DON'T DROP TOKENS, DIM THEM: BIT ALLOCATION FOR MULTI-TURN KV-CACHE COMPRESSION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic language models spend much of their wall-clock time paused, waiting for tool responses or user input, while their KV caches stay in GPU memory or are offloaded and read back on resume. Which compression of the paused cache best preserves answer quality over the turns that follow has no settled answer: the two training-free families, token selection and quantization, are built and evaluated for a single prompt answered immediately. Token selection commits a keep-set before the queries arrive, so methods that perfectly recover a planted needle collapse on multi-turn conversation retrieval tasks. Quantization keeps every token but spends its budget uniformly, and its multi-turn retrieval collapses at 2 bits. We trace both failures to one cause: precision is uniform and token selection is binary. We therefore propose **SOFT-BIT KV**, which gives every token its own bit-width instead of choosing which tokens survive: a small MLP scores each cached token from the question's per-head attention, and reverse water-filling spends a measured-byte budget over those scores, once at the pause. Averaged over four models, SOFT-BIT KV retains 94% of full-cache accuracy on multi-turn SCBench at $5.2\times$ compression, compared with 85% for TurboQuant and 72% for the best token-dropping method. At $\approx 11\times$ compression, SOFT-BIT KV retains 99% on single-turn needle retrieval and 65% on RULER, compared with 61% and 56% for the best token-dropping method, and 38% and 23% for TurboQuant. At the system level, it reduces the paused cache of a 98k-token Llama-3.1-8B trajectory by 81% and accelerates host-to-device transfer upon resumption by $5.2\times$.

1 INTRODUCTION

The key–value (KV) cache makes autoregressive decoding cheap and long contexts expensive: it grows linearly with the context and is re-read at every generated token, at 128k tokens it rivals the memory of the weights. Two families of training-free methods shrink it. *Token selection* keeps a subset of tokens at full precision and evicts the rest, ranking them by position (Xiao et al., 2024), accumulated attention (Zhang et al., 2023; Liu et al., 2023), or the attention paid by the tail of the prompt (Li et al., 2024; Cai et al., 2024; Feng et al., 2024). *Quantization* keeps every token at reduced precision, using outlier-aware low-bit codes (Liu et al., 2024; Hooper et al., 2024), an error-correcting residual (Kang et al., 2024), or near-optimal random-rotation coders (Zandieh et al., 2024; 2025). The first family decides *which* tokens survive; the second decides *how precisely* each one survives.

Both families are evaluated in a regime that agentic deployments have left behind: one prompt, one question, decoded immediately. Agentic models call tools and hold multi-turn conversations (Yao et al., 2023; Xu et al., 2023; Kim et al., 2024), so a trajectory spends much of its wall-clock *paused*, waiting for a tool result or the user's next message. Its KV cache sits in GPU memory while it waits, which raises two questions. **First, what should be done with a paused cache?** Serving stacks today keep it resident, drop it and re-prefill, or offload it to host DRAM (Kwon et al., 2023; Sheng et al., 2023; Li et al., 2025a); all three schedule an object whose size is taken as given. **Second, across many multi-turn agentic conversations, which way of compressing that object best preserves answer quality?** The pause is the cheapest moment to compress, since the trajectory is idle, and the hardest one to compress well, since the compressor must commit before it knows what the remaining turns will ask.

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.

Under that constraint neither family survives, for opposite reasons (§3). Token selection looks solved on the single-turn needle probe: SnapKV and PyramidKV recover the planted fact at 100% under a $4\times$ budget on every model we test, yet on SCBench (Li et al., 2025b), where the queries arrive after the cache is compressed, they score 4.0% on key–value retrieval against 78.4% with the full cache. A token evicted at the pause is unrecoverable for every later turn. Quantization keeps every token and is correspondingly stronger on multi-turn retrieval, but it spends the budget uniformly, protecting the one token holding the answer no better than the thousands of filler tokens around it. Its multi-turn retrieval loses 77% from 3 to 2 bits, and at 1 bit it has no lower width left.

We trace both failures to the same cause: precision is uniform, and selection is binary. Robustness across turns requires keeping every token, because any token may be queried later; a high ratio requires an average precision below one bit-width. Both hold only if precision varies across tokens. SOFT-BIT KV therefore allocates per-token precision instead of deciding per-token selection. At the pause it scores each cached token with a small MLP over the question's per-head attention, then water-fills those scores onto a few bit-widths under a byte budget. Eviction becomes the zero-bit corner of this rule, and a token the score under-ranks is degraded rather than deleted.

We evaluate at matched measured bytes on three benchmarks and five models spanning dense, interleaved local-global and hybrid-linear attention, SOFT-BIT KV is the best compressed method on SCBench multi-turn at $5.2\times$ and on RULER at the 1-bit floor (four of four models each), and the only method that retrieves in all three needle regimes on all five models at $11.2\times$. It cuts resume transfer, HBM residency and decode reads by the same ratio, shrinking a 98k-token Llama-3.1-8B cache by 81% (§5).

Contributions.

- We show that the single-turn retrieval test the field reports does not predict multi-turn behaviour, and separate the two mechanisms responsible: a keep-set committed before the queries arrive (token selection), and a uniform bit budget under which multi-turn retrieval collapses at 2 bits (§3).
- We formulate pause-time KV compression as rate–distortion bit allocation under a *measured*-byte budget, where the allocation must be committed at the pause, before the queries it will serve arrive. SOFT-BIT KV solves it by water-filling bits over each token's predicted future attention, once per trajectory, for under 0.1% of one prefill (§4).
- At matched measured bytes across five models, SOFT-BIT KV leads on SCBench multi-turn, on a three-regime needle probe and on RULER at the 1-bit floor, and cuts resume transfer, HBM residency and decode reads by the same ratio (§5).

2 RELATED WORK

Training-free KV-cache compression splits on what the budget buys: *which* tokens survive, or *how precisely* each one survives.

Token selection. The first family keeps a subset of the cache in full precision and evicts the rest, differing in the ranking. StreamingLLM (Xiao et al., 2024) is positional, keeping attention sinks plus a recent window. H2O (Zhang et al., 2023) and Scissorhands (Liu et al., 2023) rank by accumulated attention, retaining the heavy hitters that past queries attended to. SnapKV (Li et al., 2024) makes the rank query-aware, scoring the prompt with the attention of a short observation window at its tail, and the adaptive-budget variants redistribute that same total across layers (Cai et al., 2024), heads (Feng et al., 2024), or a menu of per-head policies (Ge et al., 2024). A second line selects at *decode* time: Quest (Tang et al., 2024) and Double Sparsity (Yang et al., 2024b) bound page- or channel-level scores with the current query and skip the rest, SALS (Mu et al., 2025) scores tokens in a low-rank latent space, and InfiniGen (Lee et al., 2024) speculates the next layer's important keys to prefetch them from host memory. The distinction matters here: decode-time sparsity saves per-step compute and bandwidth but still *stores* the whole cache, whereas a paused trajectory must shrink the stored object. Our four dropping baselines are therefore the prefill-time ones, all of which make a hard keep/drop decision at the pause. That decision is made before the queries are known: to be safe the keep-set would have to predict the union of what every later turn attends to, which on multi-turn retrieval covers nearly the whole context (§3).

Table 1: **Multi-turn retrieval at $5.2\times$ compression.** SCBench accuracy (%) on tasks whose queries arrive after the cache is compressed. The same token-dropping methods retrieve at 100% on the single-turn needle probe (Table 3).

Method	SCBench multi-turn		
	Retr.KV	RepoQA	R.+KV
Llama-3.1-8B-Instruct			
FullKV	78.4	50.7	66.1
SnapKV	4.0	11.4	10.7
PyramidKV	7.0	12.5	10.7
H2O	0.0	10.0	5.7
StreamingLLM	7.6	11.1	7.7
TurboQuant	**46.2**	**39.2**	**53.6**
Qwen3.5-9B-Base			
FullKV	10.4	10.0	30.5
SnapKV	3.0	6.6	11.1
PyramidKV	3.0	5.0	10.2
H2O	3.0	6.6	11.1
StreamingLLM	1.8	2.7	0.9
TurboQuant	**11.8**	**9.3**	**28.9**

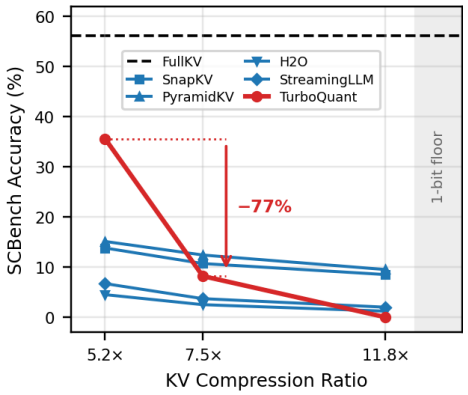

Figure 1: **Uniform precision degrades at higher compression ratios.** We compare SCBench multi-turn accuracy on Llama-3.1-8B across measured compression ratios for quantization and token-dropping methods.

Quantization. The second family retains every token and lowers the width of its code. KIVI (Liu et al., 2024) quantizes keys per channel and values per token at 2 bits; KVQuant (Hooper et al., 2024) adds pre-RoPE key quantization, non-uniform levels and outlier isolation; GEAR (Kang et al., 2024) restores a low-rank-plus-sparse error term on top of a coarse code; QJL (Zandieh et al., 2024) applies a Johnson–Lindenstrauss projection so 1-bit keys still give an unbiased inner product; and TurboQuant (Zandieh et al., 2025) rotates each vector randomly and codes it with a near-MSE-optimal Lloyd–Max quantizer, training-free and state of the art at matched bits. We use TurboQuant as both our coder and our quantization baseline, so every comparison in §5 isolates the allocation rather than the coder. These coders assign one bit-width to the whole cache, and that uniformity is how they fail: spending the budget evenly collapses multi-turn retrieval at 2 bits (§3).

Earlier work explored mixed-precision caches (Yang et al., 2024a; Duanmu et al., 2024) and importance-aware bit assignment (Yang et al., 2024a; He et al., 2024; Dong et al., 2024). These methods allocate for the *current* decode step with the query in hand, whereas a paused trajectory must commit one allocation before the remaining queries exist. The commitment under which §3 shows prefill-time scores fail. We therefore pose pause-time allocation as rate–distortion under a *measured*-byte budget, solved over a tier set (§4.2).

3 Motivation

A pause fixes what a compressor is allowed to know. Write C for the cache of a prefilled context and $q_1, \ldots, q_T$ for the queries the remaining turns will ask. A pause-time compressor applies one map $C \mapsto \hat{C}$, every later turn is answered from $\hat{C}$, and $q_2, \ldots, q_T$ are unavailable when the map is chosen. We therefore compare the two families at matched measured ratios in two settings that differ only in whether the question is visible at compression: the needle probe (§5.2), where it is in the prompt, and SCBench multi-turn (§5.1), where the questions arrive after the cache is committed.

Token selection passes the single-turn test and fails the multi-turn one. When the question is visible at compression, token selection works: under a $4\times$ budget SnapKV and PyramidKV recover the planted fact at 100% in the single- and multi-needle regimes on all five models (Table 3). When the questions arrive afterwards, it collapses (Table 1): on Retr.KV no dropping method retains a third of full-cache accuracy on either model, and on Llama-3.1-8B StreamingLLM, which never sees the query, still outperforms SnapKV there. A better ranking cannot fix this, because each turn queries a different fact, so the tokens some future turn will need cover essentially the whole context.

Quantization survives the commitment but not the ratio. Keeping every token means making no irreversible decision, and Table 1 shows what that is worth: at $5.2\times$ TurboQuant retrieves 46.2%

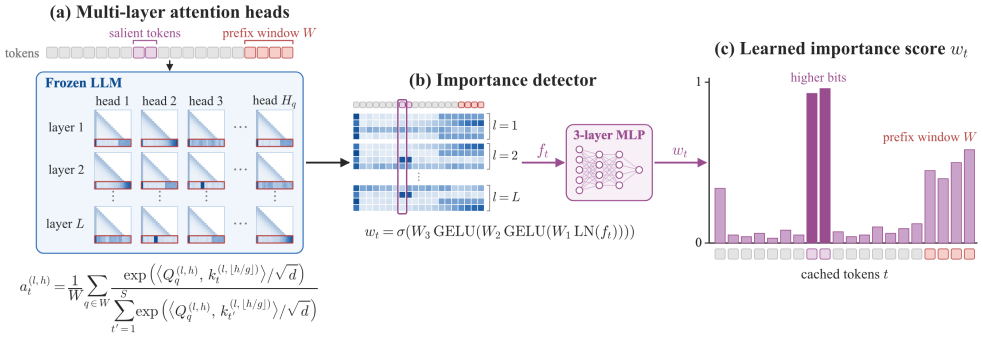

Figure 2: **The learned importance detector.** (a) Per-head attention that the question window $\mathcal{W}$ pays each cached token (Eq. (1)); (b) a three-layer MLP maps the stacked features f_t to w_t (Eqs. (2)–(3)); (c) salient tokens get a high w_t and hence more bits.

on Llama Retr.KV where the entire dropping family stays below 7.6. What it cannot absorb is the budget: from 3 to 2 bits its accuracy on the same tasks falls by 77%, below SnapKV and PyramidKV (Figure 1).

These results indicate what a pause-time compressor needs. Token selection shows that every token must stay readable, since a later turn may query any of them. Quantization shows that the average precision must fall low enough for high ratios, which uniform bit-widths cannot reach without collapsing. Both hold only if precision varies across tokens, which also lets an imperfect importance estimate degrade a token rather than delete it.

4 METHODOLOGY

At a pause, SOFT-BIT KV turns a byte budget into a per-token bit-width in three steps. §4.1 scores each cached token by the attention the question pays it; §4.2 converts those scores into bit-widths by rate–distortion allocation; and §4.3 shows that the resulting byte count sets the three costs a paused trajectory pays. Appendix A collects the notation used throughout the paper.

4.1 A LEARNED IMPORTANCE SCORE FROM DETECTOR

At the pause the cache already holds the question q_1, so the attention q_1 pays each cached token is the importance signal. Averaging that attention over heads is not the best strategy: only a few heads carry the retrieval signal, and they are not the heads that attend most (Appendix H). We therefore learn the combination via a trained detector (Figure 2).

