Attention Pays Its Bill

From the Constant-Support Law to Measured Wall-Clock:

A Fused Gather-Decode Kernel and the Regime Where Constant Support Pays

Mohammad Alsufi* **Connor Scott***

and the Brainsless Research Lab AI Systems Research Group

*Equal lead contribution `research@brainsless.com`

Brainsless Research Lab | Technical Report **BRL-2026-07** | June 2026

Abstract

BRL-2026-06 established the Constant-Support Law — a frozen transformer reproduces its dense next-token prediction from a constant number of keys, independent of context length — but reported *no wall-clock speedup*: its reference implementation ran 1.7× slower than FlashAttention, and the fused kernel was left as an open item. This report closes it. We build a fused Triton **gather-decode** kernel (block-aligned gather, GQA-structured, fp32 online softmax, CUDA-graph captured end-to-end) and measure it under pre-registered predictions against the strongest dense baseline we can construct — the fastest probed SDPA backend, sustaining 2.9–3.2 TB/s at $n \geq 128$K (86–96% of the H100's HBM roofline). At the attention-op level the kernel delivers 2.3–10.2× (batch 1, 128K to 1M context, split-k variant) and 3.9–41.9× (batch 8, 32K to 1M, same variant). End to end, sparse decode throughput per sequence is flat from 8K to 128K on H100 and within 7% from 128K to 1M on H200 (dense decays 2.4× over the same span), and a three-parameter memory-traffic model — bandwidth, step overhead, and the measured *price of finding*, $c_1 \approx 1.7$ ms at k=8, 3.3 ms assured — fits the measured H100 ($n \times$ batch) grid at in-sample $R^2 = 0.998$ with held-out error $\leq 2.2\%$, turning the law into an operator's regime map: where constant support pays, and where it honestly does not. Fidelity is measured through the real engine: on natural text at 8–32K, per-step agreement at the k=8 budget is 0.906 unconditional — below the predecessor's strict 0.99 bar. Conditioned on confident steps (dense margin>1), agreement pooled over every non-quarantined ours-policy artifact row that logs it is 1823/1851 = 0.985 at k=8 and 4060/4070 = 0.9975 at the assured k=32 budget (rule and membership in `pooled_agreement.json`); only the assured budget clears the 0.99 bar. On the long-context-trained model (Qwen2.5-7B-Instruct-1M), distant-needle retrieval certifies one-sided 97.5% lower bounds of 0.982 at 32K (497/500, k=8) and 0.981 at 128K (200/200, k=32) — on the looped-paragraph haystack those suites use; a real-prose probe at 32K reads 0.58/0.78/0.96 at k = 8/16/32 (100 trials), so retrieval assurance is haystack-conditional and budget buys it back. A matched-budget baseline panel inside the same engine (§7) separates the policies: a sliding window is fastest and never retrieves; finer page-16 selection retrieves with better per-key fidelity but pays 8× the selector operations at decode in this engine; block-128 is the granularity whose finding cost our kernel makes small. We further measure *findability* — recall at fixed budget decays with context, depends on the key geometry training induced, and collapses by 512K even on the 1M-trained model (10 paired trials) — quantifying for this engine the budget-versus-length frontier observed across sparse-attention methods. Finally, the law's strongest systems consequence: the cache does not need the GPU. With the 60 GB cache in host

RAM and the keep-set fetched over PCIe, a 24 GB A10 sustains 6.1 tok/s of million-token decode — 15× the dense-from-host floor — with the tiered path validated end-to-end at 128K; at 1M we certify throughput, with dense–sparse agreement measured as a consistency check (task retrieval there is beyond the dense reference itself under both raw completion and a chat-template probe; §8). Every number is tied to a named artifact, and the full measurement program is independently re-runnable at documented, per-run-itemized cost (`cost_ledger.json`).

1 From Law to Clock

BRL-2026-06 established the **Constant-Support Law**: a frozen transformer reproduces its dense next-token prediction at a strict fidelity bar (top-1 agreement ≥ 0.99) from a small, model-dependent *constant* number of keys — independent of context length n, verified from 8K to 1M tokens, with the oracle budget $\kappa^\star$ shrinking with model scale to a floor of 16 at 72B [Alsufi and Scott, 2026]. It also reported, verbatim:

> *"We report **no wall-clock speedup**: our PyTorch block-summary forward is actually ∼1.7× slower than the model's fused FlashAttention path at 8K, because it is an unfused reference implementation, not a kernel. [. . .] a fused kernel that converts this into end-to-end acceleration is a systems effort we explicitly leave open."*

This report closes that item. We build the fused kernel — a Triton [Tillet et al., 2019] gather-decode kernel that reads only the constant-support keep-set from the KV cache — integrate it into a real decoding engine for `Qwen2.5-7B-Instruct` [Qwen Team, 2025a] (and its 1M-token variant [Qwen Team, 2025b]), and measure wall-clock against the strongest dense baseline we can construct, under the same honesty rules as the prior report: every number is measured and tied to a named artifact, comparisons are against the *best* dense configuration (never naive PyTorch), and pre-registered predictions are reported whether they held or not. One scope statement up front: the prior report's strict 0.99 top-1 bar was certified for the *oracle* budget; the realizable engine measured here meets it conditionally (on confident steps, and on the needle task at the assured budget) but never unconditionally — so what this paper prices in wall-clock is, strictly, the realizable cousin of the oracle law, and we say so wherever the distinction bites.

The organizing instrument is a **memory-traffic model** (§2), committed before any benchmark ran (an in-repo attestation; see the pre-registration caveat in §13). Decode is memory-bound: a step's latency is, to first order, the bytes it must move. The model prices every regime (n, B) in bytes and converts the Constant-Support Law into falsifiable wall-clock predictions — including the prediction that the prior report's own GATE-3 target ("2–4× at 128K") *must fail* at batch 1, not because the kernel is weak, but because at batch 1 the 15.2 GB of weights dominate the bill. Where constant support pays is a property of the bill, not of the kernel.

Program context for cold readers. *Constant-Support Law* (BRL-2026-06): a frozen transformer reproduces its dense next-token argmax from a constant number of keys. *GATE-3* (the predecessor's runbook): "2–4× decode at 128K, 5–10× at 1M vs the best dense baseline." *Stages* S0–S10 are gated experiment phases (S0 correctness, S1 op-level, S2–S3 end-to-end, S4–S10 revision-cycle additions); *H1–H3* are the pre-registered hypotheses. The *keep-set* is the set of KV-cache positions a sparse decode step actually reads: one sink block, a local window, and k selected distant blocks of 128 tokens. $\kappa^\star$ is BRL-2026-06's oracle budget in *keys* (not blocks). Two realizable budgets recur: k=8 distant blocks (pre-registered) and k=32 (the budget that passes this paper's fidelity gates, "assured").

Contributions. (1) The fused gather-decode kernel (§3): block-aligned, GQA-structured, CUDA-graph-captured, verified to bf16-level numerical tolerance (worst rel. err 2.6×10^{-3}) against masked dense attention on identical keep-sets — both variants; the split-k path's equivalence is in the same committed artifact (`s0_correctness.json`, field `splitk_equivalence`). (2) Measured wall-clock against a near-roofline dense baseline: 10.2×/41.9× (batch 1/8) at the attention op at 1M; context-invariant end-to-end decode (§5–6). (3) The regime map: a three-parameter bill — bandwidth, step overhead, price of finding — fitted per platform, reproducing the H100 ($n \times B$) grid at in-sample $R^2 = 0.998$ ($\leq 2.2\%$ held-out), including the regimes where sparse *loses* (§9). (4) **Findability**, quantified for this engine: recall at fixed budget decays with context, recovers with slowly-growing budget, depends on the model's trained key geometry, and collapses past a measured horizon — a certified, paired-design instance of the budget-versus-length frontier documented across sparse-attention methods [Nawrot et al., 2025], and the program's next open problem (§8). (5) A worked example of pre-registered, gate-staged systems measurement: two stage gates caught a real retrieval failure and a wrong cost assumption before they could contaminate the headline numbers; both are reported, with the fixes, as results.

2 The Memory-Traffic Model (Pre-Registered)

Decoding one token with `Qwen2.5-7B-Instruct` (28 layers, GQA 28/4 heads, head dim 128, bf16) moves, per step:

- **Weights**: $W \approx 15.23$ GB, read once per step regardless of batch size B;
- **Dense KV reads**: $A_d(n) = 57{,}344\,n$ bytes per sequence ($2 \times 28 \times 4 \times 128 \times 2$ B per token) — 7.5 GB at 128K, 60.1 GB at 1M;
- **Sparse KV reads**: $A_s(n) = G + 448\,n$ bytes per sequence, where G is the gathered keep-set (blocks × 128 keys, K and V, all layers) and $448\,n$ is the block-summary scan (1/128 of the dense KV traffic). As pre-registered (top-4 distant, at most 9 blocks): $G \approx 66$ MB — 125 MB at 128K, 536 MB at 1M. The production top-8 configuration (at most 13 blocks; §8.1) gives $G \approx 95$ MB, shifting every speedup prediction by $< 0.5\%$.

