text{ledger}}(r) = m_r \sum_{e \in \mathcal{T}(\mathcal{A}_r)} r_e + m_q |\mathcal{T}(\mathcal{A}_r)|. \tag{56}$$

If marker density satisfies $r_e \leq \kappa \delta_e$ and every committed edge contains at least $\delta_{\min}$ tokens, then

$$\frac{M_{\text{ledger}}(r)}{M_{\text{tree}}(r)} \leq \frac{\kappa m_r}{m_{\text{KV}}} + \frac{m_q}{m_{\text{KV}} \delta_{\min}}. \tag{57}$$

For a PRM, one compact score record accompanies a reasoning step, while each token contributes keys and values at every layer. The ledger follows the continuation edge but remains a separate structure from the causal KV payload.

The ledger travels with each continuation handle. Shared tree edges remove duplicated causal KV, while the ledger supplies the aggregate score state of the selected path.

C VERIFIER-STATE REALIZATION

The search controller already maintains parent and survivor relations. Continuation handles carry verifier state along those relations, with reward markers normalized for the PRM in use.

C.1 SEARCH-STATE INTEGRATION

A search begins with a verifier prefill that creates the root continuation. Thereafter, each path carries a handle to its parent continuation together with the text consumed by the search algorithm. Expansion produces child texts linked to that handle. The verifier scores the children as one logical set and returns their rewards with child continuation handles. Reduction keeps the handles of the selected children as parents for the next round; branches that leave the frontier release their references to the continuation tree maintained by the search.

The interface accepts either a global retention budget or fixed quotas for root partitions. Wave-level reduction follows the strategy's selection order and returns the complete selected frontier after all scoring waves for that round have completed.

C.2 PRM CONTINUATION STATE

The two PRMs expose step rewards differently: Math-Shepherd uses a step tag, while Skywork uses blank-line boundaries and the final token of each completed step. Both map to a continuation with canonical tokens, marker positions and rewards, aggregation state, and the verifier/tokenizer/template convention that gives those coordinates meaning.

For every child, tokenization against its parent identifies the reusable causal and score prefixes. An unfinished step may move its marker while leaving tokens unchanged; execution resumes at that marker and truncates the ledger to the preceding completed step, preserving earlier rewards.

The continuation also keeps a logical marker cursor. Rewards computed beyond the step currently visible to the search remain attached to the continuation; a later expansion can advance the cursor and reuse those stored values.

C.3 TREE-STRUCTURED EXECUTION

Each scoring batch is lowered to a trie over its canonical tokens. Every unique trie node contributes one row to the shared row-wise path; candidate-specific attention contexts are gathered from those rows, and each candidate retains its own causal path and reward-marker lookup. After reduction, selected payloads are linked directly when available or materialized over the requested ancestry union. The executor pipelines ancestry restoration with layer computation, then links paged suffix state into the persistent child continuations used by the next round.