Attention window. Let $\mathcal{W}$ be the *question window*, the last W prefix positions, which hold the first question q_1 at the end of the prefill. The attention that cached token t receives from that window at layer l and query head h is

$$a_t^{(l,h)} = \frac{1}{W}\sum_{q\in\mathcal{W}} \alpha_{q,t}^{(l,h)}, \qquad \alpha_{q,t}^{(l,h)} = \frac{\exp\left(\langle Q_q^{(l,h)},\, k_t^{(l,\lfloor h/g \rfloor)}\rangle/\sqrt{d}\right)}{\sum_{t'=1}^{S}\exp\left(\langle Q_q^{(l,h)},\, k_{t'}^{(l,\lfloor h/g \rfloor)}\rangle/\sqrt{d}\right)}, \tag{1}$$

where the prefix holds S cached tokens, $Q_q^{(l,h)}$ is the query vector at position q, $k_t^{(l,u)}$ the key of token t in KV head u, d the head dimension, $g = H_q/H$ the grouped-query group size over H_q query and H KV heads, and $\alpha_{q,t}^{(l,h)} \in [0,1]$ the attention weight that position q places on token t. Eq. (1) is a column mean of a $W \times S$ *slice* of the attention matrix ($W = 64$ rows), not of the full $S \times S$ matrix, so it is accumulated in a side pass that leaves the attention kernel untouched (Appendix I).

Detector. Stacking Eq. (1) over all (l,h) pairs and appending the token's mean key norm $\tilde{n}_t$ gives the feature vector

$$f_t = \left[\log(1+\kappa\, a_t^{(l,h)})\right]_{l\le L,\, h\le H_q} \oplus\ \tilde{n}_t\ \in\ \mathbb{R}^{d_f}, \qquad d_f = L\,H_q + 1, \tag{2}$$

where $\oplus$ denotes concatenation and κ rescales before the logarithm so the small attention values that carry the retrieval signal are not flattened. A three-layer MLP of hidden width d_{mlp} maps f_t to an importance weight,

$$s_t = W_3\,\mathrm{GELU}\big(W_2\,\mathrm{GELU}(W_1\,\mathrm{LN}(f_t))\big), \qquad w_t = \sigma(s_t) \in (0,1), \tag{3}$$

with $W_1 \in \mathbb{R}^{d_{\text{mlp}} \times d_f}$, $W_2 \in \mathbb{R}^{d_{\text{mlp}} \times d_{\text{mlp}}}$, $W_3 \in \mathbb{R}^{1 \times d_{\text{mlp}}}$ and biases omitted, LN layer normalisation and σ the logistic function.[1]

Training. Each training example is a retrieval prompt: one to four facts planted in the far-back region $\mathcal{F}$, and a question asking for one of them, with templates, separators and value types randomised. Writing $y_t = 1$ for tokens of the queried answer, we minimise a class-balanced cross-entropy over $\mathcal{F}$,

$$\mathcal{L} = -\frac{1}{|\mathcal{F}|} \sum_{t \in \mathcal{F}} \Big[\rho\, y_t \log \sigma(s_t) + (1 - y_t) \log\big(1 - \sigma(s_t)\big)\Big], \qquad \rho = \frac{|\mathcal{F}| - \sum_{t \in \mathcal{F}} y_t}{\sum_{t \in \mathcal{F}} y_t}, \tag{4}$$

where ρ is the positive-class weight correcting the label imbalance.

4.2 From Scores to Bit-Widths: Rate–Distortion Allocation

A b-bit quantizer applied to a unit-norm vector has error proportional to 4^{-b}, so an allocation $\mathbf{b} = (b_1, \ldots, b_S)$ that gives token t a width b_t costs

$$\min_{\mathbf{b}} \;\; D(\mathbf{b}) = \sum_{t=1}^{S} \underbrace{w_t}_{\text{importance}} \cdot \underbrace{4^{-b_t}}_{\text{error at } b_t \text{ bits}} \qquad \text{subject to} \qquad \frac{1}{S} \sum_{t=1}^{S} b_t \;\le\; \bar{b}, \tag{5}$$

where the budget $\bar{b}$ fixes the compression ratio r of §4.3. Weighting the error by w_t is what makes the objective *functional*: bits follow what the query will read, not the magnitude of what is stored.

Optimum. Relaxing b_t to the reals, a Lagrange multiplier λ on the budget gives

$$w_t\, 4^{-b_t} = \lambda \qquad \text{that is} \qquad b_t = \Big[\log_4 \frac{w_t}{\lambda}\Big]_+, \qquad [x]_+ = \max(x, 0), \tag{6}$$

with λ set so the budget binds: every kept token carries the *same* error λ, and tokens whose importance falls below λ get no bits. Two properties follow. Bits grow logarithmically in importance, so a score wrong by a factor of four costs one bit; and eviction is the zero-bit corner of this rule rather than a different mechanism.

Tiers. Coders pack efficiently only at a few widths, so we restrict b_t to $\mathcal{B} = \{1, 2, 8\}$ and solve the integer version exactly. Promoting token t from width β to the next width β' removes $w_t(4^{-\beta} - 4^{-\beta'})$ of error for $\beta' - \beta$ extra bits, a gain per bit of

$$\Delta_t(\beta \to \beta') = w_t\, \frac{4^{-\beta} - 4^{-\beta'}}{\beta' - \beta}. \tag{7}$$

Starting every token at the lowest width and repeatedly promoting the largest Δ_t until the budget is spent is optimal, since each promotion lowers that token's own gain; one heap pass over the S tokens suffices. A few tokens end at 8 bits and the bulk at 1–2: the average meets the budget and nothing is deleted. Each vector is then coded once by TurboQuant (Zandieh et al., 2025) at its assigned width, $\hat{x}_t = Q_{b_t}^{-1}(Q_{b_t}(x_t))$.

4.3 System Efficiency

A paused cache does not stay on the accelerator: serving stacks offload it to host memory and read it back on resume (Kwon et al., 2023; Sheng et al., 2023; Li et al., 2025a). The allocation of §4.2

[1] In our experiments $d_{\text{mlp}} = 128$, so the detector holds 61K parameters on Qwen2.5-1.5B and 34K on Gemma-4-12B; the base model stays frozen and serves only as a feature extractor.

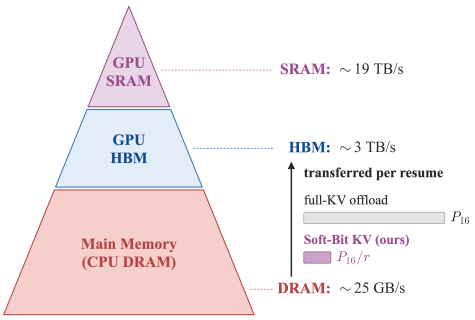

Figure 3: **The hierarchy a paused trajectory must cross.** At a pause the cache is offloaded to host DRAM and read back over the slowest link in the hierarchy. A full-KV offload moves P_{16} on every resume; soft-bit allocation moves $P(\mathbf{b}) = P_{16}/r$.

Algorithm 1 SOFT-BIT KV. Encoding runs once, every later turn is transfer and attention only.

Require: prefix C, tiers $\mathcal{B}$, budget $\bar{b}$; $\mathcal{I}_\beta = \{t : b_t = \beta\}$

TURN 1 *(once per trajectory)*

for $l = 1, \dots, L$ **do**
  $K_l, V_l \leftarrow \textsc{Prefill}_l(C)$ ▷ HBM, fp16
  $F \leftarrow F \,\|\, \textsc{AttnSlice}(Q_l, K_l)$ ▷ Eq. (1)
end for
$w \leftarrow \sigma(\mathrm{MLP}_\theta(F))$; $\mathbf{b} \leftarrow \textsc{WaterFill}(w, \bar{b}, \mathcal{B})$ ▷ Eqs. (3),(7)
for $l = 1, \dots, L$ **do**
  for all $\beta \in \mathcal{B}$ **do**
    $Z_l[\beta] \leftarrow \textsc{Pack}\, Q_\beta\big((K_l, V_l)[\mathcal{I}_\beta]\big)$ ▷ **encode**
  end for
  $\textsc{Copy}(Z_l \to \text{DRAM})$; $\textsc{Free}(K_l, V_l)$ ▷ $P(\mathbf{b})/L$
end for

TURN $k \geq 2$ *(no scoring, no encoding)*

$\textsc{Copy}(\text{DRAM} \to Z)$ ▷ DRAM→HBM, $P(\mathbf{b})$
$o \leftarrow \textsc{MixedPrecAttn}(q_k, Z)$ ▷ **decode** in SRAM
append at $\max \mathcal{B}$; copy delta ▷ $O(\text{turn})$

therefore fixes the bytes the trajectory moves, holds and re-reads. Let $\mathcal{C}(b)$ be the bytes a b-bit code occupies for one token in one stream. Summing over the $2LH$ streams (keys and values, L layers, H KV heads), $P(\mathbf{b}) = 2LH \sum_{t=1}^{S} \mathcal{C}(b_t)$, where $P(\mathbf{b})$ is the compressed cache size in bytes, $P_{16} = 4LHSd$ is the fp16 size, and $r = P_{16}/P(\mathbf{b})$ is the realised compression ratio reported throughout.

One quantity, three costs. $P(\mathbf{b})$ is charged three times per turn: once to move the cache back across the host link of bandwidth B_{link}, once as HBM residency while the turn is active, and once per generated token as decode attention re-reads it, for the n tokens a turn emits,

$$\underbrace{T_{\text{resume}} = P(\mathbf{b})/B_{\text{link}}}_{\text{transfer}}, \qquad \underbrace{M_{\text{HBM}} = P(\mathbf{b})}_{\text{residency}}, \qquad \underbrace{R_{\text{dec}} = n\,P(\mathbf{b})}_{\text{decode reads}}, \tag{8}$$

so one ratio r divides all three (Figure 3). Resume is bandwidth-bound: reloading costs P_{16}/B_{link}, against the $\Theta(LS^2d)$ FLOPs of a re-prefill.

Algorithm 1 realises this for a single allocation. Scoring and allocation run once, after the turn-1 prefill, and the encoder works layer by layer, so the full cache is never resident beside its own codes. The host then holds an immutable object: every resume reads $P(\mathbf{b})$, and every pause appends only the new turn's question and answer. A variant, SOFT-BIT KV (reload), instead recomputes the allocation at each pause from that turn's query; Appendix J gives its algorithm and accounting.

Decoding reads the packed codes. These counts hold only if the cache is never decoded in bulk, so dequantisation is fused into the attention kernel: a tile is unpacked in registers, used, and discarded, while HBM holds packed codes throughout. Neighbouring tokens may sit at different widths, so we group the cache by tier, run one uniform-width pass per group, and merge the passes with the online softmax

$$m = \max_\beta m^{(\beta)}, \qquad \ell = \sum_\beta e^{m^{(\beta)} - m} \ell^{(\beta)}, \qquad o = \frac{1}{\ell} \sum_\beta e^{m^{(\beta)} - m}\, o^{(\beta)}, \tag{9}$$

where $m^{(\beta)}, \ell^{(\beta)}, o^{(\beta)}$ are the running maximum, softmax normaliser and output of bucket β. The merge is exact: FlashAttention already rescales this way across tiles (Dao et al., 2022; Dao, 2024), and softmax does not depend on the order of the keys, so grouping by tier changes nothing. Appendix K gives the kernel.

5 EXPERIMENTS

§5.1 answers the question the paper is about—which way of compressing a paused cache best preserves answer quality across multi-turn conversations—on SCBench (Li et al., 2025b), whose queries arrive after the cache is committed. §5.2 tests whether that advantage is specific to multi-turn reuse and whether it holds across compression ratios, on the single-turn needle probe released with TurboQuant (Zandieh et al., 2025). §5.3 compares against TurboQuant at the extreme end of that sweep, where a uniform quantizer is forced to its 1-bit floor, on RULER (Hsieh et al., 2024). §5.4 leaves quality aside and measures the system cost of a paused trajectory. Appendix C puts the three benchmarks on one scale as *retained full-cache accuracy*, and Appendix B gives the full experimental settings.

5.1 MULTI-TURN CONVERSATION

SCBench (Li et al., 2025b) stresses the paused-cache regime directly: one long context is prefilled *once*, then serves up to five queries, with each turn's question and answer appended to the cache. Table 2 reports the outcome. SOFT-BIT KV outperforms baselines across four models. On Llama-3.1-8B it reaches 39.5 average accuracy against 37.9 for TurboQuant, 36.6 for QAQ and 28.8 for the best token-dropping method, and on Qwen3-4B 27.8 against 19.4, 18.2 and 22.5. Averaged over the four models it retains 94% of full-cache accuracy, against 85% for TurboQuant and 72% for the best dropping method (Appendix C). Exact string retrieval separates the families most sharply: on Retr.KV SOFT-BIT KV scores 54.4 on Llama against 46.2 for TurboQuant and at most 7.6 for a dropping method, and on Qwen3-4B it matches the full cache (19.0 vs. 19.0) where no baseline exceeds 3.6. On Gemma-4-12B, where uniform quantization is itself near-lossless (TurboQuant 30.5 vs. FullKV 30.7), SOFT-BIT KV still leads at 32.3. Re-allocating at every pause adds a further 0.8–3.5 points, bounding what a better pause-time score could buy.

5.2 NEEDLE-IN-A-HAYSTACK RETRIEVAL

Two readings of §5.1 are still open: that the margin comes from multi-turn state rather than from the allocation, and that it holds only at 5.2×. The needle probe tests both: it is single-turn, with the question in the prompt when the cache is compressed, and we sweep three matched KV-memory ratios (≈4, 8 and 11.2×) over three escalating regimes, with identical prompts and greedy decoding. Table 3 reports the outcome. The advantage is not an artefact of multi-turn state, and it widens with the ratio. At 11.2× SOFT-BIT KV averages 98.5% HIT over the five models and three regimes, against 61.1% for SnapKV and 37.8% for TurboQuant at its nearest point, and it is the only method that retrieves in all three regimes on all five models—100% everywhere except Llama's 2-hop chain at 78%. The two families fail in the ways §3 predicts: token dropping mostly holds the single-needle regime but solves the 2-hop chain only on Gemma-4-12B, while TurboQuant holds every regime it supports at 4 bits and collapses on four of five models at 1 bit.

5.3 RULER

The sweep ends where TurboQuant collapses, so we make that end point a setting of its own: at 11.8× a uniform quantizer is pinned to 1 bit, and the question is whether allocation still buys accuracy at that budget. RULER (Hsieh et al., 2024) isolates it—13 synthetic tasks at a *fixed* context length, so only the compressor varies. We run the pre-generated RULER at 4,096 tokens with 100 examples per task, scoring with `string_match_all`, and place every compressed method at the matched 11.8×. Table 4 reports the outcome. At the 1-bit floor SOFT-BIT KV is the best compressed method on all four models, averaging 59.9 against 51.7 for the best token-dropping method, 21.2 for TurboQuant and 20.9 for QAQ.