The step-time model is

$$t(n, B) \;=\; \frac{W + B \cdot A(n)}{\beta} \;+\; c_0, \tag{1}$$

with effective bandwidth β and a fixed per-step overhead c_0 (kernel launches; small under CUDA graphs). Both free parameters are fitted on the *dense* rows only; the sparse prediction then has no free parameters, and its fit quality is itself a hypothesis (H3). The full pre-registered table (committed before any S1–S3 measurement) is in `preregistration/predictions.json`; headline predictions: batch-1 e2e speedup 1.02× at 8K, 1.45× at 128K (**predicted GATE-3 fail**), 4.6× at 1M; at 128K with batching, 2.8× at B=4 and 4.5× at B=8 — the weights term W/B amortizes, exposing the KV term the kernel eliminates.

3 The Kernel

3.1 Why the reference was slow

BRL-2026-06's reference (`fast_kernel.gather_attend`) was a prefill-shaped, per-head Python loop: it found the right keys but paid for the finding in eager-mode launches and uncoalesced gathers. The lesson is not that gathering is slow; it is that the gather must be *fused* and *block-aligned*.

3.2 Design

We build decode-first — decode is where the law's read-saving is purest (one query, n cached keys) and where the prior gate is stated. One Triton program per (b, h_{kv}):

1. **Select** (PyTorch, $O(n/128)$): mean-pool keys per 128-token block once after prefill (the summaries are never updated during decode — generated tokens stay inside the sliding local window by construction); score the summaries with the kv-group's mean query; keep {block 0 (sink)} ∪ {last 4 local blocks} ∪ {top-k distant blocks} — k=8 in production after the S0b ablation of §8.1, which also replaced this mean scoring with per-head $[k_{\max}, k_{\min}]$ bounds scoring — ≤ 13 blocks = 1,664 keys, emitted as a padded `int32` id list. Distant candidates exclude the sink and local ranges *before* the top-k, so the list is duplicate-free by construction (a duplicate block would double-count its keys in the softmax).
2. **Gather-attend** (Triton, fused): the program loads its kv-group's 7 query rows once into registers, then streams the selected blocks with an fp32 online softmax. Because selection is block-aligned, every K/V tile is a *contiguous* 128 × 128 region of the cache: loads are fully coalesced and the indirection costs one integer load per block. No (n)-long score vector, let alone an ($n \times n$) matrix, ever exists; per-step attention work and memory are $O(1)$ in n.

GQA is exploited structurally: one kv-head's tile is loaded once and shared by its 7 query heads. The whole decode step — embedding, 28 layers with the kernel, LM head, greedy argmax, position bookkeeping — is captured in a single CUDA graph [Gray, 2019] (all state lives in fixed-shape device tensors), so per-step launch overhead is amortized for *both* the sparse path and the dense baseline.

3.3 What stays dense

Prefill is exact dense attention (FlashAttention for full sequences; an exact LSE-merged chunked path beyond a configured threshold — 256K by engine default, raised to 512K in the long-context sessions, so chunking engaged only beyond 512K there), so the KV cache is bit-identical to the baseline's: sparsity exists only in decode-time *reads*, the regime BRL-2026-06 §6 isolated as the one that works (store-time eviction fails the fidelity bar). KV memory is not compressed, and we do not integrate a paged runtime (vLLM) — both stated as limitations (§13).

4 Measurement Methodology

Apples-to-apples. Dense and sparse share everything — weights, KV tensors, prefill, the hand-rolled decode step, CUDA-graph capture — except the attention read. The dense baseline is additionally *probed*: at each configuration we time all eligible SDPA backends (FlashAttention, memory-efficient, cuDNN) at the actual decode shape and the baseline gets the fastest; the chosen backend is recorded in the op-level artifact (cuDNN won every probe; the memory-efficient backend errored at these decode shapes and is recorded as ineligible), and the e2e harness re-probes per configuration. Timing runs are unmasked over the full static cache (eligible for the fastest backends). The cache carries up to 264 zeroed slack slots (generation window plus warmup), which add extra *dense* read traffic — a bias in sparse's favor that we bound rather than ignore: ($264/n$) of dense KV traffic, i.e. 3.2% at 8K, 0.8% at 32K, ≤ 0.21% at $n \geq$ 128K — at least an order of magnitude below every reported speedup contrast. As an external anchor we also report vLLM [Kwon et al., 2023] dense throughput (prefix caching disabled, distinct prompts).

Timing. CUDA events around graph-replay loops; per configuration, 3 repeats per mode interleaved D/S/D/S/D/S within one process (controls thermal/clock drift on rented GPUs); op-level microbenchmarks graph-capture 20 back-to-back calls and report time per call, so launch overhead does not masquerade as op time. One disclosed flattery in that design: across replayed calls the sparse keep-set (3.4 MB per layer at k=8, 9.7 at k=32) can sit resident in the H100's 50 MB L2, while dense reads stream far past it — the op-level sparse latencies are therefore a best case; the end-to-end numbers, where each layer's full forward evicts the last, carry no such advantage and are the deployment-honest measure. Run-to-run session variance (2–4%) is the observed spread between the retired and committed S1 sessions (the retired session predates the quarantine convention and was not kept — the 2–4% figure is therefore attested for that pair) and, auditable, between the k=8/k=32 grid sessions' shared dense cells.

Fidelity. Measured through the *real* decode engine, eager and exactly masked: (i) per-step argmax agreement, with *both* engines teacher-forced on the same true wikitext continuation (64 steps per configuration; §8.2) — the per-step analogue of BRL-2026-06's top-1 bar without cascade artifacts (every fidelity artifact in this report, including the original timing grids, used this forcing; an early code docstring misdescribed it as dense-trajectory forcing and was corrected — the code never had such a path); (ii) free-running exact-prefix length (reported, not gated: a single bf16 argmax flip cascades by construction); (iii) the distant-needle task: a unique 5-digit code at a random distant position, answered greedily by both engines — run first as consistency tiers (60 at 128K, 10 at 512K, 3–5 at 1M) and then, in the internal-review revision stage (S8–S10), powered to the predecessor's standard (500 trials at 32K, 200 at 128K, on the long-context-trained model; §8.4), all with Wilson 95% intervals and paired (discordant-pair) inference where both engines run the same prompts. We do *not* inherit BRL-2026-06's 500-trial certification: our selection granularity differs (per kv-head at decode time vs. per repeated head per query block), so we re-measure.

5 Op-Level Results

Table 1 reports decode-attention latency (graph-captured, time per call) for the gather-decode path — selector *included*; finding is part of our cost — against the fastest dense SDPA backend at each shape (cuDNN won every probe); Figure 1(a) plots the batch-1 medians. Three facts structure the table (artifact `s1_oplat.json`, H100 80GB):

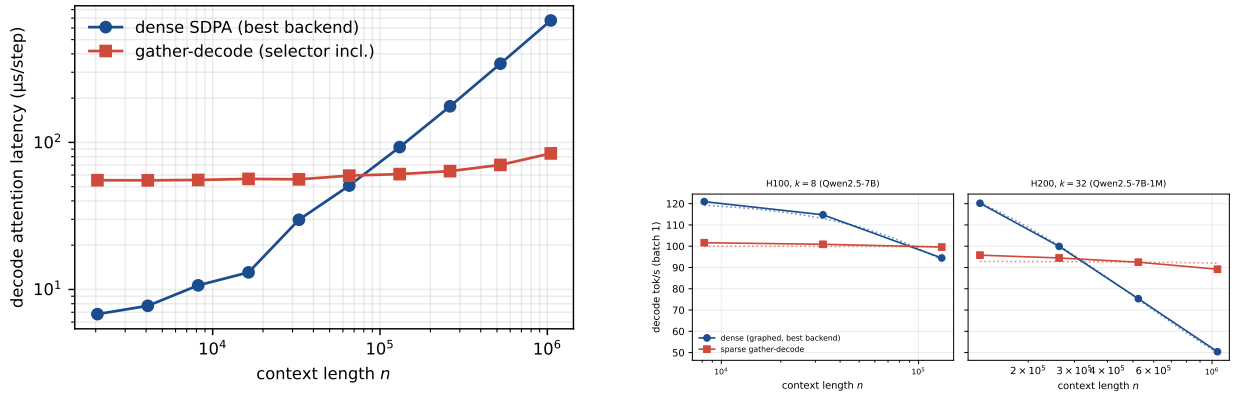
1. **The dense baseline is not a strawman.** cuDNN attention sustains 2.9–3.2 TB/s at $n \geq 128$K — 86–96% of the H100's 3.35 TB/s HBM3 peak in the roofline sense [Williams et al., 2009] (95% at batch 1; the top of the range needs the batch-8 shapes). Whatever we beat, we beat near the roofline.
2. **The sparse path is a flat 35–149 μs, independent of n to first order.** That is the Constant-Support Law as wall-clock: the gather kernel itself costs $\sim 23\,\mu$s; the rest is the *selector* — the literal price of finding. Dense latency doubles with every doubling of n; sparse latency barely moves (fused: 55.6 μs at 8K $\rightarrow$ 84.2 μs at 1M, batch 1; split-k: 35.2 $\rightarrow$ 65.9 μs; at batch 8 the same $O(n/128)$ summary scan grows the op $\sim$3$\times$ across the sweep, to 126.9 μs split-k — while dense grows to 5,324 μs at the same cell, a 42$\times$ gap).
3. **Crossover is where the bill says it is.** At batch 1 the crossover sits between 32K and 64K for split-k (2.28$\times$ at 128K, 10.24$\times$ at 1M) and between 64K and 128K for the fused variant (1.53$\times$ at 128K, 8.02$\times$ at 1M). At batch 8 the split-k crossover drops below 16K, reaching 11.51$\times$ at 128K, 18.9$\times$ at 256K and **41.94**$\times$ at 1M — batching multiplies the KV bytes the kernel eliminates while the sparse floor grows only mildly.

	n	dense μs	split-k (op-level best)		fused (used end-to-end)	
			μs	speedup	μs	speedup
batch 1	8K	10.7	35.2	0.30×	55.6	0.19×
	32K	29.8	37.1	0.80×	56.1	0.53×
	64K	50.9	37.9	1.34×	59.4	0.86×
	128K	92.9	40.7	2.28×	60.9	1.53×
	512K	343.0	51.0	6.72×	70.3	4.88×
	1M	674.8	65.9	**10.24**×	84.2	8.02×
batch 8	32K	174.9	44.8	3.91×	63.2	2.77×
	128K	674.1	58.6	11.51×	77.3	8.72×
	512K	2667.5	88.7	30.06×	109.7	24.32×
	1M	5323.9	126.9	**41.94**×	149.4	35.63×

Table 1. Decode-attention op latency per call (median of 50 graph-replayed iterations of 20 calls), Qwen2.5-7B shapes (GQA 28/4, D=128), H100. **Every cell of this table, both kernel variants, comes from the single committed artifact `s1_oplat.json`** (one measurement session; an earlier session's values, 2–4% apart, were retired with it — that spread is the honest run-to-run clock variance on rented GPUs). Dense = fastest probed SDPA backend (cuDNN at every shape; 2.9–3.2 TB/s achieved at $n \geq 128$K). Sparse columns include the full per-layer selector; the once-per-step context setup shared by all 28 layers is hoisted out of the per-call timing (it is part of c_1 in the end-to-end accounting of §9). Smaller n and remaining batch-8 rows are in the artifact.

The price of finding, paid down twice. Our first selector implementation cost $\sim 85\,\mu$s per layer — a ∼dozen small CUDA kernels whose *execution* time (not launch overhead; these were graph-captured) set the floor, putting the batch-1 crossover at 128K. We had pre-registered the floor as launch overhead (30–60 μs); the miss taught us where the cost actually lives. Two optimization rounds followed. First, hoisting step-invariant work out of the per-layer path (the validity mask and the sink/local ids are shared by all 28 layers) and fusing the two bounds GEMMs into one via a $[k_{\max} \| k_{\min}]$ concatenation cut the floor to 56 μs. Second, the fused kernel's own occupancy: one program per (batch, kv-head) is 4 programs at batch 1 on a 132-SM H100. A **split-k variant** — one program per (batch, kv-head, selected block), plus a log-sum-exp reduce — raises parallelism 13× and cuts the batch-1 sparse op to 35–66 μs across the sweep: batch-1 op speedup becomes **2.28**× at 128K and **10.24**× at 1M (vs. 1.53×/8.02× fused; both columns in `s1_oplat.json`), moving the batch-1 crossover to ∼32–64K. The remaining floor is dominated by the selector's top-k and scoring GEMM; a single-kernel selector would cut it again, and we leave that open with its composition measured.

H1 verdict. Pre-registered: batch-1 crossover in [8K, 64K], $\geq 8\times$ at 1M, judged on the fused configuration the prediction described. Measured (committed artifact): 0.86× *at* 64K — the crossover lies above the committed band's upper edge, a marginal miss but a miss; the 1M clause reads 8.02×, over the $\geq 8\times$ bar by only 0.2% — within the 2–4% run-to-run clock variance, so we score it "holds at the point estimate, not robustly." The full four-clause audit: crossover band missed (above), 1M $\geq 8\times$ marginally held, 128K 1.5–3.5× held (1.53×), anti-strawman ≥ 1500 GB/s held (2889+). Two held cleanly, one marginally, one marginally failed. (The later split-k variant happens to move the crossover back inside the band; the verdict above is for the configuration the prediction was made about, and the improvement is reported as engineering, not as a retroactive pass. The end-to-end sessions ran the fused kernel because split-k landed after they were committed; by Table 1's per-layer deltas (∼20 μs × 28 layers) a split-k engine would save ∼0.55 ms/step, so the e2e numbers understate the method's own best configuration — in the conservative direction.)



(a) Op latency vs n (batch 1). **(b)** End-to-end throughput vs n (batch 1).

Figure 1. The law as wall-clock: dense cost scales with context, sparse cost is flat. Dotted lines in panel (b): the fitted traffic model (Eq. 2); panel (a) shows raw op-level medians for the *fused* variant (the kernel the end-to-end engine runs); Table 1 carries both variants.

6 End-to-End Decode

Table 2 reports whole-model decode throughput — embedding to argmax, CUDA-graphed, A/B-interleaved — for the shared engine in dense and sparse modes (artifact `s2_e2e.json`; H100; the 128K/batch-8 and long-n cells from the H200 session, `s3_*.json`).

Context-invariant decode. The sparse engine's per-sequence throughput is flat in context length — ≈100 tok/s/seq at 8K, 32K, and 128K alike — while the dense engine decays as the KV term grows. That flat line is the Constant-Support Law expressed as a service-level property: with gather-decode, serving a 128K-context request costs the same per token as serving an 8K one.

Below the crossover, sparse is honestly slower. At 8K batch 1 the sparse engine runs at 0.84× dense: the keep-set saves only ∼0.37 GB of reads (dense reads 0.47 GB of KV; sparse still reads ∼0.10 GB of keep-set and summaries) against a 15.7 GB bill while paying ∼1.7 ms/step of selector work. Nothing about the kernel fixes this — it is the bill. And fidelity is not free: the budget that assures needle retrieval (k=32, §8.4) costs $c_1 = 3.3$ ms and shifts every crossover right. We report both columns; the speed a deployment gets is the speed at the budget its workload needs.

6.1 Long context on H200, and the bandwidth-dependence of the crossover

Table 3 extends batch-1 decode to 1M tokens on the in-distribution `Qwen2.5-7B-Instruct-1M` model (H200, k=32; artifact `s3_1m.json`). Two structural facts: per-sequence sparse throughput stays flat (89–96 tok/s from 128K to 1M — an 8× longer context costs 7%), and the crossover sits *later* on H200 than on H100 — faster HBM ($\beta \approx 4.5$ vs 3.0 TB/s) makes dense KV reads cheaper while the selector floor c_1 does not shrink at all — it measures *higher* on H200 (§9). The regime map is hardware-relative: the better the memory system, the longer the context needed before finding pays.

		dense	sparse, k=32 (assured; haystack-cond. §8.4)		sparse, k=8 (pre-registered)	
n	batch	tok/s	tok/s	speedup	tok/s	speedup
8K	1	121.0	86.2	0.70×	101.6	0.84×
8K	4	445.9	337.1	0.74×	397.2	0.89×
32K	1	114.8	85.6	0.73×	100.9	0.88×
32K	4	367.9	336.1	0.89×	395.5	1.07×
128K	1	94.5	84.6	0.88×	99.6	1.05×
128K	2	152.0	169.1	1.08×	196.1	1.29×
128K	4	221.1	330.9	1.46×	388.7	1.76×

Table 2. End-to-end greedy decode throughput (median of 3 interleaved repeats of 256 tokens), Qwen2.5-7B, H100; batch-2 rows at 8K/32K omitted for space (artifacts `s2_e2e_k32.json`, `s2_e2e.json`). The *leading* column is the budget that passes this paper's own fidelity gates (k=32, §8.4); the k=8 column is the pre-registered budget, which fails the needle gate at 32K and is reported for the prediction audit. The three-parameter model of §9 (β, c_0 fitted on the dense rows; c_1 on the sparse rows) reproduces the k=8 column at $R^2 = 0.998$ and prices the k=32 column with its measured $c_1 = 3.3$ ms (field `fit_k32` of `model_fit.json`; that fit reuses the k=8 session's β, c_0 on the k=32 session's sparse rows — cross-session by necessity, within the disclosed 2–4% drift). Dense backend: probed-fastest (cuDNN) at every cell. Speedups are *within-session* interleaved ratios; the printed dense column is the k=8 session's, and the k=32 session's own dense baseline differs by 1–3% (rented-GPU clock drift), so dividing across columns reproduces the ratios only to that tolerance.

n	dense tok/s	sparse tok/s	speedup	forced agreement
128K	120.2	95.8	0.80×	0.844
256K	99.9	94.5	0.95×	0.844
512K	75.3	92.5	1.23×	1.000
1M	50.4	89.2	**1.77**×	0.938
128K, batch 8	395.0	715.2	**1.81**×	0.906

Table 3. Decode at long context: Qwen2.5-7B-Instruct-1M, H200, k=32, 128 generated tokens (artifacts `s3_1m.json`, `s3_128k_b8.json`). Agreement cells are 32 teacher-forced steps each — Wilson 95% half-widths up to ±0.13 — and are consistency checks, not certifications; §8.2 carries the powered fidelity measurements. The dense engine's own needle accuracy at 1M is 0/3 under this protocol; §8.4 reports the root-cause investigation (the harness is cleared; the effect is length-driven).