5.4 MEMORY SAVING AND SPEEDUP

Table 5 instantiates the cost model of §4.3 on Llama-3.1-8B ($L=32$, $H=8$, $d=128$) at SCBench's mean prefill $S = 98$k, where the fp16 cache occupies $P_{16} = 12.9$ GB, and pairs it with decode latency measured on one H100 under CUDA graphs over all layers, against the serving default of offloading the fp16 cache and reading it back on every resume.

Table 2: Evaluation of KV cache compression methods on multi-turn SCBench, grouped by SCBench's four capability categories (Li et al., 2025b)(accuracy %, mean over turns).

Method	Comp.	String Retrieval			Semantic Retrieval				Global Information			Multi-tasking		Avg
		Retr.KV	Retr.PS	VT	RepoQA	En.QA	Zh.QA	Choice	M.Shot	Summ.	MF	S.+N.	R.+KV	
Llama-3.1-8B-Instruct (128k context; mean prefill 98k tokens)														
FullKV	1×	78.4	40.4	45.0	50.7	20.9	21.6	68.2	49.3	40.6	21.6	56.8	66.1	46.6
Matched to TurboQuant at 5.2×														
SnapKV	5.2×	4.0	1.8	41.2	11.4	16.8	19.7	62.9	48.9	39.2	20.0	56.8	10.7	27.8
PyramidKV	5.2×	7.0	1.0	**44.4**	12.5	17.2	**21.0**	63.8	48.9	**39.4**	**22.6**	**57.3**	10.7	28.8
H2O	5.2×	0.0	0.0	6.8	10.0	19.8	20.3	60.7	51.9	38.9	16.6	43.4	5.7	22.8
StreamingLLM	5.2×	7.6	4.6	2.3	11.1	9.9	16.8	52.8	51.5	32.6	15.4	31.4	7.7	20.3
TurboQuant	5.2×	46.2	9.8	29.1	39.2	**20.7**	18.3	**72.1**	73.0	36.8	8.4	47.3	**53.6**	37.9
QAQ	5.2×	44.6	8.4	27.6	37.9	19.6	17.4	70.8	71.6	35.5	7.2	45.9	52.1	36.6
Soft-bit KV (ours)	5.2×	**54.4**	**18.2**	28.1	**43.0**	19.7	18.8	69.0	**73.7**	36.8	11.0	47.9	53.0	**39.5**
(query-conditioned soft-bit, keeps full-precision KV)														
Soft-bit KV (reload)	5.2×	71.6	27.2	32.9	48.4	20.7	22.4	68.2	70.4	37.0	14.0	54.9	48.4	43.0
Qwen3-4B-Instruct-2507 (32k eval context; dense full-attention)														
FullKV	1×	19.0	19.2	21.4	28.0	22.0	0.7	62.0	73.3	27.2	6.6	37.3	39.3	29.7
Token dropping, matched to TurboQuant at 5.2× (K3/V3)														
SnapKV	5.2×	3.0	1.6	**23.0**	9.1	18.6	**0.9**	58.4	**73.3**	**27.6**	**6.8**	36.2	12.0	22.5
PyramidKV	5.2×	3.6	1.2	22.7	6.1	16.5	0.8	58.1	73.0	27.3	5.6	35.4	10.2	21.7
H2O‡	5.2×	3.0	1.6	**23.0**	9.1	18.6	**0.9**	58.4	**73.3**	**27.6**	**6.8**	36.2	12.0	22.5
StreamingLLM	5.2×	3.0	1.8	4.4	3.6	10.7	**0.9**	53.2	71.1	26.0	3.6	35.4	1.8	18.0
TurboQuant	5.2×	2.6	0.4	11.7	11.1	15.9	0.8	48.0	64.1	24.8	4.0	31.8	17.3	19.4
QAQ	5.2×	2.4	0.3	10.8	10.2	14.9	0.7	46.8	62.5	23.9	3.5	30.7	16.2	18.2
Soft-bit KV (ours)	5.2×	**19.0**	**8.8**	21.1	**25.5**	**21.4**	0.8	**61.6**	71.5	27.3	5.8	**37.1**	**33.9**	**27.8**
Soft-bit KV (reload)	5.2×	19.0	17.8	14.0	28.0	20.3	3.5	58.1	70.7	27.7	11.2	36.2	39.8	28.9
Gemma-4-12B-it (32k eval context; interleaved local/global attention)														
FullKV	1×	17.8	21.8	12.4	27.7	22.3	0.6	61.1	69.6	16.3	55.6	24.6	38.2	30.7
Whole-model quantization (every token kept)														
QAQ	5.2×	17.9	18.3	13.0	25.5	18.2	0.5	63.8	67.9	15.8	55.6	23.7	35.5	29.6
TurboQuant	5.2×	18.6	19.0	13.6	26.6	19.1	**0.5**	65.0	69.3	16.4	56.8	24.6	36.8	30.5
Soft-bit KV (ours)	5.2×	**18.8**	**19.6**	**13.8**	**28.4**	**22.1**	**0.5**	**65.6**	70.4	**18.0**	**57.5**	**25.2**	**37.5**	**32.3**
Soft-bit KV (reload)	5.2×	19.2	20.6	14.3	30.2	23.4	0.5	66.8	73.5	19.0	59.2	26.0	38.6	33.1
Token dropping, matched to TurboQuant at 5.2×														
SnapKV	5.2×	3.8	3.4	13.3	11.6	16.8	**0.5**	59.8	69.3	16.2	51.6	23.5	13.4	23.6
PyramidKV	5.2×	3.6	2.2	12.6	9.5	15.9	**0.5**	54.1	71.5	16.1	49.8	21.5	10.7	22.3
H2O‡	5.2×	3.8	3.4	13.3	11.6	16.8	**0.5**	59.8	69.3	16.2	51.6	23.5	13.4	23.6
StreamingLLM	5.2×	3.0	3.4	3.9	3.0	9.6	0.4	50.6	**74.4**	14.8	40.8	22.5	1.8	19.0
Qwen3.5-9B-Base (hybrid linear attention; KV cache in 8/32 full-attention layers)§														
FullKV	1×	10.4	15.2	8.9	10.0	24.1	0.7	44.9	57.0	23.6	29.0	27.2	30.5	23.5
Compression on the 8 full-attention layers, matched to TurboQuant at 5.2× (K3/V3)														
SnapKV	5.2×	3.0	4.6	9.5	6.6	24.3	**0.7**	41.4	63.7	23.1	**27.4**	27.2	11.1	20.2
PyramidKV	5.2×	3.0	4.4	**9.6**	5.0	22.6	0.6	40.5	**65.2**	23.1	26.2	26.4	10.2	19.7
H2O‡	5.2×	3.0	4.6	9.5	6.6	24.3	**0.7**	41.4	63.7	23.1	**27.4**	27.2	11.1	20.2
StreamingLLM	5.2×	1.8	2.8	4.4	2.7	13.3	0.6	45.4	57.0	22.4	17.2	25.3	0.9	16.1
TurboQuant	5.2×	**11.8**	**14.8**	8.1	**9.3**	24.6	**0.7**	43.6	59.3	**23.4**	26.6	**27.5**	**28.9**	23.2
QAQ	5.2×	9.0	9.2	8.1	5.6	**26.7**	0.5	42.3	62.2	23.1	18.9	26.2	23.8	21.3
Soft-bit KV (ours)	5.2×	**11.8**	12.6	8.7	8.3	**26.7**	**0.7**	**45.8**	61.9	23.2	23.8	27.3	28.4	**23.3**
Soft-bit KV (reload)	5.2×	11.0	19.0	18.4	3.1	22.9	0.5	37.4	75.6	23.1	43.8	26.1	25.2	25.5

Table 3: Evaluation of KV cache compression methods on Needle-in-a-haystack HIT (%) on retrieval across five models, organized into three sections by *matched compression ratio*.

Method	Comp.	Llama-3.1-8B-Ins.			Qwen2.5-1.5B-Ins.			Gemma-4-12B-it			Qwen3-4B-Ins.			Qwen3.5-9B-Base		
		sgl	mul	hrd	sgl	mul	hrd	sgl	mul	hrd	sgl	mul	hrd	sgl	mul	hrd
FullKV	1×	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Compression ≈4.0×																
SnapKV	4.0×	**100**	**100**	0	**100**	**100**	0	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	0
PyramidKV	4.0×	**100**	**100**	33	**100**	50	0	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	0
H2O	4.0×	33	0	0	0	0	0	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	0
StreamingLLM	4.0×	0	**100**	0	0	**100**	0	0	**100**	0	0	**100**	0	0	**100**	0
TurboQuant (K4/V4)	3.7×	**100**	**100**	**100**	0†	0†	0†	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
Soft-bit KV (ours)	4.0×	**100**	**100**	**100**	**100**	**100**	**78**	**100**	**100**	**100**	67	**100**	**100**	**100**	**100**	**100**
Compression ≈8.0×																
SnapKV	8.0×	**100**	**100**	33	**100**	50	0	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	0
PyramidKV	8.0×	**100**	**100**	33	**100**	50	0	**100**	**100**	0	**100**	**100**	0	**100**	**100**	0
H2O	8.0×	67	0	0	0	0	0	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	0
StreamingLLM	8.0×	0	50	0	0	50	0	0	**100**	0	0	50	0	0	50	0
TurboQuant (K2/V2)	6.8×	**100**	**100**	**100**	89	**83**	**56**	**100**	**100**	**100**	**100**	**100**	0	**100**	**100**	**100**
Soft-bit KV (ours)	8.0×	**100**	**100**	**100**	**100**	**83**	**56**	**100**	**100**	**100**	33	**100**	**100**	**100**	**100**	**100**
Compression ≈11.2×																
SnapKV	11.2×	**100**	**100**	0	**100**	0	0	**100**	**100**	**100**	**100**	**100**	0	67	50	0
PyramidKV	11.2×	**100**	**100**	0	**100**	50	0	**100**	**100**	0	33	0	0	**100**	50	0
H2O	11.2×	**100**	0	0	0	0	0	**100**	**100**	**100**	**100**	**100**	0	67	50	0
StreamingLLM	11.2×	0	50	0	0	50	0	0	**100**	0	0	50	0	0	50	0
TurboQuant (K1/V1)	11.8×	**100**	**100**	0	0	0	0	**100**	**100**	**100**	0	0	0	67	0	0
Soft-bit KV (ours)	11.2×	**100**	**100**	**78**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**

Memory and transfer. At $\bar{b} = 3$ bits/token ($r = 5.2\times$) the cache occupies 2.5 GB, so a resume moves 2.5 GB rather than 12.9 GB over the 25 GB/s host link: 100 ms instead of 516 ms, saving 416 ms per turn. Re-allocating at every pause keeps 380 ms of that saving in exchange for host memory. **Decode** attention is bandwidth-bound, so reading $P(\mathbf{b})$ instead of P_{16} makes it faster rather than slower: the mixed-precision kernel of Appendix K attends over the compressed cache in

Table 4: Evaluation of KV cache compression methods on RULER benchmark at an aggressive 11.8× compression ratio (1-bit regime).

Method	Comp.	Single NIAH			Multi-key / query / value NIAH					M.hop	Aggregation		QA		Avg.
		S-1	S-2	S-3	MK-1	MK-2	MK-3	MQ	MV	VT	CWE	FWE	QA-1	QA-2	
Llama-3.1-8B-Instruct															
FullKV	1×	100.0	100.0	100.0	100.0	100.0	100.0	99.8	99.8	99.2	99.4	95.3	81.6	65.0	95.4
SnapKV	11.8×	**100.0**	95.0	0.0	93.0	79.0	0.0	89.8	77.8	**93.8**	34.8	81.0	81.2	**62.0**	68.3
PyramidKV	11.8×	**100.0**	95.0	0.0	94.0	83.0	0.0	87.8	71.0	87.0	23.4	75.3	**82.5**	**62.0**	66.2
H2O	11.8×	4.0	4.0	0.0	12.0	29.0	0.0	4.5	7.8	11.8	**68.9**	**94.0**	66.5	55.0	27.5
StreamingLLM	11.8×	10.0	18.0	**5.0**	13.0	9.0	**4.0**	8.2	11.0	6.8	0.7	82.7	78.9	46.0	22.6
TurboQuant	11.8×	6.0	4.0	0.0	5.3	8.7	0.0	0.8	1.1	10.2	25.6	60.4	50.9	41.3	16.5
QAQ	11.8×	5.0	3.0	0.0	4.3	7.4	0.0	0.6	0.9	8.8	23.8	58.2	48.6	39.1	15.4
Soft-bit KV (RH)	11.8×	99.0	**100.0**	2.0	**100.0**	94.0	0.0	**99.5**	57.2	37.6	36.2	83.0	76.0	55.0	64.6
Soft-bit KV (MLP)	11.8×	99.0	**100.0**	2.0	**100.0**	**100.0**	3.0	99.0	**91.0**	64.0	39.2	83.3	71.0	53.0	**69.6**
Gemma-4-12B-it															
FullKV	1×	100.0	100.0	96.0	99.0	99.0	100.0	99.5	96.2	99.8	99.2	17.3	76.3	61.0	88.0
SnapKV	11.8×	**100.0**	**69.0**	3.0	27.0	12.0	**11.0**	20.8	9.5	81.4	**78.9**	**6.3**	71.3	**57.0**	42.1
PyramidKV	11.8×	**100.0**	37.0	2.0	13.0	5.0	0.0	8.8	4.8	12.2	67.8	2.7	66.3	52.0	28.6
H2O‡	11.8×	**100.0**	**69.0**	3.0	27.0	12.0	**11.0**	20.8	9.5	81.4	**78.9**	**6.3**	71.3	**57.0**	42.1
StreamingLLM	11.8×	9.0	12.0	4.0	11.0	7.0	3.0	9.2	8.2	6.0	62.1	1.3	**72.0**	43.0	19.1
TurboQuant	11.8×	73.0	**69.0**	4.0	**59.0**	**56.0**	0.0	**44.8**	35.2	62.4	65.0	2.0	62.7	55.0	45.2
QAQ	11.8×	71.4	67.5	3.7	57.4	54.2	0.0	43.1	33.6	60.7	63.2	1.9	63.5	53.1	44.1
Soft-bit KV (RH)	11.8×	**100.0**	27.0	0.0	34.0	42.0	1.0	8.2	13.8	73.4	39.4	2.3	63.1	49.0	34.9
Soft-bit KV (MLP)	11.8×	**100.0**	**69.0**	**26.0**	56.0	54.0	1.0	20.0	**54.5**	**99.8**	41.9	**6.3**	62.7	52.0	**49.5**
Qwen3-4B-Instruct-2507															
FullKV	1×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.8	100.0	95.9	84.3	86.7	65.0	94.7
SnapKV	11.8×	96.0	86.0	2.0	31.0	5.0	0.0	**77.0**	8.2	75.4	34.8	**54.0**	**79.3**	**60.0**	46.8
PyramidKV	11.8×	73.0	46.0	2.0	11.0	2.0	0.0	30.5	5.5	3.6	6.5	18.3	68.8	52.0	24.6
H2O	11.8×	96.0	86.0	2.0	31.0	5.0	0.0	**77.0**	8.2	75.4	34.8	**54.0**	**79.3**	**60.0**	46.8
StreamingLLM	11.8×	10.0	18.0	5.0	13.0	8.0	**3.0**	10.2	11.0	8.6	**40.2**	46.0	77.0	45.0	22.7
TurboQuant	11.8×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	13.3	25.0	3.0
QAQ	11.8×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	12.0	23.8	2.7
Soft-bit KV (RH)	11.8×	99.0	15.0	2.0	6.0	0.0	0.0	0.0	1.2	11.8	12.4	34.0	48.7	45.0	21.2
Soft-bit KV (MLP)	11.8×	**100.0**	**99.0**	**72.0**	**95.0**	**57.0**	0.0	53.0	**84.5**	**85.6**	13.8	48.3	71.1	58.0	**64.4**
Qwen3.5-9B-Base															
FullKV	1×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.8	100.0	99.9	95.0	30.3	32.0	88.8
SnapKV	11.8×	96.0	99.0	2.0	80.0	**13.0**	1.0	94.8	26.0	67.4	55.9	54.7	**26.7**	**29.0**	49.7
PyramidKV	11.8×	90.0	97.0	2.0	72.0	10.0	0.0	88.8	17.5	7.0	17.5	10.0	20.7	26.0	35.3
H2O	11.8×	96.0	99.0	2.0	80.0	**13.0**	1.0	94.8	26.0	67.4	55.9	54.7	**26.7**	**29.0**	49.7
StreamingLLM	11.8×	9.0	18.0	**5.0**	13.0	8.0	**3.0**	10.2	11.0	8.2	**57.9**	63.0	22.7	16.0	18.8
TurboQuant	11.8×	16.0	27.0	0.0	43.0	**13.0**	0.0	16.5	6.8	14.6	19.1	**65.0**	20.8	21.0	20.2
QAQ	11.8×	17.6	28.6	0.0	45.2	14.1	0.0	17.8	7.5	15.8	20.4	66.8	22.1	22.4	21.3
Soft-bit KV (RH)	11.8×	95.0	**100.0**	2.0	47.0	3.0	0.0	26.5	9.0	20.2	46.0	42.3	23.1	**29.0**	34.1
Soft-bit KV (MLP)	11.8×	**100.0**	**100.0**	0.0	**98.0**	11.0	0.0	**97.2**	**93.8**	**92.2**	27.8	59.7	19.0	28.0	**55.9**