GATE-3 verdict (from the BRL-2026-06 runbook: 2–4× decode at 128K, 5–10× at 1M, vs the best dense baseline). *Not met in any measured cell* — including (128K, batch ≥ 4) and (1M, batch 1), which the pre-registration predicted would pass (2.8×/4.5×) or approximately pass (4.6×); only the (128K, batch 1) failure was pre-registered. Measured maxima are 1.81× (128K, batch 8, H200, assured budget) and 1.77× (1M, batch 1). The shortfall is fully priced by the fitted bill: the engine's own fixed cost c_0 — paid identically by both modes and absent from the gate's FLOP-count reasoning — caps the ratio. The $c_0 \to 0$ projection (§9) recovers the gate's 128K clause (2.1×) but not its 1M batch-1 clause (3.6×, still below 5×; ∼20× at batch 8, above it); the attention-op measurements (11.5–41.9×, Table 1) sit entirely above the gate's ranges rather than match them. The gate was written about attention; the harness bill is what a deployment actually pays.

The regime map (§9) is therefore the central deliverable: constant-support decoding is a long-context, batched-serving technology, and the fitted model — with the hardware's β and the

workload's budget plugged in — tells an operator where to switch it on (for this model family and engine; transfer to other stacks is by substitution, §13).

7 Matched-Budget Baselines

The closest systems to this work are page-granular query-aware sparsity (Quest [Tang et al., 2024], whose bound-scoring rule our selector adopts) and sink-plus-window streaming (StreamingLLM [Xiao et al., 2024b]). We run both *inside the same engine* — same weights, cache, prefill, harness; only the keep-set policy changes — on the in-distribution 1M model (artifacts `s6_*.json`; the Quest-128K cell is `s6_quest16_131072_v2.json`, an immutable re-measurement after an engine plumbing fix — the broken first run is retained as `..._prefix_bug.json` with an audit note; 60 paired needle trials and 256 true-text forced steps per cell). This is a *policy-and-granularity* comparison inside one harness, not a reproduction of the published systems: our Quest-style row keeps Quest's page-16 bound scoring but runs it through our unfused PyTorch selector (published Quest fuses its estimation kernels and reports end-to-end wins at its own operating points), it inherits our engine's always-kept sink+local set and applies sparsity to all 28 layers (published Quest keeps the first two dense), and the page budget is matched to ours-k=8 to within 7%: (1 sink +32 local +64 selected) $\times$16-token pages = 1,552 keys vs our 13 blocks $\times$128 = 1,664 — the page-granular sink is 16 tokens where ours is 128, a 112-key deficit that slightly disfavors the page policy (which still wins per-key fidelity). The window row is run twice: at StreamingLLM's own default-scale budget (640 keys) and at a window budget-matched to ours-k=32 (4,608 vs 4,736 keys; artifact `s6_stream4k6_*.json`), so no policy is penalized by budget.

	$n = 32$K				$n = 128$K			
policy (keys)	needle	agree	ppl	spd	needle	agree	ppl	spd
dense (all)	1.00	—	11.68	1.00	1.00	—	7.53	1.00
window, k=0 (640)	0.00	0.809	16.14	1.04	0.00	0.777	10.27	1.27
window, k=0 (4,608)	0.00	0.938	12.29	0.80	0.00	0.949	7.59	0.98
ours, k=8 (1,664)	1.00	0.895	12.56	0.88	0.87	0.832	8.61	1.05
ours, k=32 (4,736)	1.00	0.914	11.74	0.73	1.00	0.898	7.88	0.88
Quest-style p16, k=64 (1,552)	1.00	0.922	12.04	0.65	1.00	0.898	7.71	0.75

Table 4. Keep-set policies in one engine, Qwen2.5-7B-Instruct-1M, H100, batch 1. "needle" = 60-trial solve rate (dense reference 60/60 at both lengths); "agree"/"ppl" = 256-step true-text teacher forcing; "spd" = e2e decode throughput vs dense (within-session interleaved; each column's cells come from their own sessions, and the printed dense row is the ours-k=32 session's at 32K and the ours-k=8 session's at 128K — third significant digits are session-dependent). k counts distant *blocks* for ours, 16-token *pages* for Quest-style, and is 0 for the window rows. The window rows cannot retrieve at any budget (the needle is outside any window); page-16 selection retrieves perfectly and is the per-key fidelity leader, but in this engine costs 0.65–0.75×: 16-token pages mean 8× as many summaries, and the cost is the *selector's op execution* (the extra summary *bytes* are 0.1–0.4 GB/step, i.e. 34–135 μs at β — the measured 3.8–4.4 ms/step gap against the key-matched block-128 row is unfused scoring work, which published Quest fuses). Block-128 runs 0.88–1.05× at k=8 and 0.73–0.88× at the assured k=32, and the only block-128 cell above 1.0× (k=8, 128K) is also the cell that fails the needle gate (0.87) — speed at the budget a workload needs is the honest read-off.

The matched-budget window row sharpens the structural point: given ours-k=32's key budget, the window's distributional fidelity becomes competitive (perplexity within 1.1–5.7% of its own session's dense reference, KL 0.023–0.031; the table's dense row is the ours-session's) — and its retrieval stays exactly zero (0/60 at both lengths). Retrieval failure is the *policy's* geometry (the

needle lies outside any window), not a budget artifact; meanwhile the small-window row's +38% perplexity was a budget artifact, and we print both rows rather than quote the flattering one. The distributional ladder at 32K: mean per-step KL(dense‖sparse) is 0.374 for the 640-key window, 0.023 for the matched 4.6K window, 0.072 at ours-k=8, 0.020 at ours-k=32 — at which point sparse perplexity is within 0.6% of dense *and* retrieval is 60/60. Eviction baselines (H2O-class) are deliberately absent: BRL-2026-06 already measured store-time eviction failing its fidelity bar at far smaller compression than read-time sparsity sustains, and our engine keeps every key by design.

8 Fidelity Under the Kernel

8.1 The S0 gate caught a real failure

The cheapest stage of the program produced its first scientific result. With *mean-key* block summaries scored by the GQA group-mean query — the natural first reading of BRL-2026-06's selector — the engine passed every synthetic test and the text-continuation bar (teacher-forced agreement 1.000 at 8K; the free-running sparse engine generated 64/64 *identical* tokens), yet failed the distant needle: 6/25 solves vs. the dense engine's 25/25 (`s0_7b_8k.json`). The two fidelity probes dissociate: text continuation is served by sink+local+*any* semantically near blocks; needle retrieval requires finding *one specific* distant block.

The diagnosis is double dilution. A ∼8-token needle inside a 128-token block is averaged away on the *key* side by the mean summary; the retrieval head's query is averaged away on the *query* side by the group mean. The ablation separates the two cleanly (Table 5): doubling the block budget under mean scoring recovers only 0.84 — more blocks cannot fix a scoring rule that cannot see the needle — while Quest-style bounds scoring [Tang et al., 2024] per query head ($\text{score}_g = \text{relu}(q_g) \cdot k_{\max} + (-\text{relu}(-q_g)) \cdot k_{\min}$, an upper bound on any token's dot product in the block, then *max* over the group's 7 heads) recovers 25/25 at the original budget. This replicates, at decode time and through a fused kernel, the same failure-and-fix BRL-2026-06 documented for its v2 selector. The selector revision was committed as a pre-registration amendment *before* any S1–S3 measurement; it changes predicted traffic by $< 0.5\%$.

selector scoring	distant blocks	needle solve @8K	dense reference
mean-key, group-mean query	4	6/25 (0.24)	25/25
mean-key, group-mean query	8	21/25 (0.84)	25/25
bounds (per-head, max over group)	4	**25/25** (1.00)	25/25
bounds (per-head, max over group)	8	**25/25** (1.00)	25/25

Table 5. S0b selector ablation (Qwen2.5-7B, n=8K, 25 needle trials each; artifacts `s0b_needle_*.json`; the mean/top-4 row is the original S0 run, `s0_7b_8k.json`, produced under the pre-amendment defaults — reproducing it today requires `--score-mode mean --topk 4`, and that artifact predates per-run config logging). Scoring, not budget, is the binding constraint: doubling the block budget under mean scoring recovers 0.84; bounds scoring recovers 1.00 at the original budget. Production config: bounds, top-8 distant blocks.