Table 5: **Cost of a resume** on Llama-3.1-8B at $S = 98$k, with $B_{\text{link}} = 25$ GB/s. "Traffic" is host→device bytes per resume; **Turn** = transfer + n×decode at $n = 150$, with speedup relative to the full-KV offload.

	Bytes			Transfer latency		Decode attention		Turn ($n=150$)	
Design	**Host DRAM**	**Traffic / resume**	**HBM resident**	**per resume**	**saved**	**ms/token**	**speedup**	**latency**	**speedup**
Full-KV offload (serving default)	12.9 GB	12.9 GB	12.9 GB	516 ms	—	4.59	1.00×	1205 ms	1.00×
$r = 5.2\times$									
Soft-bit, single allocation	**2.5 GB**	**2.5 GB**	**2.5 GB**	**100 ms**	**416 ms**	**1.41**	**3.27×**	**311 ms**	**3.87×**
Soft-bit, per-turn allocation	9.2 GB	3.4 GB	3.4 GB	136 ms	380 ms	1.41	3.27×	347 ms	3.47×
$r = 11.8\times$									
Soft-bit, single allocation	**1.1 GB**	**1.1 GB**	**1.1 GB**	**44 ms**	**472 ms**	**1.30**	**3.53×**	**239 ms**	**5.04×**
Soft-bit, per-turn allocation	9.2 GB	2.0 GB	2.0 GB	80 ms	436 ms	1.30	3.53×	275 ms	4.38×

1.41 ms per generated token against 4.59 ms for `flash_attn_with_kvcache` over fp16, a 3.27× speedup, rising to 1.30 ms (3.53×) at 11.8×. Adding the resume to 150 generated tokens, a turn falls from 1205 ms to 311 ms (3.87×) at 5.2× and to 239 ms (5.04×) at 11.8×; per-turn allocation costs about 36 ms more (347 ms, 3.47×).[2]

6 Conclusion

We show that token eviction can discard information needed by later queries, while uniform quantization degrades retrieval at low precision. We introduce Soft-bit KV, which allocates per-token precision under a byte budget while retaining every token. Experiments on SCBench, needle retrieval, and RULER demonstrate improved quality, alongside reduced cache storage and faster decode attention.

[2]The per-turn rows of Table 5 decode the same gathered allocation as the single-allocation rows; their per-turn re-scoring and gather are not included.

REFERENCES

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*, 2024. arXiv:2406.02069.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2307.08691.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2205.14135.

Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. QAQ: Quality adaptive quantization for LLM KV cache. *arXiv preprint arXiv:2403.04643*, 2024.

Haojie Duanmu, Zhihang Yuan, Xiuhong Li, Jiangfei Duan, Xingcheng Zhang, and Dahua Lin. SKVQ: Sliding-window key and value cache quantization for large language models. *arXiv preprint arXiv:2405.06219*, 2024.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. *arXiv preprint arXiv:2407.11550*, 2024. arXiv:2407.11550.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive KV cache compression for LLMs. *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.01801.

Gemma Team, Google DeepMind. Gemma 4. Model card, `https://huggingface.co/google/gemma-4-12B-it`, 2026. Checkpoint `google/gemma-4-12B-it`.

Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. ZipCache: Accurate and efficient KV cache quantization with salient token identification. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2401.18079.

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 (COLM)*, 2024.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. GEAR: An efficient KV cache compression recipe for near-lossless generative inference of LLM. *arXiv preprint arXiv:2403.05527*, 2024.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph E. Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2025a.

Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir H. Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, and Lili Qiu. SCBench: A KV cache-centric analysis of long-context methods. In *International Conference on Learning Representations (ICLR)*, 2025b.

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2404.14469.

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.17118.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. *International Conference on Machine Learning (ICML)*, 2024. arXiv:2402.02750.

Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Junlin Mu, Hantao Huang, Jihang Zhang, Minghui Yu, Tao Wang, and Yidong Li. SALS: Sparse attention in latent space for KV cache compression. *arXiv preprint arXiv:2510.24273*, 2025.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

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

Qwen Team. Qwen3.5. Model card, `https://huggingface.co/Qwen/Qwen3.5-9B-Base`, 2026. Checkpoint `Qwen/Qwen3.5-9B-Base`.

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 (ICML)*, 2023.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference. *International Conference on Machine Learning (ICML)*, 2024. arXiv:2406.10774.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2309.17453.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023.

June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. No token left behind: Reliable KV cache compression via importance-aware mixed precision quantization. *arXiv preprint arXiv:2402.18096*, 2024a.

Shuo Yang, Ying Sheng, Joseph E. Gonzalez, Ion Stoica, and Lianmin Zheng. Post-training sparse attention with double sparsity. *arXiv preprint arXiv:2408.07092*, 2024b.

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

Amir Zandieh, Majid Daliri, and Insu Han. QJL: 1-bit quantized JL transform for KV cache quantization with zero overhead. *arXiv preprint arXiv:2406.03482*, 2024.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. TurboQuant: Online vector quantization with near-optimal distortion rate. *arXiv preprint arXiv:2504.19874*, 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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2306.14048.

A NOTATION

For clarity, we summarise the symbols used throughout the paper.

Pause and setting.

S	Number of tokens in the prefilled prefix (the cache being compressed).
T	Number of turns the paused trajectory still has to serve.
$q_1, \dots, q_T$	Queries of those turns; only q_1 may be visible at the pause.
$C,\ \hat{C}$	Cache of the prefilled context, and its compressed form.
$\mathcal{W},\ W$	Question window (the last W prefix positions) and its length; $W = 64$.

Model.

L	Number of layers.
$H_q,\ H$	Query heads and KV heads per layer.
g	Group size $g = H_q/H$; query head h reads KV head $\lfloor h/g \rfloor$.
d	Head dimension.
$Q_q^{(l,h)}$	Query vector at position q, layer l, query head h.
$k_t^{(l,u)},\ v_t^{(l,u)}$	Key and value of token t at layer l, KV head u.

Importance score (§4.1).

$\alpha_{q,t}^{(l,h)}$	Attention weight that query position q places on cached token t.
$a_t^{(l,h)}$	Attention token t receives from the question window, averaged over its W rows.
$\tilde{n}_t$	Mean key norm of token t, normalised by the layer mean.
κ	Rescaling applied before the logarithm in the feature map; $\kappa = 100$.
$f_t,\ d_f$	Feature vector of token t and its dimension, $d_f = LH_q + 1$.
W_1, W_2, W_3	Weights of the detector MLP.
d_{mlp}	Hidden width of the detector MLP; $d_{\text{mlp}} = 128$ in our experiments.
$s_t,\ w_t$	Detector logit for token t and its importance $w_t = \sigma(s_t) \in (0, 1)$.
$\mathcal{F},\ y_t$	Far-back training region, and label $y_t = 1$ for tokens of the queried answer.
$\rho,\ \mathcal{L}$	Positive-class weight in the training loss, and the loss itself.

Bit allocation (§4.2).

$b_t,\ \bar{b}$	Bit-width assigned to token t, and the average-bit budget.
$\mathbf{b}$	The allocation $(b_1, \dots, b_S)$.
$\mathcal{B},\ \beta$	Tier set of realisable widths, $\mathcal{B} = \{1, 2, 8\}$, and one of its tiers.
$D(\mathbf{b})$	Functional distortion, the importance-weighted coder error.
λ	Budget multiplier; at the optimum every retained token has distortion λ.
Δ_t	Distortion removed per extra bit when token t is promoted one tier.
$Q_b,\ Q_b^{-1}$	Encoder and decoder of the b-bit coder; $\hat{x}_t = Q_{b_t}^{-1}(Q_{b_t}(x_t))$.

System cost (§4.3).

$\mathcal{C}(b)$	Bytes a b-bit code occupies for one token in one stream.
$P(\mathbf{b}),\ P_{16}$	Size of the compressed cache and of the fp16 cache, in bytes.
r	Realised compression ratio $r = P_{16}/P(\mathbf{b})$.
B_{link}	Host–device bandwidth.
n	Tokens generated in a turn.
T_{resume}	Transfer time to reload the cache on resume.
M_{HBM}	HBM the trajectory occupies while active.
R_{dec}	Bytes decode attention reads over a turn.
$m^{(\beta)}, \ell^{(\beta)}, o^{(\beta)}$	Running maximum, softmax normaliser and output of tier bucket β.

B EXPERIMENTAL SETTINGS

Model architectures and datasets. We use five open-weight decoders spanning an order of magnitude in scale and three attention designs: Llama-3.1-8B-Instruct (Llama Team, AI @ Meta, 2024) at its native 131,072-token context, Qwen2.5-1.5B-Instruct (Qwen Team, 2024) and Qwen3-4B-Instruct-2507 (Qwen Team, 2025) at 32k, Gemma-4-12B-it (Gemma Team, Google DeepMind, 2026), whose interleaved local/global layers carry a long-range cache only in the 8 global layers (of 48), and Qwen3.5-9B-Base (Qwen Team, 2026), whose linear-attention layers hold no KV cache so that only its 8 full-attention layers (of 32) can be compressed. All five use grouped-query attention, so the same per-head codec applies unchanged.

In §5.1 we evaluate multi-turn conversation ability on `SCBench` (Li et al., 2025b) in its *multi-turn* mode, on the full test split of all 12 tasks— `Retr.KV`, `Retr.Prefix-Suffix`, `VT` (string retrieval), `Code.RepoQA`, `En.QA`, `Zh.QA`, `En.MultiChoice` (semantic retrieval), `ICL.ManyShot`, `En.Sum`, `Math.Find` (global information), and the composites `Mix.Sum+NIAH` and `Mix.RepoQA+KV` (multi-tasking)—on Llama-3.1-8B-Instruct, Qwen3-4B-Instruct-2507, Gemma-4-12B-it and Qwen3.5-9B-Base, with up to five turns per shared context (Appendix D). In §5.2 we evaluate the single-turn `needle-in-a-haystack` test distributed with TurboQuant (Zandieh et al., 2025): a $\approx$2k-token filler haystack with spliced memos, in three escalating regimes—*single* (direct retrieval), *multi* (selection among distractors) and *hard* (2-hop chaining)—on all five models (Appendix E). In §5.3 we evaluate `RULER` (Hsieh et al., 2024) at a fixed 4,096-token context, using the pre-generated release of its 13 synthetic tasks with 100 examples per task, on the four larger models (Appendix F). Across all three benchmarks our setup emphasizes what a compressed cache can still *retrieve* under a fixed byte budget, the capability that separates the methods; the aggregation and summarization tasks are retained because they show where compression is nearly free.

Inference protocol. We evaluate the agentic scenario rather than a static reconstruction benchmark: forward the shared prefix to obtain the real cache, *compress* it once at the pause, rebuild a HuggingFace `DynamicCache`, then *resume*, generating every answer token against the compressed cache. Each turn's question and answer are appended as produced, so a later turn reads the compressed prefix together with the uncompressed dialogue before it. All methods see byte-identical prompts, decode greedily, and use each model's own chat template (raw text for the base model).

KV cache compression baselines. Our baselines are the strongest training-free members of the two families of §2. *Token dropping*: SnapKV (Li et al., 2024), PyramidKV (Cai et al., 2024), H2O (Zhang et al., 2023) and StreamingLLM (Xiao et al., 2024). *Quantization*: TurboQuant (Zandieh et al., 2025), denoted TQ Kx/Vy for x-bit keys and y-bit values. *Mixed-precision quantization*: QAQ (Dong et al., 2024), which assigns per-token bit-widths from an attention-aware score with separate key and value sensitivities. It varies precision across tokens and therefore the baseline that isolates our contribution from graded precision itself (§2). **FullKV**, the uncompressed fp16 cache, is the reference every method tries to preserve, and *Soft-bit KV (reload)* is the per-turn variant of §4.3, which re-allocates at every pause from that turn's own query (Appendix J), included to show how much of the remaining gap is due to allocating once rather than to the codec. Every comparison is made at a matched compression-ratio budget with an identical inference configuration, so a difference between rows reflects only how the bytes were spent.