8.2 Distributional fidelity on natural text

Argmax agreement over a few dozen steps is a fragile metric — it cannot distinguish a retrieval failure from a coin-flip at a near-tie, and its denominators are small. The primary fidelity protocol is therefore **distributional**: both engines are teacher-forced on the same *true* wikitext continuation (256–1024 steps), and we report per-step KL(dense‖sparse), total variation, and the perplexity each

engine assigns the true next tokens (per-step logits are stored fp16; the smallest reported KLs — 0.0018–0.0032 — sit within an order of magnitude of that quantization's noise floor, identically for both engines, so orderings are robust but the last digit is not). Table 4 carries the single-window values; replication over five disjoint wikitext windows (`s10_fidwin_*.json`) gives the claim its uncertainty: at the assured budget, Δppl = $+0.43\% \pm 1.85\%$ (mean $\pm$ sd across windows) — statistically indistinguishable from zero — and at k=8, $+0.91\% \pm 1.70\%$; mean KL is tight and cleanly budget-ordered, 0.014 ± 0.004 at k=32 vs 0.038 ± 0.010 at k=8 (non-overlapping window ranges). One dispersion honesty note: the baseline panel's window (a different corpus seed) reads 0.072 at k=8 — outside the five-window range — so window-to-window spread at the small budget exceeds what the five-window sd alone suggests; the k=32 value (0.020) sits inside its range. The single-window 128K cell reads +5.3% and awaits the same replication. Scale and family both help: on Qwen2.5-14B the same budget (batch 1) gives mean KL 0.0032 (`s7_Qwen25-14B-Instruct.json`), 3.7–6.4× lower than the nearest 7B comparators — but those are at 32K (five-window mean 0.0143; `s10_fidwin_k32_*.json`), and the same budget's pure *length* effect on Llama (0.0018 at 8K $\rightarrow$ 0.0145 at 32K) exceeds that gap, so the comparison is length-confounded and we print it as a single matched-budget data point, not a scale trend — the direction BRL-2026-06's shrinking $\kappa^\star$ predicts — and on a second family, Llama-3.1-8B, KL is 0.0018–0.0029 at 8K and 0.0085–0.0145 at 32K, with sparse perplexity equal to dense to three significant figures at 8K (3.67 vs 3.67) and needle recall 60/60 at every budget $k \in \{8, 16, 32\}$ at 32K (`s7_Meta-Llama-31-8B-Instruct.json`). The method is not a Qwen artifact. We flag scale as a direction consistent with BRL-2026-06's shrinking $\kappa^\star$, not a fitted trend.

Long generation. Block summaries are computed at prefill; without maintenance, tokens generated past the local window would lose selectable access to the model's own output. The engine therefore folds each new key into its block's running bounds (∼6 extra ops per layer-step, optional flag). Validation at 1,024 teacher-forced steps — twice the local window — holds agreement 0.92, mean KL 0.041, and Δppl +1.8% (`s9_longgen_updates.json`).

8.3 Per-step agreement: ties, not retrieval

On wikitext (real prose; degenerate corpora bias this measurement in opposite directions — looped filler pushes dense attention into the diffuse aggregative regime that is the law's known boundary, while synthetic random-fact text removes predictability so the argmax sits on near-ties everywhere), teacher-forced per-step agreement at the k=8 budget is 0.906 at both 8K and 32K — flat over that range (artifacts `s2d_fid_budget_*.json`, `s2e_fid_final_*.json`); the 1M-model cells at 128K–256K read 0.844 from only 32 teacher-forced steps (Wilson 95% $[0.68, 0.93]$), too underpowered to extend the flatness claim, and we flag them as such in Table 3. At 8K, agreement rises with budget: the k=32 grid reads 0.969 at batch 1 (0.977/0.988 at batch 2/4; `s2_e2e_k32.json`). The cleaner budget read-out is the paired single-prefill sweep: $0.906 \rightarrow 0.969 \rightarrow 0.984$ at 8K and $0.906 \rightarrow 0.922 \rightarrow 0.938$ at 32K for $k = 8 \rightarrow 16 \rightarrow 32$ (`s2e_fid_final_*.json`); the 32K *grid* cells, measured in different sessions on different prompts, read 0.922 at k=8 and 0.906 at k=32 — ordering reversed by session noise, not a contradiction of the paired sweep.

The margins say what the misses are. Real text decodes at a median top1−top2 logit margin of just 0.75–1.44; the disagreements concentrate at margins ≤ 0.5 — several at exactly 0.0, i.e. ties that any perturbation flips, including floating-point reordering. Conditioning on confident steps (margin > 1): we pool over *every* non-quarantined ours-policy artifact row that logs the statistic — the rule and the full pool membership are computed by `analysis.py` into `pooled_agreement.json` (11 rows at k=8, spanning the 7B base and 1M models; 22 at k=32, spanning four checkpoints in

two families; 64–1,024 steps) — giving 1823/1851 = 0.985 (Wilson 95% [0.978, 0.990]) at k=8 and 4060/4070 = **0.9975** ([0.995, 0.999]) at the assured k=32 budget. Only the assured budget clears the predecessor's 0.99 bar; the k=8 pool sits just below it, dominated by the 128K window (142/154; row rates span 0.92–1.00). One scoping conflict, disclosed: the mechanical rule admits three base-model 128K rows into the k=32 pool that §8.4 declares an invalid fidelity regime; excluding them moves the pool from 0.9975 to 0.9984, so we keep the pre-stated rule and note the immateriality. Composition, for the same reason: 56% of the k=32 pool's confident steps come from the near-perfect 14B and Llama rows; restricting to the 7B family gives 1797/1806 = 0.995, still clearing the bar. These intervals treat teacher-forced steps as independent; steps within a window are autocorrelated, so they are anti-conservative — the row-level picture (30 of 33 rows strictly at or above 0.98; the three below, all k=8, run 0.922/0.974/0.9799) is the robustness check the iid assumption cannot give. Individual small cells scatter around these pools (23/23 at 32K, 38/39 at 8K in the original 64-step runs). Across the pools, confident flips occur at 0.2–1.5% of confident steps, with margins up to 6.9 — the 3.5 maximum of the original 64-step cells does not bound the 128K windows. The strict unconditional 0.99 argmax bar is, on real text, partly a bar on reproducing coin flips. The honest statement, scoped to the original 64-step base-model cells: *in those runs the sparse engine reproduces every confident dense prediction at 32K and all but one at 8K, disagreeing otherwise only where the dense model is near-indifferent (margin $\leq$ 0.5).* Across the full k=8 pool the picture is less clean — 28 confident flips in 1,851 steps, concentrated in the 128K window — which is why the pooled 0.985, not the clean small-cell story, is the number we put in the abstract.

8.4 Findability: the budget the law does not fix

The needle task — a unique code at a random distant position, answered greedily through the real engine — is where realizable finding is stressed, and it produced the report's second principal measurement (Fig. 2; artifacts `s2c_needle_*`, `s3_findability_*`). That fixed-budget sparse attention degrades with context length is documented across methods and tasks [Nawrot et al., 2025]; what this section adds is the quantification *for this engine and selector* — paired against a same-cache dense reference, certified at 500-trial power where it holds, with the collapse localized to a measured horizon:

- On the **base** model (Qwen2.5-7B-Instruct, 32K-native), recall at fixed budget *decays with context*: 125/125 at 8K (pooled with the powered replication `s10_base8k_100.json`) but 0.88 (Wilson [0.83, 0.92], 200 trials) at 32K with k=8 — one-sided Fisher exact $p = 5\times10^{-6}$; restoring it needs the budget to grow — 0.983 at k=16, 1.00 (60/60) at k=32.
- Beyond its native window the base model is not a valid fidelity reference at all: at 128K its *own dense* needle accuracy collapses to 0.67 (RoPE extrapolation), and the 128K dense–sparse agreement cells of the timing grid read 0.57–0.64 (`s2_e2e.json`) — numbers we print because they are in the artifact, and which measure two engines disagreeing *inside* a regime where the reference itself is broken. Long-n fidelity must be measured on the long-context-trained model, and is (`s8_cert_128k.json`).
- On **Qwen2.5-7B-Instruct-1M**, the picture changes qualitatively at moderate length, and is now measured at the predecessor's 500-trial *power*: at 32K, k=8 solves 497/500 and k=16 499/500 against dense 500/500 (`s8_cert_32k.json`) — one-sided 97.5% lower bounds (Wilson) 0.982 and 0.988; at 128K both k=16 and k=32 give 200/200 (lower bound 0.981, `s8_cert_128k.json`; k=8 gives 0.90 in the 30-trial paired surface). A zero-failure run of n=200

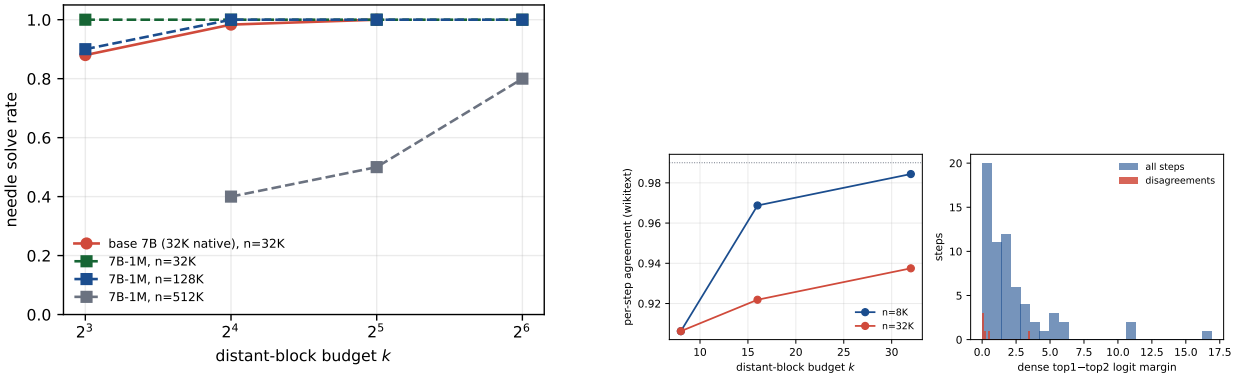
demonstrates at most ≈ 0.98, so we state lower bounds rather than claim the 0.99 bar itself. Paired designs sharpen every such contrast: the base-model 32K discordant pairs are 24 dense-only vs 0 sparse-only solves (one-sided exact binomial $p = 2^{-24}$). Long-context training appears to shape keys so the relevant block wins the block-summary score outright.

- The certifications above are *haystack-conditional*, and we measured how much: replacing the looped-paragraph filler with real prose (wikitext) at the same 32K length drops k=8 recall from 0.994 to **0.58** (Wilson $[0.48, 0.67]$, 100 trials, dense 100/100), k=16 to 0.78, while k=32 holds 0.96 ($[0.90, 0.98]$; `s10_filler_wikitext_32768.json`). Real prose populates the cache with semantically competitive blocks, and the budget the selector needs grows with the background's competitiveness — assurance is a property of (model, budget, *haystack*), not of the selector alone. The looped haystack is the easy case and the certified numbers above should be read as its upper envelope.
- But at 512K even this model's findability collapses at small budgets: 0.4 at k=16, 0.5 at k=32, 0.8 at k=64 (10 paired trials, dense 10/10; labeled consistency-scale). The required budget is flat-to-logarithmic out to 128K and then turns sharply upward.

The dense 0/3 at 1M, investigated. A dense reference failing its own retrieval task at the title length would normally scream harness bug, so we treated it as one before believing it. The suspect was the chunked-prefill path (used only beyond 512K): forcing the 512K needle suite through it reproduces the full-sequence result — dense 10/10 exactly, sparse 4/10 vs. 5/10 at k=32, a one-solve difference (`s4_512k_chunked.json`) — clearing the numerics. A position probe then separates distance from length: at a 1M total context the dense engine solves 0/5 needles even when the needle sits at 80–95% depth — 50–200K tokens from the query, a range it handles perfectly inside a 512K context — and 0/5 at distant positions (`s4_1m_near.json`, `s4_1m_far.json`). The collapse tracks *total length*, not retrieval distance. The harness is cleared; the residual effect is a property of this model's raw-completion behavior at the million-token range (its vendor's 1M results use a chat-template protocol), and every 1M fidelity statement in this report is scoped accordingly: at 1M we certify *throughput* (with agreement as a consistency check, Table 3), and no engine — dense included — certifies task-level retrieval under raw completion. We then probed the protocol itself (`s10_chat_long.json`; a deliberately small probe — 8 trials per arm at 512K, 5 at 1M — powered to detect only gross protocol effects): under the vendor's chat template, with generation length sized for chat-style answers, the dense engine solves 8/8 at 512K — the protocol works where raw completion also works — and sparse retrieval under chat matches its raw-completion rates at the same length within noise (4/8 at k=32 and 6/8 at k=64 under chat, vs. 5/10 and 8/10 raw; overlapping Wilson intervals, different trial counts), so the chat template neither helps nor hurts the *finding* problem. At 1M the chat template does not rescue the dense engine either: 0/5 under chat, as under raw completion — so the total-length effect stands under both protocols we can construct, and the vendor's reported 1M retrieval (obtained through its own evaluation stack) remains unreconciled with this engine; we flag that as an open external check rather than assume either side is wrong. The gap between claimed and effective context length is itself well documented [Hsieh et al., 2024]; what this paragraph contributes is the root-cause discipline of not letting it masquerade as a sparse-attention failure.

Findability, then, is not only a selector property but a *model* property, and it is the program's open frontier — the exact inverse of BRL-2026-06's result. Support is constant (oracle); finding *cost* is cheap (the 56 μs floor; even k=64 adds little); but finding *reliability* at a fixed budget depends on the key geometry the model was trained into, and beyond a model-dependent horizon ($\sim$128K–

512K here) the one-shot block-summary score stops ranking the right block into any small top-k. Candidate remedies — finer summary granularity, multi-round selection (BRL-2026-06's cross-layer route), or training the *selector* for findability — are taken up in BRL-2026-08; the surface in Fig. 2 is the measured target they must hit.



(a) Needle recall vs distant-block budget. **(b)** Wikitext agreement vs budget; disagreement margins.

Figure 2. Findability and fidelity. Left: needle recall vs distant-block budget. The 32K-native model, at its own native length, already needs budget growth (0.88 at k=8 → 1.00 at k=32); the 1M-trained model holds recall ≥ 0.9 out to 128K (1.00 at $k \geq 16$, 0.90 at k=8) and collapses at 512K for $k \leq 32$ and degrades to 0.8 at k=64. Right: per-step disagreements concentrate at small margins (≤ 0.5), with one confident exception at 8K (margin 3.5).

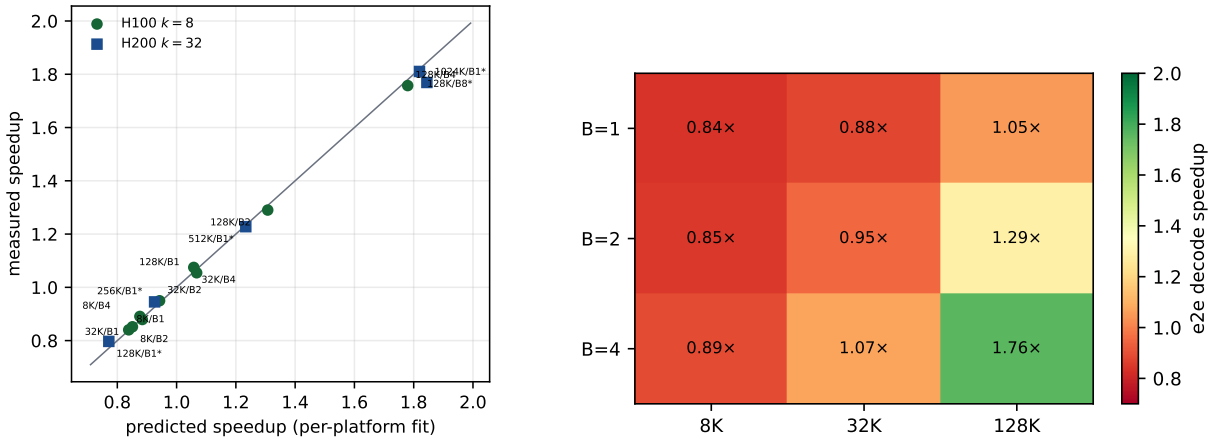
9 The Regime Map

The pre-registered model (Eq. 1) gains one term the pre-registration missed and the measurement found: the engine-level price of finding,

$$t_{\text{dense}}(n, B) = \frac{W + B\,A_d(n)}{\beta} + c_0, \qquad t_{\text{sparse}}(n, B) = \frac{W + B\,A_s(n)}{\beta} + c_0 + c_1. \tag{2}$$

β and c_0 are fitted on the *dense* rows only; c_1 is one parameter fitted on the sparse rows. The fit gives $\beta = 3.05$ TB/s (consistent with the op-level cuDNN bandwidth), $c_0 = 3.2$ ms (the engine's fixed step cost: ∼ 400 small kernels of norms/projections/bookkeeping per step — identical in both modes), and $c_1 = 1.74$ ms (28 layers of selector work — 11% above the $28 \times 56\,\mu$s op-level floor measured independently in §5; the keep-set is priced at the production 13 blocks). With these three numbers the model reproduces **every** H100 grid cell: $R^2 = 0.998$ on the speedup over the ($n \times B$) grid (Fig. 3; artifact `model_fit.json`). These R^2 values are in-sample fit quality; the *predictive* check holds out all batch-4 cells, fits on batches 1–2, and predicts the held-out speedups to within 2.2% (max 2.11% at the 8K cell; field `holdout_B4`; an interpolation check inside the measured grid — no held-out cell exists at batch 8, long n, or on H200). The H200 long-context rows (Table 3) are priced by the same equation with all three parameters refitted per platform: $\beta \approx 4.5$ TB/s, $c_0 \approx 3.2$ ms, and $c_1 = 4.1$ ms — 23% above the H100's k=32 value, so the selector floor is *not* bandwidth-independent across platforms (plausibly clock/occupancy differences; we flag rather than explain it). The H200 rows fit at $R^2 = 0.992$ (`fit_h200`; three parameters on five cells, in-sample only — no held-out check at H200); the measured 1.77× at 1M emerges from the same bill. One number the regime map does *not* price: first-token cost — prefilling the 1M context takes 833 s of dense compute on the H200 (`s3_1m.json`); the map prices decode only, and says so.