We use released reference code throughout. The token-dropping baselines and FullKV run inside SCBench's own harness, released with MInference,[3] which implements the shared-context multi-turn KV lifecycle; SnapKV and StreamingLLM are native to it, and we port H2O's heavy-hitter cluster and PyramidKV's layer-wise budget from KVCache-Factory,[4] the codebase released with PyramidKV (Cai et al., 2024), so all four share one dense HuggingFace+FlashAttention-2 backend and differ *only* in which prompt-KV tokens survive prefill. For TurboQuant we use the released TurboQuant-v3 implementation[5] unmodified, at the configuration of Appendix G. QAQ runs from

[3] `https://github.com/microsoft/MInference`
[4] `https://github.com/Zefan-Cai/KVCache-Factory`
[5] `https://github.com/Kubenew/TurboQuant-v3`

the authors' released code,[6] on the same backend and at the same matched ratios as every other method.

Computational resources. All experiments run on NVIDIA H100 80GB GPUs, one model per GPU, so no inter-GPU communication enters the measurements. Host–device transfer is over PCIe Gen4 $\times$16, whose $\approx$25 GB/s is the B_{link} used in §4.3 and Table 5.

Metrics. We report three quantities. (i) **Compression ratio**: fp16 cache size divided by compressed size, computed from *measured* byte footprints rather than nominal bit-widths (Appendix G). (ii) **Task quality**, always in the benchmark's own metric—SCBench's per-task metric averaged over turns, HIT on the needle probe (the percentage of needle depths whose greedy continuation contains the gold string), and RULER's `string_match_all`. (iii) **System cost**: host-to-device bytes per resume, HBM residency and transfer latency (§5.4).

Matching the two families. The families are not commensurable on a single knob. Token dropping keeps a subset of tokens in fp16, so its ratio is set by a token budget and can be made arbitrarily aggressive; quantization keeps every token, so its ratio is set by bit-width and cannot reach the 16–32$\times$ regime a budget can. We therefore match on the one axis they share, the measured KV-memory compression ratio, and read comparisons only within a matched block. TurboQuant's random rotation is seed-sensitive, so its needle and RULER numbers average three quantizer seeds ($\{7, 42, 123\}$); on SCBench greedy multi-turn generation is seed-invariant on these tasks and we report a single seed (Appendix G).

Per-benchmark budgets. On SCBench every compressor receives the same $4{,}096$-token budget at prefill (SnapKV, PyramidKV and H2O: capacity $4{,}096$ with a 32-token observation window; StreamingLLM: 128 sink + $3{,}968$ recent), with Llama at `max_seq_length` $= 131{,}072$ and the other models at $32{,}768$, up to 5 turns per example. On the needle probe the three matched ratios correspond to 512, 256 and 183 kept tokens for the dropping methods and to 4-, 2- and 1-bit codes for TurboQuant. On RULER all compressed rows sit at $11.8\times$: a 347-token budget ($4096/11.8$), 1-bit codes, and an average-bit budget calibrated per model for SOFT-BIT KV.

[6] `https://github.com/ClubieDong/KVCacheQuantization`

C RETAINED FULL-CACHE ACCURACY

The abstract compares SOFT-BIT KV with the baselines across three benchmarks whose native metrics differ: SCBench's per-task scores, needle HIT, and RULER's `string_match_all`. To put them on one scale we report *retained full-cache accuracy*. For a benchmark evaluated on model set $\mathcal{M}$, a method's retained accuracy is $\sum_{m\in\mathcal{M}} \text{score}_m \big/ \sum_{m\in\mathcal{M}} \text{score}_m^{\text{FullKV}}$, the ratio of its mean score to the full cache's mean score over the same models. Per-model scores are the Avg. columns of Tables 2 and 4 and, for the needle probe, the mean HIT over the three regimes of Table 3; FullKV scores 100 on every needle cell, so retained accuracy there equals HIT. The *best token-dropping method* is the single dropping method with the highest retained accuracy on a benchmark, which is SnapKV on all three.

Table 7 gives the three comparisons quoted in the abstract, and Table 6 breaks each down by model. The per-model view shows what the averages hide. On SCBench, SOFT-BIT KV exceeds the full cache on Gemma-4-12B, the one model on which uniform quantization is itself near-lossless. On the needle probe, TurboQuant retrieves nothing on Qwen2.5-1.5B or Qwen3-4B while holding every regime on Gemma-4-12B. On RULER, SOFT-BIT KV leads on every model.

Table 6: **Retained full-cache accuracy (%) per model**: each method's score divided by FullKV's on the same model. **Avg.** is the ratio of mean scores over the models in that panel, the quantity reported in Table 7. RULER lists both SOFT-BIT KV selectors; the abstract quotes the trained-MLP selector. Needle scores are mean HIT over the three regimes, with TurboQuant at 11.8×. Bold marks the best compressed method per column.

(a) SCBench multi-turn at 5.2×, four models

Method	Llama-3.1 8B	Qwen3 4B	Gemma-4 12B	Qwen3.5 9B	Avg.
SnapKV	59.7	75.8	76.9	86.0	72.1
PyramidKV	61.8	73.1	72.6	83.8	70.9
H2O	48.9	75.8	76.9	86.0	68.3
StreamingLLM	43.6	60.6	61.9	68.5	56.2
TurboQuant	81.3	65.3	99.3	98.7	85.1
SOFT-BIT KV (ours)	**84.8**	**93.6**	**105.2**	**99.1**	**94.2**

(b) Needle-in-a-haystack at ≈11×, five models

Method	Llama-3.1 8B	Qwen2.5 1.5B	Gemma-4 12B	Qwen3 4B	Qwen3.5 9B	Avg.
SnapKV	66.7	33.3	**100.0**	66.7	39.0	61.1
PyramidKV	66.7	50.0	66.7	11.0	50.0	48.9
H2O	33.3	0.0	**100.0**	66.7	39.0	47.8
StreamingLLM	16.7	16.7	33.3	16.7	16.7	20.0
TurboQuant	66.7	0.0	**100.0**	0.0	22.3	37.8
SOFT-BIT KV (ours)	**92.7**	**100.0**	**100.0**	**100.0**	**100.0**	**98.5**

(c) RULER at 11.8×, four models

Method	Llama-3.1 8B	Gemma-4 12B	Qwen3 4B	Qwen3.5 9B	Avg.
SnapKV	71.6	47.8	49.4	56.0	56.4
PyramidKV	69.4	32.5	26.0	39.8	42.2
H2O	28.8	47.8	49.4	56.0	45.3
StreamingLLM	23.7	21.7	24.0	21.2	22.7
TurboQuant	17.3	51.4	3.2	22.7	23.1
SOFT-BIT KV (RH)	67.7	39.7	22.4	38.4	42.2
SOFT-BIT KV (ours)	**73.0**	**56.2**	**68.0**	**63.0**	**65.2**

Table 7: **Retained full-cache accuracy (%)**, the comparisons quoted in the abstract. Each entry is a method's mean score over the listed models divided by FullKV's mean score over the same models (Appendix C). On the needle probe, token dropping and SOFT-BIT KV run at $11.2\times$ and TurboQuant at $11.8\times$, its nearest realisable ratio. *SnapKV is the best token-dropping method on all three benchmarks. Per-model values are in Table 6.

Benchmark	Ratio	#Models	SOFT-BIT KV	TurboQuant	SnapKV*
SCBench multi-turn	$5.2\times$	4	**94**	85	72
Needle	$\approx 11\times$	5	**99**	38	61
RULER	$11.8\times$	4	**65**	23	56

D SCBENCH DETAILS

Benchmark and protocol. SCBench (Li et al., 2025b) evaluates KV-cache methods in the *shared-context* regime: a single long context (a document, codebase, or fact dictionary) is prefilled *once*, and the model then answers a sequence of queries against it, with each turn's question and answer appended to the cache. The benchmark defines two shared-context modes. In *multi-turn* mode (which we use) the turns form one conversation—the previous turn's answer is prepended to the next question, so the cache carries a growing dialogue—mirroring long-context chat and multi-step tool use. In *multi-request* mode the same cached context serves independent requests from different sessions. Both modes share the property that defines the paused agentic trajectory: any compression is committed *before* the future queries are known, and a token evicted at prefill is unrecoverable for every later turn. We run the full test split of all 12 tasks (Table 8) in multi-turn mode, up to 5 turns per example, greedy decoding, with each model's chat template. All compressors operate at prefill with an identical 4,096-token budget (SnapKV/H2O: capacity 4,096, 32-token observation window; StreamingLLM: 128 sink + 3,968 recent tokens) on the same dense flash-attention backend, so they differ only in *which* tokens they keep. Reported numbers are accuracy averaged over turns.

Table 8: **The 12 SCBench tasks**, organized by SCBench's four capability categories (Li et al., 2025b). n = test examples; turns = queries per shared context (tasks marked † natively run up to 8 turns, truncated to 5 in our runs). The last column marks whether a task *discriminates multi-turn conversation ability*: **Yes** if its later turns query information that could not be located when the cache was compressed at prefill (each turn needs a *new* fact), so a prefill-time compressor must have kept it blind—these are the tasks on which multi-turn KV reuse is actually stressed. **No** marks global-aggregation tasks whose answer is supported by many context subsets (any 4k keep-set suffices, so multi-turn compression is not stressed); **Partial** marks paragraph-level QA that is only weakly localized; and **Mixed** marks the summary+needle composite, one of whose two interleaved sub-metrics (the needle) is new-fact-per-turn and the other (summary) is not.

Task (harness id)	Category	n	Turns	Metric	Multi-turn conv.?
Retr.KV (`scbench_kv`)	String Retrieval	100	5	substring match	**Yes**
Retr.PS (`scbench_prefix_suffix`)	String Retrieval	100	5	substring match	**Yes**
VT / Retr.MultiHop (`scbench_vt`)	String Retrieval	90	5	string match (all)	**Yes**
Code.RepoQA (`scbench_repoqa`)	Semantic Retrieval	88	5	code-sim. pass@1	**Yes**
En.QA (`scbench_qa_eng`)	Semantic Retrieval	69	5	F1	Partial
Zh.QA (`scbench_qa_chn`)†	Semantic Retrieval	35	8→5	F1 (char)	Partial
En.MultiChoice (`scbench_choice_eng`)	Semantic Retrieval	58	4	accuracy	Partial
ICL.ManyShot (`scbench_many_shot`)	Global Information	54	5	accuracy	No
En.Sum (`scbench_summary`)	Global Information	70	5	ROUGE-L-Sum	No
Math.Find (`scbench_mf`)	Global Information	100	6→5	accuracy	No
Mix.Sum+NIAH (`scbench_summary_with_needles`)†	Multi-tasking	70	8→5	ROUGE / passkey acc.	Mixed
Mix.RepoQA+KV (`scbench_repoqa_and_kv`)†	Multi-tasking	88	8→5	pass@1 / substring	**Yes**

What each category tests. We describe each of the 12 tasks in turn, with a concrete data example (drawn from the actual test set); the four category labels follow SCBench (Li et al., 2025b).

Retr.KV (String Retrieval). The context is a large JSON dictionary of UUID key–value pairs, and each turn names a *different* key whose exact 36-character value must be reproduced. *Example:* given the entry `"3f2a8c..."`: `"cb59052b-9128-4979-9c0e-e1de4adcf73b"`

among hundreds of others, the turn "the value of key `3f2a8c...` is" targets the string `cb59052b-9128-4979-9c0e-e1de4adcf73b`. The queried key changes every turn and occupies a tiny fraction of the context, so no prefill-time importance estimate can know which tokens matter.

Retr.PS (String Retrieval). The context is a list of random strings, and each turn gives a prefix and a suffix and asks for the unique string matching both—so the needle cannot be found by any single contiguous match. *Example:* prefix `_Tmw` and suffix `+KJB` target the unique string `_TmwCWPliaoDImvXiOvic+KJB`.

VT / Retr.MultiHop (String Retrieval). The context interleaves chains of variable assignments (`VAR X1 = 12345`, `VAR X2 = X1`, ...), and each turn names a value and asks for *all* variables that resolve to it, forcing the model to follow multi-step assignment chains. It therefore also tests whether the *relational* structure survives compression, not just isolated facts. *Example:* the queried value resolves to the variable set `[CTLFS, VVUTI, QBORE, IHQFY, JWJQC]`.

Code.RepoQA (Semantic Retrieval). The context is a concatenated code repository, and each turn gives a natural-language function description and asks for the verbatim function body, scored by code similarity. *Example:* the description of "a predicate deciding whether a line's closing parenthesis may be omitted" targets the exact source of `def _can_omit_closing_paren(line, *, last, line_length):`. A new function is queried each turn, so like the string-retrieval tasks it needs a fact not identifiable at prefill.

En.QA (Semantic Retrieval). Open-ended questions over a long English book, scored by token-level F1. *Example:* a question about a character's name targets the short span `Peyton`. Answers are paragraph-level and partially redundant with the book's global topic, placing this task between the retrieval and global groups in compression sensitivity.

Zh.QA (Semantic Retrieval). The Chinese-document counterpart of En.QA: each turn poses a Chinese-language question over a long Chinese document and is scored by character-level F1. *Example:* the answer is a short span such as a queried person or place name. Its compression sensitivity mirrors En.QA.

En.MultiChoice (Semantic Retrieval). Four-way multiple-choice questions over a long book, answered with a single letter. *Example:* a question about the entity "Snowfield" targets option `C`. Because the correct option is often inferable from the book's gist, the task is only weakly localized.

ICL.ManyShot (Global Information). The context is hundreds of in-context labeled classification examples, and each turn classifies a new input into one label. *Example:* a new input targets the label `D`. The label mapping is spread across many demonstrations, so any 4k-token keep-set preserves it and the task tolerates aggressive eviction.

En.Sum (Global Information). Each turn requests an (updated) summary of the document, scored by ROUGE-L. *Example:* a target summary begins "The key contribution of the paper ... is that it ...". The relevant signal is distributed across the whole context, so eviction costs little.

Math.Find (Global Information). Each turn asks for an aggregate statistic (e.g., the largest number) of a long numeric array. *Example:* the maximum over one array is `891`. Many different subsets of the array support the answer, so this task does not discriminate compression methods.

Mix.Sum+NIAH (Multi-tasking). Interleaves summarization turns with passkey-retrieval turns over the *same* cache; the reported number averages the two sub-metrics (ROUGE-L and passkey accuracy). *Example:* a summary turn ("summarize the document so far") alternates with a needle turn ("the passkey is" $\rightarrow$ a planted digit string). Its needle half needs a new fact per turn and stresses multi-turn reuse; its summary half does not—hence the **Mixed** marking.