H2/H3 verdicts. Both fail as committed; we state this plainly because the two hypotheses shared the same two-term model, and a paper whose brand is honest gates does not get to let H2 absorb the failure while H3 claims the win. *H2*: the point predictions are systematically high (e.g. 2.82× predicted vs. 1.76× measured at 128K/batch 4) — refuted at the committed threshold (measured $< 0.7\times$ of prediction). *H3 as pre-registered* — the fit quality of the *no-free-parameter* sparse prediction from that same two-term model — fails its $R^2 \geq 0.95$ bar decisively: $R^2 = 0.296$ (field `h3_prereg_two_term_r2` of `model_fit.json`). What holds, at $R^2 = 0.998$, is the **amended** three-term model of Eq. 2, with c_1 fitted on the sparse rows; it is an amendment, not the committed hypothesis, and we label it so. The corrected model is not a post-hoc patch: c_1 is independently measured at the op level, and c_0 is visible in any per-step trace. H2's *fidelity* clause is likewise refuted as committed: teacher-forced agreement at the k=8 budget is 0.906, below the unconditional 0.99 bar (`s2e_fid_final_*.json`), and the 32K needle rate 0.88 (Wilson [0.83, 0.92], `s2_e2e.json`) excludes the dense 200/200 — so the runbook's S2 fidelity gate also fails as written. The margin analysis of §8.2 is our post-hoc diagnosis of that miss, not the committed bar; the k=32 budget restores the needle bar (§8.4) at the measured extra cost in Table 2.



(a) Measured vs modeled speedup, all cells. **(b)** The regime map: e2e speedup over ($n \times B$).

Figure 3. The *amended* three-parameter bill prices every regime ($R^2 = 0.998$ in-sample; $\leq 2.2\%$ held-out error on batch-4 cells; artifact `model_fit.json`). Panel (b) maps the k=8 grid — the budget that fails the fidelity gates; the assured k=32 grid peaks at 1.46× at the same (128K, batch-4) cell (Table 2). H3 as pre-registered (two-term, no free sparse parameters) failed, §9.

What a faster runtime would get. Both engines pay the same c_0; the speedup ratio is therefore capped by harness overhead that has nothing to do with attention. Substituting $c_0 \to 0$ (a perfectly fused runtime) into Eq. 2, with measured β and c_1 (labeled projection): 2.1× at (128K, batch 4), 3.6× at (1M, batch 1) and ∼20× at (1M, batch 8) at the k=8 budget; a fused selector ($c_1 \to 0$) raises the last to ∼ 25×, converging on the op-level numbers of Table 1 as it must. Because k=8 fails this paper's fidelity gates, the deployment-honest version of the same projection uses the *assured* budget's c_1 with each platform's own fit (3.3 ms/H100 for the 128K cell; 4.1 ms/H200 with $\beta = 4.5$ TB/s for the 1M cells, where that hardware was measured): 1.7× at (128K, batch 4), 2.2× at (1M, batch 1), 12.5× at (1M, batch 8) — the numbers an operator should plan with until findability at small k improves (BRL-2026-08).

10 The Cache Does Not Need the GPU: KV Tiering

The Constant-Support Law's strongest systems implication is one BRL-2026-06 filed under "store — unsolved": if a decode step *reads* only a few dozen blocks per layer (13 at k=8; the demo runs the assured budget's 37), the other 99.5–99.9% of the cache has no business occupying HBM. We demonstrate this directly. The 60 GB KV cache of a 1M-token context is placed in pinned *host* memory; the GPU holds the weights, the block summaries (∼0.5 GB), and a per-layer staging buffer; the selector doubles as a prefetcher, pulling the keep-set over PCIe each step.

On a 24 GB A10 — a card on which dense 1M-context decoding cannot exist (the cache alone is 2.5× the GPU's entire memory) — the tiered engine decodes at **6.1** tok/s (artifact `s5_tiered.json`; the 1M cache was prefilled once on an H200 and shipped to the A10 host via network volume — prefill at 1M needs more than 24 GB of *compute* memory, and we say so rather than hide it). The honest comparison is the dense-from-host floor: streaming the full cache over the measured 24.0 GB/s PCIe link would cap dense at 0.40 tok/s, so constant-support reading beats the only dense option this hardware has by **15**× — and the gap widens with β_{PCIe} arithmetic, not kernel tuning. What 6.1 tok/s carries at 1M is *mechanics*, not task performance: the decoded text there is degenerate looping (consistent with §8.4's finding that no engine, dense included, certifies 1M retrieval under raw completion), so the demo certifies throughput and traffic, and we print that scoping here rather than only in the appendix. The tiered path is *validated at the certified length*: the same card, same prompt construction, at 128K — where §8.4 certifies retrieval at 200/200 — solves the needle at 6.6 tok/s (2.1× the 128K dense-from-host floor; artifact `s5_tiered_128k.json`; after emitting the correct code the continuation degenerates into chat-control tokens, as raw-spliced prompts do once the answer is complete), so the 1M degradation is the model's length effect, not a tiering artifact. What changes here is not a speedup but the *memory economics* of long context: HBM residency stops being the boundary of the context window — host RAM, and by the same argument NVMe, becomes the boundary, at terabytes per node.

11 Three Quantities, Revisited

BRL-2026-06 §6 separated three costs; this report prices each in wall-clock. **Read** — solved: the gather kernel itself reads the constant keep-set in $\sim 23\,\mu$s per layer regardless of n (the 35–84 μs batch-1 op cost is kernel + selector); dense reads scale linearly and lose by up to 41.9× at the op level. **Find** — now a *measured price*, not an asymptotic claim: $c_1 \approx 1.7$ ms per decoded token in PyTorch (28× $\sim 60\,\mu$s), which is what makes short-context batch-1 decoding net-negative; and, more fundamentally, the budget that finding requires is a property of the *model*, not just the selector: the base-32K model needs the distant budget to grow with n to hold needle recall ($8 \to 16+$ blocks from 8K to 32K), while the long-context-trained 1M model is findable at k=8 at 32K and k=16 at 128K (k=8 gives 0.90), collapsing at small budgets by 512K (§8). **Store** — untouched, as in BRL-2026-06: dense and sparse hold the same KV cache; at 1M that is 60 GB either way. Read-time sparsity accelerates; it does not shrink memory.

12 Related Work

FlashAttention-2/3 [Dao, 2024, Shah et al., 2024] and flash-decoding [Dao et al., 2023] make *dense* attention IO-optimal; our baseline inherits them via the probed SDPA backends (cuDNN attention won every shape) and sustains 86–96% of the HBM roofline at $n \geq 128$K — we accelerate by reading less, not by out-engineering the read. FlashInfer [Ye et al., 2025] provides serving-grade kernels including sparse gathers. Quest [Tang et al., 2024] introduced the min/max block-bound scoring

our selector adopts (per kv-head, max over the GQA group); MInference [Jiang et al., 2024] applies dynamic sparsity to prefill; NSA [Yuan et al., 2025] trains the sparsity in. H2O [Zhang et al., 2023] and StreamingLLM [Xiao et al., 2024b] *evict* — the store-time strategy BRL-2026-06 measured to fail its fidelity bar; we keep every key and sparsify reads only. On the memory-tier side, KV offloading is an active line: FlexGen schedules dense caches across GPU/CPU/disk for throughput-oriented batch inference; InfLLM combines block-summary selection with CPU-resident context memory; InfiniGen and ShadowKV speculate or reconstruct the needed KV from host-side stores; SparQ and MagicPIG reduce read traffic by other selection rules. Our tiering section should be read *with* that literature [Sheng et al., 2023, Xiao et al., 2024a, Lee et al., 2024, Sun et al., 2024, Ribar et al., 2024, Chen et al., 2025], not as its discovery: what we add is the measured accounting — a constant, law-bounded per-step PCIe transfer, a measured dense-from-host floor, and end-to-end validation at a retrieval-certified length on a fixed budget. On the findability side, that fixed-budget sparse attention needs a budget growing with context is documented across methods, models, and tasks by Nawrot et al. [2025]; the dependence of retrieval on specific attention structure is the retrieval-heads line [Wu et al., 2024, Xiao et al., 2025]; and the gap between claimed and effective context length that our 1M investigation re-encounters is the subject of RULER [Hsieh et al., 2024]. Our findability section quantifies that frontier for one engine and selector under paired, certified designs — a measurement within this program, not a new phenomenon. Distinct from all of these, this report's contribution is the *accounting*: a memory-traffic model whose pre-registered two-term form was amended by one measured term (the price of finding, §9 — the pre-registered form itself *failed* its gate and is reported so), refitted per platform, that then reproduces measured throughput across the H100 ($n \times B$) grid at in-sample $R^2 = 0.998$ with $\leq 2.2\%$ held-out error, and converts the Constant-Support Law into an operator's regime map.