Mix.RepoQA+KV (Multi-tasking). Interleaves RepoQA turns with Retr.KV turns over one cache, averaging the pass@1 and substring-match sub-metrics. *Example:* a turn retrieving a described function body alternates with a turn retrieving a UUID value by key. Both halves are new-fact-per-turn, so a single compressed cache must simultaneously serve two exact retrieval query distributions—the hardest case for prefill-time compression.

Which tasks certify multi-turn conversation ability. All 12 tasks are evaluated in multi-turn mode, but only a subset actually *stresses* it: the ones whose *later* turns query facts that were not identifiable when the cache was compressed—the three String Retrieval tasks, RepoQA, and Mix.RepoQA+KV, with Mix.Sum+NIAH stressing it on its needle half (the rows marked **Yes/Mixed** in Table 8). On these, Table 2 shows the characteristic multi-turn signature: with the full cache, Llama-3.1-8B *improves* over turns on Retr.KV (58→78→85→85→86% across turns 1–5, as the dialogue locks in the answer format), whereas every query-agnostic compressed cache is already dead at turn 1 ($\leq$2%) and never recovers—the needles were evicted at prefill. The ordering among baselines is the ordering of query awareness: SnapKV (Li et al., 2024), which selects by the attention of the final prompt window (the first question), beats the query-agnostic heavy-hitter rule of H2O (Zhang et al., 2023) and the purely positional sink+recency rule of StreamingLLM (Xiao et al., 2024), but all three collapse on exact retrieval because none can condition on queries that have not yet arrived. By contrast, the Global Information tasks are nearly compression-insensitive—SnapKV even exceeds FullKV on Llama many-shot—so multi-turn claims should rest on the retrieval and composite columns, not on benchmark-wide averages alone. This is precisely the regime our pause-time setting targets: at a pause, the *next* query is often known, which is the information all prefill-time methods above are structurally denied.

E NEEDLE-IN-A-HAYSTACK DETAILS

Origin of the probe. The needle probe of §5.2 builds on the generation test distributed with the open-source PyTorch reference implementation of TurboQuant (Zandieh et al., 2025).[7] That test hides a single fact (`The secret project code name is AURORA-7749.`) inside a repeated neutral filler paragraph, at a configurable relative depth of a $\approx 2{,}048$-token document, and asks whether the fact is reproduced verbatim in a greedy continuation. We retain its construction—the filler-repetition haystack, the memo delimiters, the depth parameterisation and the exact-substring criterion—so that our *single* regime is directly comparable to it, and add two further regimes, *multi* and *hard*, which vary the structure of the retrieval while holding haystack, length, decoding and backend fixed.

Haystack and prompt construction. The haystack is one topically neutral filler paragraph—"`The quarterly review covered budget allocations, departmental spending and projected revenue. Infrastructure upgrades for the western offices were noted and coordinated with facilities.`"—repeated $\lfloor 2048/|\text{filler}| \rfloor$ times, where $|\text{filler}|$ is its length under the evaluated model's tokenizer. One or more *memos*, each delimited by `--- Memo ---` and `--- End ---`, are spliced between repetitions at prescribed relative depths, and the question is appended after the haystack. Because the filler shares no content word with any question, a token's relevance is determined by the memo structure alone rather than by topical similarity to the query. Realised prompt lengths are 2,011–2,123 tokens across the five models and three regimes; the spread is tokenizer- and regime-dependent.

Scoring. Each regime is evaluated at a set of needle depths, and a cell reports HIT: the percentage of depths whose 24-token greedy continuation contains the gold string as a case-insensitive substring—the criterion RULER also uses for string matching. The *single* and *hard* regimes are evaluated at depths $\{0.3, 0.5, 0.7\}$, so their per-cell HIT takes values in $\{0, 33, 67, 100\}$; the *multi* regime is evaluated at $\{0.3, 0.5\}$ (see the note on depth collision below), so its per-cell HIT takes values in $\{0, 50, 100\}$. All methods run through one HuggingFace+FlashAttention-2 capture-decode harness on byte-identical prompts with greedy decoding.

Table 9: **The three needle regimes.** Each plants memos into the same filler haystack and asks one question. "Isolates" names the retrieval capability the regime is designed to vary: the regimes differ only in how many memos compete with the queried one, and in whether the memo carrying the gold string shares any surface form with the question.

Regime	Memos planted	Question	Gold	Isolates
single	1	"What is code phrase ALPHA?"	`4471`	direct retrieval
multi	4, at spread depths	"What is code phrase DELTA?"	`7314`	selection among distractors
hard (2-hop)	2, chained	"What is the VAULT access key?"	`X7K9Q-42M`	chaining to a non-salient span

single (direct retrieval). A single memo is spliced at the target depth, `--- Memo --- Code phrase ALPHA is 4471. --- End ---`, and the question reads "`What is code phrase ALPHA? Answer with the number only.`" with gold `4471`. The queried identifier appears verbatim in both the memo and the question, so the answer-bearing span is maximally salient to any query-conditioned score computed at prefill. This is the regime inherited from the reference implementation, and it serves as the easy control.

multi (selection among distractors). Four memos of identical shape— `ALPHA→4471`, `BRAVO→8853`, `CHARLIE→2096`, `DELTA→7314`—are placed at depths spread across the haystack, and the question asks for `DELTA` alone. Because the distractor memos are structurally indistinguishable from the queried one, a criterion that merely recognises "memo-shaped" text retains all four and spends its budget on three irrelevant spans. The regime therefore separates *detecting* a needle from *selecting* the right one. It is also the regime in which a purely positional keep-set can score without any query awareness, since the memos nearest the end of the haystack fall inside a recency window.

[7] `https://github.com/tonbistudio/turboquant-pytorch`

hard (2-hop chaining). Two memos form a chain. `Memo A: Project ZETA launch code is X7K9Q-42M` is placed at depth p, and `Memo B: The VAULT access key equals Project ZETA's launch code` at depth $\min(0.92, p + 0.3)$. The question is "`What is the VAULT access key? Answer with the exact code only.`" with gold `X7K9Q-42M`. The design point is a deliberate dissociation between salience and evidence: Memo A carries the gold string but shares no content word with the question, while Memo B shares the question's words but carries no answer. Answering therefore requires resolving Q→B→A, and the span that must survive compression is precisely the one that a question-conditioned importance score has no lexical reason to rank highly. This is the property the regime exists to test, and it is what distinguishes it from RULER's variable-tracking task (Appendix F), whose intermediate hops remain lexically tied to the query.

Depth collision in the *multi* regime. The four memo depths are computed as $\mathrm{clip}(p - 0.25 + 0.2(j{+}1),\ 0.03,\ 0.97)$ for $j = 0, \ldots, 3$. At $p = 0.7$ the last two depths both saturate the upper clip at 0.97 and map to the same insertion index; the generator resolves the tie in favour of the earlier memo, so the `DELTA` memo—the one the question asks for—is not planted at that depth, and the item is unanswerable for every method. We therefore evaluate *multi* at $\{0.3, 0.5\}$ only, and report its HIT over those two depths; the *single* and *hard* regimes are unaffected, since their depths cannot collide. We verified by direct inspection of the generated prompts that the gold string is present at every evaluated depth of all three regimes.

Operating points. The two compression families are steered by different knobs, so each configuration is reported at the KV-memory ratio it actually realises rather than at a nominal target (Table 10). Token-dropping budgets are kept-token counts against the $\approx 2{,}048$-token prompt; TurboQuant is set by integer bit-width, which leaves it no freedom to land on an arbitrary target; soft-bit KV's average-bit budget is calibrated per model to meet the matched ratio. The three sections of Table 3 correspond to the three columns below.

Table 10: **Realised compression ratios** for the needle probe, measured on the evaluated prompts; ranges span the five models. Token dropping and soft-bit KV can be tuned onto a target ratio. TurboQuant cannot: retaining every token makes bit-width its only knob, and 1 bit is a hard floor near $12\times$.

Family	**Knob**	$\approx 4\times$	$\approx 8\times$	$\approx 11\times$
Token dropping	kept tokens	512 (4.03–4.06×)	256 (8.04–8.13×)	183 (11.20–11.37×)
TurboQuant	bit-width	K4/V4 (3.71–3.77×)	K2/V2 (6.84–7.03×)	K1/V1 (11.80–12.38×)
Soft-bit KV	avg. bits	3.9 (3.98–4.04×)	1.9 (7.90–8.15×)	0.8 (10.85–11.32×)

F RULER DETAILS

Benchmark and protocol. RULER (Hsieh et al., 2024) generates synthetic long-context tasks at a controlled sequence length. Because every example in a task is produced to the same target length, context length is held *constant* and only the compressor varies—the property that makes RULER the most controlled of our three evaluations. We use the pre-generated `simonjegou/ruler` release at a 4,096-token context, which ships 500 examples for each of 13 tasks, and evaluate the first 100 per task (1,300 examples per model per method). Each example supplies a `context`, a `question`, an `answer_prefix` that pins the output format, and a list of one or more gold strings. We assemble the prompt as `context + question`, apply the model's chat template (raw text for base models), then append `answer_prefix`, and decode greedily for the task's own `max_new_tokens`. Scoring is RULER's `string_match_all`: the fraction of an example's gold strings that appear case-insensitively in the continuation, so multi-answer tasks receive partial credit rather than all-or-nothing. Every compressed row sits at a single matched $11.8\times$—token-droppers receive a 347-token budget ($4096/11.8$), TurboQuant runs at 1 bit, and soft-bit KV's average-bit budget is calibrated per model to the same measured ratio, with either the retrieval-head (RH) or the trained-MLP keep-set.

Single-turn by construction. Each RULER example poses exactly one query against one context, issued after the cache is built. RULER therefore does not exercise the multi-turn constraint that SCBench is built around (Appendix D): there is no later query whose target was unknown at compression time. It complements SCBench precisely by removing that variable—together with context length, which SCBench leaves free—so that a difference between methods can be attributed to the keep-set or the code alone.

Table 11: **The 13 RULER tasks** at 4k context, grouped by the category headings used in Table 4. "Target" is what must be reproduced; "Distractors" names what competes with it inside the haystack; "Refs" is the number of gold strings per example, i.e. the denominator of `string_match_all`; `max_new` is RULER's per-task generation budget. The NIAH family varies two axes independently—the needle (a 7-digit number versus a 36-character UUID) and the haystack (repeated noise, essay prose, or nothing but competing needles).

Task (harness id)	Category	Target	Haystack	Distractors	Refs	max_new
S-1 (niah_single_1)	Single NIAH	7-digit number	repeated noise sentences	none	1	128
S-2 (niah_single_2)	Single NIAH	7-digit number	essay prose	prose only	1	128
S-3 (niah_single_3)	Single NIAH	UUID	essay prose	prose only	1	128
MK-1 (niah_multikey_1)	Multi-key NIAH	7-digit number	essay prose	other needles	1	128
MK-2 (niah_multikey_2)	Multi-key NIAH	7-digit number	competing needles only	other needles	1	128
MK-3 (niah_multikey_3)	Multi-key NIAH	UUID, keyed by UUID	competing needles only	other needles	1	128
MQ (niah_multiquery)	Multi-query NIAH	4 numbers, 4 keys	essay prose	prose only	4	128
MV (niah_multivalue)	Multi-value NIAH	4 numbers, 1 key	essay prose	prose only	4	128
VT (vt)	Multi-hop	5 variable names	assignment chains + noise	other chains	5	30
CWE (cwe)	Aggregation	10 frequent words	numbered word list	rare words	10	120
FWE (fwe)	Aggregation	3 frequent words	coded word stream	rare coded words	3	50
QA-1 (qa_1)	QA	short answer span	SQuAD documents	other documents	1	32
QA-2 (qa_2)	QA	short answer span	HotpotQA documents	other documents	1	32

Single NIAH (S-1, S-2, S-3). A single "magic" value is hidden in a haystack and queried by a nonsense two-word key; the three variants escalate along two independent axes, the haystack and the needle. *S-1* uses the repeated sentence "`The grass is green. The sky is blue. The sun is yellow. ...`" as filler—*Example:* the query "`What is the special magic number for wandering-age mentioned in the provided text?`" targets `8090293`. *S-2* keeps the numeric needle but replaces the filler with real essay prose, which supplies lexically rich distractor content. *S-3* keeps the essay haystack and replaces the needle with a 36-character UUID—*Example:* the key `amused-quart` targets `1ff49b78-8946-4e85-b59c-de66bacfb3d0`. The UUID variant is the structurally harder one: the answer spans many tokens, none of which is individually distinctive, so it is preserved only if *every* one of its tokens survives compression, and any partial retention scores zero under exact string match.

Multi-key NIAH (MK-1, MK-2, MK-3). The haystack is populated with other key–value needles, so the queried key must be selected among structurally identical competitors. *MK-1* embeds distractor needles in essay prose. *MK-2* removes the prose entirely, so the context consists of nothing but competing needles—*Example:* lines such as "`One of the special magic numbers for omniscient-performance is: 9796605.`" repeat with different keys, and the queried key `innocent-clause` targets `5774152`. *MK-3* makes both key and value UUIDs, so the query itself is a 36-character string—*Example:* "`What is the special magic uuid for 455ac762-7428-4656-b3ee-4d3b5a104129?`" targets `1ee34dc4-3b04-4a8b-805b-fdc94e7ed827`. MK-3 compounds both difficulties: a multi-token needle that must match exactly, sitting among hundreds of tokens with near-identical surface form.

Multi-query and multi-value NIAH (MQ, MV). Both broaden retrieval to four gold strings and are scored by partial credit. *MQ* asks for the values of four different keys in one question—*Example:* "`used-commotion, squealing-dibble, annoying-decimal, and depressed-tweet`" targets four 7-digit numbers. *MV* asks instead for all four values associated with a *single* key. Because the four targets are independent, losing one costs only 25 points, so these tasks measure graceful degradation rather than the all-or-nothing behaviour of the single-answer variants.

Multi-hop (VT). Chains of variable assignments (`VAR QPE = 64886`, then further variables assigned to earlier ones) are interleaved with noise filler, and the query names a value and asks for *all* variables that resolve to it. *Example:* the value `15311` targets the five names `[FITJT, VGCAO, ZJQUQ, TYFAD, DROFS]`. The task is multi-hop in the sense that the chain must be followed, but each assignment statement is short, syntactically distinctive, and lexically tied to the queried value—so, unlike the 2-hop regime of Appendix E, every hop remains surface-relevant to the question.

Aggregation (CWE, FWE). Both ask for the most frequent items in a long list rather than for a localised needle, so their evidence is distributed over the entire context. *CWE* gives a numbered list of words and asks for the top 10—*Example:* the gold set is `[appliance, meter, forest, ferret, kilt, behalf, fire, activity, arthur, authenticity]`. *FWE* gives a stream of nonsense coded words interleaved with dots and asks for the top 3—*Example:* `[fkmgoo, quqtyf, rsrvqx]`. These are the two tasks whose answers no small keep-set can support in principle, because the answer is a count over the whole list rather than a span within it; they therefore probe coverage rather than selection.

QA (QA-1, QA-2). Real-document question answering embedded in a haystack of distractor documents. *QA-1* draws from SQuAD—*Example:* "`In what country is Normandy located?`" targets `France`. *QA-2* draws from HotpotQA and requires combining two documents—*Example:* "`Were Scott Derrickson and Ed Wood of the same nationality?`" targets `yes`. Both instruct the model to emit the answer alone. Unlike the synthetic tasks, their answers are natural-language spans whose surface form the model may paraphrase, so absolute scores are lower for every method including the uncompressed baseline, and the headroom within which methods can differ is correspondingly narrow.

G TurboQuant Baseline: Configuration, Ratio Accounting, and a Non-Monotonicity

Configuration. We use the released MSE-only TurboQuant coder (Zandieh et al., 2025): each key/value vector is normalised to the unit sphere, rotated by a fixed random rotation, and quantised per coordinate with a Lloyd–Max codebook, storing packed indices plus an fp16 norm. We sweep symmetric bit-widths Kx/V$y \in$ {4/4, 2/2, 1/1} and keep a 32-token fp16 residual window. At a pause we quantise the prefix cache once, then decode against the quantised cache, so every generated token is produced under compression.

Cross-table consistency. All three tables (SCBench, needle, RULER) use the identical released `TurboQuantV3` coder with the same configuration (K/V 2-bit, 32-token fp16 residual window, no protected layers); we verified that the two call sites in our harness—an in-place chunked pass for the long SCBench caches and the library's `compress_kv` elsewhere—produce *bitwise-identical* quantised tensors. The coder's random rotation is seed-dependent, so we average over three quantiser seeds ($\{7, 42, 123\}$) on needle and RULER; on SCBench we verified that greedy multi-turn generation is seed-invariant to numerical precision (different rotations flip no argmax on these tasks), so the single-seed and three-seed numbers coincide and we report a single seed. This makes the TurboQuant rows directly comparable across tables.

Why the advantage is model-dependent, not an implementation artifact. The same coder is near-lossless on Llama-3.1-8B (RULER 92.0 vs. FullKV 95.4) yet destructive on Qwen2.5-1.5B (RULER 23.9 vs. 85.0). We confirmed this is a capacity effect, not unfair compression: on *both* models the 2-bit coder produces the same per-vector reconstruction error ($\|\Delta\|/\|x\| = 0.34$ for keys and values) at the same measured ratio, using the same code path. The amplifier is architectural—Qwen2.5-1.5B has only 2 key/value heads, each shared across $g = 6$ query heads under grouped-query attention, versus 8 heads at $g = 4$ on Llama—so identical KV noise propagates to six times as many queries, and a smaller model has less redundancy to absorb it. The MSE-only coder minimises reconstruction error but is inner-product biased, and attention depends on $q \cdot k$ rather than on reconstruction; a high-capacity model tolerates this bias where a small one does not.

Ratio accounting. We report *measured* ratios, not the nominal $16/b$. For head dimension $d = 128$ the compressed footprint is $16b$ bytes of indices plus 2 bytes of fp16 norm per vector against $2d = 256$ bytes uncompressed, so the true ratio is $256/(16b + 2)$: $3.9\times$ at 4 bits, $7.5\times$ at 2 bits and $14.2\times$ at 1 bit, further reduced by the residual window to the $3.7/6.8/11.8\times$ of Table 3. **This is the ceiling that matters for our comparison: because all tokens are retained, no bit-width can reach the $16\times$–$32\times$ regime that token dropping occupies, and 1 bit is the floor.** Reporting $16/b$ instead would overstate the quantizer by $\sim 20\%$ and understate the gap.

Seed sensitivity. The random rotation is seeded, and outcomes move with the seed. All TurboQuant needle numbers are therefore averaged over three seeds $\{7, 42, 123\}$; single-seed runs of this coder are not reliable at the granularity of a 3-position HIT.

An unexplained non-monotonicity on the 1.5B model. On Qwen2.5-1.5B, K4/V4 collapses to 0 while the *more* aggressive K2/V2 retrieves at $89/56/56$ (Table 3); the 8B model shows no such inversion. We report what we established and what we did not.

It is not noise. The cell aggregates 3 needle positions $\times$ 3 seeds; K4/V4 misses $9/9$ and K2/V2 hits $8/9$, a separation far outside binomial fluctuation.

It is not a reconstruction failure. On real Qwen KV the relative reconstruction error is properly ordered: 0.008 at 8 bits, 0.097 at 4, 0.341 at 2, 0.600 at 1. Whatever is happening, the 4-bit code reconstructs *better* and still generates worse.

It lives entirely on the key path. Quantising keys alone reproduces the inversion exactly ($0/3$ at 4 bits, $3/3$ at 2 bits on the single-needle probe), whereas quantising values alone is harmless at both widths ($3/3$, $3/3$). Any explanation must therefore be about how quantised keys enter the attention scores, not about value payload error.

A candidate mechanism, measured but insufficient. Attention consumes inner products $q \cdot k$, and an MSE-optimal quantizer is not inner-product unbiased. We confirm the bias exists and grows as the code coarsens: the mean score shift $\mathbb{E}[q \cdot \hat{k} - q \cdot k]$ measured on real keys is 0.08, 1.21 and 4.07 at 4, 2 and 1 bits. This is exactly the term TurboQuant's two-stage variant cancels with a 1-bit QJL correction—which, however, corrects scores *at attention time* and so cannot be applied as a drop-in cache transform; we did not implement it. But the bias is *monotone* in bit-width, so on its own it does not explain an inversion in which 4 bits is worse than 2. We therefore do not claim a validated mechanism: we record the inversion as an open anomaly of this coder configuration on this model, and we use the K2/V2 operating point when quoting Qwen numbers.

Why we report it anyway. The episode is itself evidence for the paper's premise. Every scalar that a conventional quantizer optimises—per-vector reconstruction error—moved in the right direction while end-task behaviour moved in the wrong one. That dissociation is the reason §4 optimises a *functional* (output-side) distortion instead.

H WHY A LEARNED HEAD COMBINATION

§4.1 claims that averaging the question-window attention over heads is a weak importance score and that the heads which matter are not the heads that attend most. This appendix reports the comparison behind both claims.

Setup. We probe two models: Qwen2.5-1.5B-Instruct (28 layers, 12 query heads, 2 KV heads) and Gemma-4-12B-it, whose 8 global full-attention layers are the only ones holding long-range KV, so the detector reads those 8×16 heads. Each probe plants needles in a 2k-token haystack and asks one question, in three regimes—*single* (direct retrieval), *multi* (selection among distractors) and 2-*hop* (chaining)—crossed with two haystacks, repetitive *filler* and *real text*. We report HIT over 4–8 seeds $\times$ 4 needle depths as the KV budget is swept.

Selectors compared. All three run in one harness with identical storage; only the ranking that decides which tokens are kept at full precision differs. **Average** ranks tokens by the mean of $a_t^{(l,h)}$ over *all* heads. **Top-heads** ranks by the mean over a calibrated committee of retrieval heads (the best of the top $1/3/5\%$ fractions), and is training-free. **Trained MLP** is the detector of Eq. (3).

Metric. Table 12 reports, for each setting, the *largest* compression ratio at which a selector is still retrieval-lossless, meaning its HIT matches the uncompressed model's own HIT on that task and stays matched at every larger budget. We deliberately use full-KV rather than a gold-only oracle as the bar: keeping only the gold tokens can de-noise and exceed full-KV, which would mark a lossless selector as a failure. "fails" means the selector never reaches that bar anywhere in the swept range.

Table 12: **Largest retrieval-lossless compression ratio** for the three scores, higher is better. Ceiling HIT is the uncompressed model's own HIT on that task; where it is below 1.00 the task is model-limited and the selectors tie. Oracle $\times$ is the ratio a gold-only keep-set reaches, an upper bound no realisable selector can pass.

Model	Task	Ceiling / oracle	Average	Top-heads	Trained MLP
filler haystack					
Qwen2.5-1.5B	single	1.00 / 29×	5.2×	21.1×	**28.9×**
Qwen2.5-1.5B	multi	0.50 / 30×	5.2×	13.9×	**28.1×**
Qwen2.5-1.5B	2-hop	1.00 / 26×	fails	5.2×	**23.3×**
Gemma-4-12B	single	1.00 / 27×	7.2×	17.0×	**25.9×**
Gemma-4-12B	multi	0.75 / 27×	18.1×	25.6×	25.6×
Gemma-4-12B	2-hop	1.00 / 24×	5.1×	12.5×	**22.7×**
real-text haystack					
Qwen2.5-1.5B	single	0.97 / 39×	fails	23.7×	**31.8×**
Qwen2.5-1.5B	multi	0.47 / 40×	fails	10.2×	**31.3×**
Qwen2.5-1.5B	2-hop	1.00 / 35×	fails	fails	**13.0×**
Gemma-4-12B	single	1.00 / 32×	11.3×	28.2×	**30.7×**
Gemma-4-12B	multi	0.75 / 33×	24.7×	31.2×	31.0×
Gemma-4-12B	2-hop	1.00 / 30×	fails	8.7×	**14.8×**

Averaging over heads is the weakest score. It fails outright in five of the twelve settings and never sustains more than 24.7×, and the failures are concentrated where retrieval is hard: every 2-hop setting and every real-text Qwen setting. Restricting the average to a calibrated committee of retrieval heads (top-heads) recovers much of the gap, which is itself evidence that the signal lives in a few heads rather than in the pooled mean. The trained detector is best in ten of the twelve settings and ties on the two multi-needle rows where the uncompressed model is itself at 0.47–0.75 HIT, so no selector can separate. On real-text 2-hop it is the only score that retrieves at all on Qwen.

The heads that matter are not the heads that attend most. Scoring each input feature of the trained detector by its first-layer weight norm, the top-weighted heads overlap the raw attention-to-gold ranking by only 6% on Qwen and 17% on Gemma. The reliance is also structured differently per model: it spreads across depth on Qwen and concentrates in Gemma's mid-to-late global layers.

A committee chosen by "which heads attend to the answer" therefore recovers a different, and weaker, set than the discriminative combination the detector learns, which is why top-heads sits between the average and the trained score throughout Table 12.

Table 13: **Detector configuration for every evaluated model.** Each model has its own detector, trained once offline with the base model frozen; none is transferred between models. The feature dimension follows $d_f = LH_q + 1$ over the layers that hold a long-range KV cache: all layers for the dense models, and only the 8 global (Gemma-4) or full-attention (Qwen3.5) layers for the hybrid ones. All runs use AdamW at learning rate 10^{-3} with question window $W = 64$ and $d_{\mathrm{mlp}} = 128$, on the synthetic retrieval data of §4.1. Parameter counts and steps are read from the released checkpoints.

Model	Scored layers × heads	d_f	Params	Steps	Used in
Llama-3.1-8B-Instruct	32 × 32 (all)	1025	150.0K	3000	Tables 2–4
Qwen2.5-1.5B-Instruct	28 × 12 (all)	337	60.6K	3000	Table 3
Qwen3-4B-Instruct-2507	36 × 32 (all)	1153	166.7K	3000	Tables 2–4
Gemma-4-12B-it	8 × 16 (global)	129	33.5K	2000	Tables 2–4
Qwen3.5-9B-Base	8 × 16 (full-attn)	129	33.5K	2000	Tables 2–4

I Cost of the Query-Conditioned Selectors

Our two selectors read the same signal: the attention each cached token *receives from the question window*, the last $W = 64$ prefix positions, which hold the first question q_1 at the end of the prefill. This appendix states what that signal costs. The question is not cosmetic—a selector whose score requires the full $S \times S$ attention matrix cannot run on a modern serving stack—so we make the accounting explicit and separate it per selector.

The scoring primitive. The score of cached token t for layer l and query head h is the question-window attention $a_t^{(l,h)}$ of Eq. (1). This is a column statistic of a $W \times S$ *slice* of the attention matrix—64 rows—not of the $S \times S$ matrix. The distinction is what makes the selectors deployable.

Why this is compatible with FlashAttention. FlashAttention (Dao et al., 2022; Dao, 2024) never materialises $\mathrm{softmax}(QK^\top/\sqrt{d})$: it tiles the computation, carries a running softmax normaliser in SRAM, and writes back only the output and a per-row log-sum-exp. One therefore cannot read an attention map out of a prefill. That is a real obstacle for scores accumulated over *all* past queries, as in heavy-hitter eviction (Zhang et al., 2023; Liu et al., 2023). Eq. (1) sidesteps it by construction, because it needs only W query rows. We compute it in a side pass that leaves the attention kernel untouched: during prefill we hook the W question-window rows of Q (a few MB, below), and at the pause we stream the cached K once, accumulating column sums of $\exp(Q_{\mathrm{win}}K^\top/\sqrt{d} - \mathrm{LSE})$ in tiles—the same online-softmax factorisation FlashAttention itself uses, so the $W \times S$ slice is never materialised either. The same structural property makes SnapKV's observation-window score cheap (Li et al., 2024), while Quest (Tang et al., 2024) avoids attention values altogether by bounding page scores.

A budget identity. Scoring m head-slices costs $2mWSd$ FLOPs (the softmax is linear in WS and dominated by the GEMM), whereas causal prefill attention costs $2LS^2 d_{\mathrm{q}}$ with $d_{\mathrm{q}} = Hd$. Their ratio collapses to

$$\frac{\text{scoring}}{\text{prefill attention}} = \rho\,\frac{W}{S}, \qquad \rho = \frac{m}{LH}, \tag{10}$$

where ρ is the fraction of head-slices scored. Two consequences matter. First, the relative cost *falls* as context grows—precisely the regime in which cache compression is worth doing. Second, the two selectors differ only in ρ; everything else about their cost is identical. Table 14 instantiates Eq. (10).

I.1 Retrieval-Head Selector

The training-free selector scores a token by its mean attention over a small, fixed committee of *retrieval heads*, calibrated once per model. Calibration is offline and query-agnostic: we rank all (l, h) pairs by the attention they place on held-out planted answers and keep the top ρ. This costs 24 prefills on Qwen (8 prompts × 3 needle types) and 18 on Gemma; it is a one-time, seconds-scale expense that never recurs at serving time.