13 Limitations

One model family, one scale. All wall-clock numbers are Qwen2.5-7B (and its 1M variant) on H100/H200. The traffic model's first two terms generalize by substitution (any W, KV geometry, β); the third does not — c_1 measured 23% *higher* on the faster platform, so an operator must re-fit it per platform before trusting any crossover. Even then it is a labeled projection, not a measurement; 72B-scale runs were outside this report's budget. **Our engine, not a production runtime.** The harness's fixed step cost ($c_0 = 3.2\,\text{ms}$) is $\sim 3.5\times$ vLLM's implied $\sim 0.9\,\text{ms}$ under Eq. 2 at the same model (164 vs 121 tok/s dense at 8K, batch 1 — a single-run anchor with TTFT subtracted, not a timed grid, and the vLLM version was not recorded — a release-hygiene miss we disclose; artifact `s2_vllm.json`), and since both modes pay c_0, our *measured* e2e ratios understate what a fused runtime would get; the $c_0 \to 0$ ceiling is reported as a projection. vLLM integration (paged KV) remains open, as in BRL-2026-06. **KV memory is not reduced** — read-time sparsity only; the 60 GB cache at 1M is paid by both paths. **Fidelity scope.** The per-step bar is measured on wikitext and the needle task through greedy decode; sampling-based generation, aggregative tasks (the law's known boundary), and multi-hop retrieval (BRL-2026-06's cross-layer case) are not covered. Needle trials at 512K–1M are small (10/3) and labeled consistency checks. **Needle haystack.** The retrieval suites use a looped paragraph corpus as filler (unique needle, repetitive background — the easiest haystack for a selector, arguably the hardest for dense attention's aggregative regime), with needle positions sampled uniformly at 10–60% depth (deeper positions probed separately in §8.4's 1M investigation). The corpus-variation probe (§8.4) quantifies the haystack effect at 32K; the long-n certifications were not re-run under real prose, so their wikitext counterparts are unknown and likely lower.

Rented GPUs. Clocks are not locked; D/S interleaving within one process bounds drift but does not eliminate it (run-to-run sessions differ by 2–4%; the headline fidelity runs `s2d/s2e` executed on L40S, the timing grids on H100/H200, stated per artifact). **Pre-registration is attested, not externally verifiable**: the predictions file and the results entered version control in the same release commit; future reports will commit pre-registrations as separate timestamped commits or external registrations before any measurement stage runs. **Selector floor.** The 56 μs/layer finding cost is PyTorch op execution; a fully fused selector kernel would cut it further — left open, with the floor's composition measured so the next engineer knows what to fuse. **Generation-window sizing.** The static cache is sized at engine construction ($n + \text{gen\_len} + 8$; 264 extra slots in the H100 grid runs, 136 in the H200 sessions (gen=128) — the same slack bounded in §4); generations longer than the constructed window require re-prefill or a larger window — a harness constraint, not a method one, and unpriced here.

14 Conclusion

BRL-2026-06 proved a law and confessed a debt: *"no wall-clock speedup... a fused kernel... we explicitly leave open."* This report pays it. The fused gather-decode kernel turns the Constant-Support Law into measured wall-clock: at the attention op, up to 41.9× against a dense read sustaining 86–96% of the H100's memory roofline; end to end, decode whose per-sequence cost does not depend on context length, worth 1.77× at 1M tokens even inside an unfused harness (whose own fixed costs both modes pay). Just as important is what the bill refused to pay: below the crossover the law buys nothing (and the measurement says so), the engine's own fixed costs cap end-to-end ratios (and the three-parameter model prices the cap at $R^2 = 0.998$), and finding — cheap in microseconds — demands a budget that depends on the model's key geometry, with long-context training making keys findable at the realizable engine's fixed budget (itself orders of magnitude above the oracle $\kappa^\star$). The next debts are named: a fused selector kernel, a paged-runtime integration, sparse *prefill* (this report prices decode only; first-token cost at 1M still pays the full dense bill), frontier-scale validation, and the findability geometry of trained keys as an object of study in itself — the last is taken up immediately in BRL-2026-08.

Code availability

The complete implementation — kernels, engine, baselines, benchmark suite, analysis, and the per-stage launchers whose names appear in Appendix A — is released with this report (`papers/brl07/`, version-controlled from a clean initial commit). The repository deliberately retains the one-off diagnostic and fix entrypoints of the program's iteration history: the audit trail is part of the method.

Compute statement

All experiments ran on Modal (serverless GPUs): L40S for development and fidelity stages, H100 for op-level, end-to-end, baseline and family grids, H200 for long-context sessions, A10 for the tiering demonstration (provisioned as Modal's `A10G` SKU; the driver reports NVIDIA A10, and we use the driver's name throughout). We publish the price of verification: re-running every number in this report — including the internal-review revision cycle (baselines, certifications, second family, split-k, tiering, the chat-protocol probes, the matched-window baseline, and the diagnostic runs) — costs **$135.20** of serverless GPU time (sum of the per-run estimates in `results/cost_ledger.json`, which itemizes every run of this project and only this project; the successor project's ledger lives in its own repository). We report this as a property of the measurement program: any claim here can be independently re-measured for tens of dollars: the ledger prices each run, and Appendix A names the command per artifact. Budget history, for the record: the pre-registration committed a $150 cap; the PI raised it to $200 and then $500 during the revision cycle.

References

Mohammad Alsufi and Connor Scott. Attention has a type: Constant support, linear finding. Technical Report BRL-2026-06, Brainsless Research Lab, 2026. URL https://brainsless.com.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. MagicPIG: LSH sampling for efficient LLM generation. *International Conference on Learning Representations*, 2025.

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

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-Decoding for long-context inference. PyTorch blog, 2023.

Alan Gray. Getting started with CUDA graphs. NVIDIA Developer Blog, 2019.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Conference on Language Modeling*, 2024.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Symposium on Operating Systems Principles*, 2023.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. *USENIX OSDI*, 2024.

Piotr Nawrot, Robert Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo M. Ponti. The sparse frontier: Sparse attention trade-offs in transformer LLMs. *arXiv preprint arXiv:2504.17768*, 2025.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025a.

Qwen Team. Qwen2.5-1M technical report. *arXiv preprint arXiv:2501.15383*, 2025b.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. SparQ attention: Bandwidth-efficient LLM inference. In *International Conference on Machine Learning*, 2024.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

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.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. ShadowKV: KV cache in shadows for high-throughput long-context LLM inference. *arXiv preprint arXiv:2410.21465*, 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 *International Conference on Machine Learning*, 2024.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Wenhao Wu, Yizhong Wang, Guangxuan Xiao, Hao Peng, and Yao Fu. Retrieval head mechanistically explains long-context factuality. *arXiv preprint arXiv:2404.15574*, 2024.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. *Advances in Neural Information Processing Systems*, 2024a.

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

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *International Conference on Learning Representations*, 2025.

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. *arXiv preprint arXiv:2501.01005*, 2025.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, 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.

A Artifact Manifest

Every number in this report is reproducible from a JSON artifact in the `brl07-results` Modal volume, produced by the named command (run from `papers/brl07`). Code: `papers/brl07/` (kernel, engine, benches, this manifest's commands).

artifact	produced by	used in
`s0_correctness.json`	`modal run modal_kernel.py::run_unit`	§3
`s0_7b_8k.json`, `s0b_needle_*.json`	`::run_unit`, `::run_s0b`	Table 5
`s1_oplat.json`	`::run_op`	Table 1
`s2_e2e.json`	`::run_e2e`	Table 2, §8.4
`s2_vllm.json`	`::run_vllm_anchor`	§13
`s2c_needle_*.json`, `s2d/e_fid_*.json`	`::run_recall`, `::run_fid_*`	§8
`s3_findability_*.json`, `s3_1m.json`, `s3_128k_b8.json`	`::run_million`	§6, Fig. 2
`s2_e2e_k32.json`	`::s2_e2e --topk 32 --out s2_e2e_k32.json`	Table 2 (leading col.)
`s4_512k_chunked.json`, `s4_1m_{near,far}.json`	`::run_debug_1m`	§8.4
`s5_tiered.json`	`::run_tier_{prefill,decode}`	§10
`s5_tiered_128k.json`	`::run_tier_prefill` and `::run_tier_decode`, each with `--n 131072` (decode adds `--out s5_tiered_128k.json`)	§10
`s6_*.json`	`::run_baselines`; `::run_quest_fix` (32K quest); `::run_stream_128k`; `::s10_quest_v2` (128K quest); `::run_stream_matched`	Table 4
`s7_*.json`	`::run_families`	§8.2
`s8_cert_{32k,128k}.json`	`::run_certs`	§8.4
`s9_longgen_*.json`	`::run_longgen`	§8.2
`s10_*.json`	`::s10_*` (review-revision runs)	§8.2, §8.4
`s10_chat_long.json`	`::s10_chat_1m`	§8.4
`model_fit.json`, `pooled_agreement.json`	`analysis.py`	§9, §8.3
`cost_ledger.json`	maintained per run	compute stmt.

Table 6. Number ↔ artifact map. Pre-registered predictions (committed before S1–S3) in `preregistration/predictions.json`, including the post-S0 selector amendment. Result files prefixed with an underscore — and the suffix-marked `s6_quest16_131072_prefix_bug.json` — are quarantined/superseded (kept for audit, excluded from every pool and table; each carries a note).