Table 14: **Selector cost at $S = 32$k**, from Eq. (10) with $W = 64$. m is the number of scored (layer, head) slices out of LH (336 for Qwen; 128 for Gemma, whose selector uses only the 8 global-attention layers). The two models here are representative; every evaluated model has its own detector, with configurations in Table 13. "% prefill" is the scoring FLOPs as a fraction of one full prefill forward pass (linear layers + attention); "K read" is the key bytes the side pass streams, which bounds its wall-clock because the pass is bandwidth- rather than compute-bound. One-time costs are amortised over every future pause for that model. Figures are analytic FLOP/byte counts, not measured latency.

Model	Selector	ρ	m	scoring	% prefill	K read	one-time cost
Qwen2.5-1.5B	Retrieval-head	1%	3	1.6 GF	0.0008%	24 MiB	24 calibration prefills
	Retrieval-head	3%	10	5.4 GF	0.0028%	80 MiB	24 calibration prefills
	Retrieval-head	5%	16	8.6 GF	0.0044%	128 MiB	24 calibration prefills
	Trained MLP	100%	336	180 GF	0.095%	448 MiB	3000 steps, 61K params
Gemma-4-12B	Retrieval-head	3%	3	3.2 GF	0.0004%	48 MiB	18 calibration prefills
	Trained MLP	100%	128	137 GF	0.016%	1024 MiB	2500 steps, 34K params

Because only the committee is scored, ρ is small and Eq. (10) makes the selector nearly free: at 32k context and $\rho = 3\%$, scoring is 5.4 GFLOPs, or 0.003% of a prefill—roughly $33\times$ cheaper than scoring every head. The saving compounds on the memory side, which is what actually governs wall-clock here: the pass touches only the KV heads its committee maps to, streaming 80 MiB of keys instead of 448 MiB. Storage is negligible in both directions—the hooked Q_{win} is 5.2 MiB on Qwen and 4.0 MiB on Gemma, and the score vector is $O(S)$.

The cost profile therefore has an unusual shape: the selector is cheap enough that its overhead is not a design constraint at all, and the binding consideration is accuracy rather than compute. That is the trade this appendix exists to expose—§I.2 buys accuracy on the hard 2-hop regime at roughly $33\times$ the scoring cost, and both remain far below the prefill they ride on.

I.2 Trained MLP Selector

The learned selector scores a token with a small MLP over its *full* per-head feature vector, $\text{LN} \to \text{Lin}_{d_f \to 128} \to \text{GELU} \to \text{Lin}_{128 \to 128} \to \text{GELU} \to \text{Lin}_{128 \to 1}$, with $d_f = LH + 1$ (337 on Qwen, 129 on Gemma using its global layers) and 61K / 34K parameters. Because the first layer mixes every head, the selector needs all LH slices of Eq. (1): $\rho = 1$ by construction, and this single fact accounts for its entire cost premium over §I.1.

Concretely, at 32k context scoring costs 180 GFLOPs on Qwen and 137 GFLOPs on Gemma—still only 0.095% and 0.016% of a prefill, and 0.20% of prefill *attention*, consistent with $W/S = 0.2\%$. The ratio improves with context (3.1% of attention at 2k, 0.05% at 128k), so the learned selector is least affordable exactly where compression is least needed. Worth separating: the *learned* part is not the expense. The MLP forward is 3.9 GFLOPs at 32k, about 2% of the selector's total; feature extraction dominates by a factor of 46. Peak memory is likewise modest—the $S \times d_f$ feature matrix is 21 MiB at 32k, and because the MLP acts per token we tile over keys and score each block as it is produced, reducing the working set to 2.6 MiB for a 4k block.

Training is the one genuinely additional cost, and it is one-time and offline: 3000 steps on Qwen (2500 on Gemma) with the base model frozen, each step a single short-context prefill plus a backward pass through a 61K-parameter head—minutes on one GPU. The model is not updated, so a deployment can reuse the frozen backbone unchanged.

Finally, the cost structure suggests its own remedy. Since $\rho = 1$ is forced only by the MLP's dense first layer, pruning that layer's inputs to the highest-weight heads would interpolate between the two selectors; we have not run this ablation and flag it as future work rather than a result.

Algorithm 2 Soft-bit with a **per-turn allocation** (query-conditioned). Encoding still runs once, in line 4; the turn's query enters at lines 9–10 and its whole effect is which precomputed level line 13 gathers for each token.

Require: prefix C, tiers $\mathcal{B}$, budget $\bar{b}$, floor $\beta_{\min} = \min \mathcal{B}$

TURN 0 — prefill and encode *every* level *(once)*

1: **for** $l = 1, \ldots, L$ **do**
2: $K_l, V_l \leftarrow \textsc{Prefill}_l(C)$ ▷ HBM, fp16
3: **for all** $\beta \in \mathcal{B}$ **do**
4: $Z_l[\beta] \leftarrow \textsc{Pack}\big(Q_\beta(K_l), Q_\beta(V_l)\big)$ ▷ **quantize** *all* tokens at *every* level, HBM
5: **end for**
6: $\textsc{AsyncCopy}(Z_l \to \text{DRAM})$; $\textsc{Free}(K_l, V_l)$ ▷ HBM→DRAM, Eq. (11)$/L$ bytes
7: **end for**

TURN k — resume, re-allocate *(still no quantization)*

8: $\textsc{AsyncCopy}(Z[\beta_{\min}] \to \text{HBM})$ ▷ DRAM→HBM, $P(\beta_{\min}\mathbf{1})$ bytes
9: $Q_{\text{win}} \leftarrow \textsc{Prefill}(\text{question}; Z[\beta_{\min}])$ ▷ floor-precision pass; breaks the query/key circularity
10: $w \leftarrow \textsc{HeadScore}(Q_{\text{win}}, Z[\beta_{\min}])$; $\mathbf{b} \leftarrow \textsc{WaterFill}(w, \bar{b}, \mathcal{B})$ ▷ HBM only; $\mathbf{b}$ changes every turn
11: **for** $l = 1, \ldots, L$ **do**
12: **for all** $\beta \in \mathcal{B}$ **do**
13: $\textsc{AsyncGather}\big(Z_l[\beta], \mathcal{I}_\beta\big) \to \text{bucket}_l[\beta]$ ▷ DRAM→HBM; the tier *is* the destination, so bucketing is free
14: **end for**
15: **end for**
16: $o \leftarrow \textsc{MixedPrecAttn}(q_k, \text{bucket})$ ▷ **dequantize** per tile in SRAM (Alg. 3)
17: append turn-k KV at every level; $\textsc{AsyncCopy}$ the delta ▷ HBM→DRAM, $O(\text{turn length})$

J PER-TURN ALLOCATION: ALGORITHM AND ACCOUNTING

§4.3 allocates once, at the pause. The allocation can instead be recomputed at every pause from that turn's query, which is the policy the per-turn row of Table 2 measures. This appendix gives the pipeline and what it costs.

The obstacle, and why it is not re-encoding. Scoring a token needs this turn's question-window attention, which needs the keys—exactly what has not been fetched yet. The tempting fix, re-encoding the cache at the new allocation, would defeat the purpose: it requires P_{16} bytes on the device first, which is the transfer the allocation exists to avoid. It is also unnecessary. Both variants reconstruct a cached vector as $\hat{x}_t = Q_{b_t}^{-1}\big(Q_{b_t}(x_t)\big)$, and the query enters only through the subscript: it chooses *which* coder applies to a token, never *what* the coder computes. Encoding therefore lifts out of the per-turn loop even though the allocation does not.

Level store. Evaluating $Q_\beta(x_t)$ once for every β in the tier set $\mathcal{B}$ and keeping all $|\mathcal{B}|$ results costs

$$\textstyle\sum_{\beta\in\mathcal{B}} P(\beta\mathbf{1}) \;=\; 2LHS \sum_{\beta\in\mathcal{B}} \mathcal{C}(\beta), \tag{11}$$

which for the tier sets we use is *smaller* than the P_{16} an fp16 cache would occupy, and reduces every turn's allocation to a gather over precomputed codes. Algorithm 2 then breaks the circularity by fetching the floor level $\beta_{\min}$ of the whole cache, scoring the question against it, and upgrading only the selected tokens. Per-resume traffic is $P(\beta_{\min}\mathbf{1}) + P(\mathbf{b})$, an upper bound that double-counts the floor-level codes of the upgraded tokens. Relative to Algorithm 1 the policy pays on two axes—a larger host footprint, Eq. (11) against $P(\mathbf{b})$, and a larger resume—and on neither of the two that would disqualify it: no re-encoding, and no full-precision transfer. Deciding what to move before moving it is also InfiniGen's strategy (Lee et al., 2024), which speculates a layer's important keys from the previous layer's hidden state; the level store instead fetches a coarse copy of the *whole* cache, spending bandwidth to score every layer against the turn's true query rather than a predicted one.

K THE MIXED-PRECISION ATTENTION KERNEL

§4.3 claims that a soft-bit cache occupies $P(\mathbf{b})$ bytes of HBM rather than P_{16}. That claim is only true if the cache is never decoded in bulk. This appendix gives the kernel that makes it true (Algorithm 3) and analyses its cost; its measured decode latency is reported in Table 5.

Why a mixed allocation needs a new kernel at all. Uniform-width quantized attention is well established (Liu et al., 2024; Hooper et al., 2024): a tile of packed codes is read from HBM, expanded in SRAM, consumed, and discarded, so the decoded tensor never has an address in HBM. A soft-bit cache breaks the assumption those kernels rest on. Because b_t varies per token, the byte offset of token t is $\sum_{s<t}\lceil db_s/8\rceil$ rather than $t\lceil db/8\rceil$, so a warp reading consecutive tokens issues variable-stride loads and coalescing collapses. Writing a kernel that handles a ragged layout directly is the wrong response. The right one is to remove the raggedness.

Algorithm 3 MIXEDPRECATTN: one query block against a tier-bucketed cache. The decoded tile $\tilde{K}$ lives in registers and is overwritten each iteration; HBM holds packed codes only.

Require: query block $Q \in \mathbb{R}^{B_q \times d}$; buckets $\{(\hat{K}^{(\beta)}, \hat{V}^{(\beta)})\}_{\beta\in\mathcal{B}}$ packed in HBM; codebooks $\{\mathcal{T}_\beta\}$; rotation R

$\tilde{Q} \leftarrow RQ$ ▷ once per block; removes R^{-1} from every inner iteration
for all $\beta \in \mathcal{B}$ **do**
  $o^{(\beta)}, \ell^{(\beta)}, m^{(\beta)} \leftarrow 0,\ 0,\ -\infty$
  for each tile $\mathcal{J}$ of $\hat{K}^{(\beta)}$ **do**
    load codes of $\mathcal{J}$ ▷ HBM→SRAM, $|\mathcal{J}|\lceil d\beta/8\rceil$ bytes — the only HBM traffic
    $\tilde{K}, \tilde{V} \leftarrow \mathcal{T}_\beta[\text{codes}]$ ▷ **dequantize** in registers; never written back
    $s \leftarrow \tilde{Q}\tilde{K}^\top/\sqrt{d};\ m' \leftarrow \max(m^{(\beta)}, \max_j s_j);\ p \leftarrow e^{\,s-m'}$
    $\ell^{(\beta)} \leftarrow e^{\,m^{(\beta)}-m'}\ell^{(\beta)} + \sum_j p_j;\ o^{(\beta)} \leftarrow e^{\,m^{(\beta)}-m'}o^{(\beta)} + p\tilde{V};\ m^{(\beta)} \leftarrow m'$
  end for
end for
merge $\{(o^{(\beta)}, \ell^{(\beta)}, m^{(\beta)})\}_\beta$ by Eq. (9) ▷ exact; the FlashAttention tile rescaling
return $R^\top o$ ▷ once per block; the value-side rotation

Bucketing. Partition the cached tokens by tier, $\mathcal{I}_\beta = \{t : b_t = \beta\}$, and store each bucket contiguously. Within a bucket the width is constant, the layout is dense, and an existing uniform-width kernel applies unchanged. Two properties make this reordering admissible. First, attention over a set of keys is invariant to their order: $\mathrm{softmax}$ is applied over the key axis and the output is a weighted sum, so a permutation of the cache permutes the weights identically and leaves o fixed. Second, position is carried in the *value* of a stored key, not in its index, because RoPE has already been applied at prefill; no re-indexing or re-rotation of the *keys* is needed. Causal masking is likewise not a concern for the cached prefix, since every cached token precedes every token being generated—only the newly prefilled question tokens mask against each other, and those live in a separate full-precision buffer. In Algorithm 2 bucketing is free: $\mathcal{I}_\beta$ is already the index set of a gather, so writing that gather into a per-tier destination buffer costs nothing beyond the address arithmetic the gather performs anyway.

The *query* side of that argument does not come for free, and the distinction is the one implementation trap in this design. Stored keys keep the RoPE angle of their original position, so the decoder must keep counting in original positions; a runtime that derives the next query's position from the cache *length*—the default in HuggingFace, and the behaviour a compacting eviction method inherits—places that query at $|\mathcal{I}|$ while the keys it reads span $[0, S)$, making most relative offsets negative and out of distribution. This is silent: the shapes are right and no error is raised. We measured it while running the eviction arm of §4.3: with positions taken from the compacted length, generation degenerates into character repetition, and with true positions restored the same configuration emits well-formed answers. Any implementation that removes tokens must therefore carry the original position index alongside the surviving keys.

Folding the rotation out of the inner loop. TurboQuant-style coders (Zandieh et al., 2025) apply a random rotation R before scalar quantization, so a decoded key is $\hat{k}'_t = R\hat{k}_t$ and a naive kernel would invert R per tile. It need not. For the key side, $q^\top R^\top \hat{k}'_t = (Rq)^\top \hat{k}'_t$, so rotating the *query* once per block leaves the inner loop a plain dot product against rotated codes. For the value side, $\sum_t a_t R^\top \hat{v}'_t = R^\top\big(\sum_t a_t \hat{v}'_t\big)$, so the accumulator stays in rotated space and $R^\top$ is applied once to the finished output vector. What remains inside the loop is a table lookup into a codebook of 2^β entries—at most 512 bytes for $\beta = 8$, hence SRAM-resident.

Cost. The kernel reads $P(\mathbf{b})$ bytes per query block instead of P_{16}, which is the whole point: decode attention at long context is bandwidth-bound, its arithmetic intensity under grouped-query attention being the group size g, so time scales with bytes read and the dequantization arithmetic occupies units that are otherwise idle. The costs it adds are $|\mathcal{B}|$ kernel launches instead of one, a merge whose cost is $O(|\mathcal{B}|B_q d)$ and therefore negligible against the $O(B_q S d)$ body, and one rotation of Q and of o per block. With $|\mathcal{B}| = 3$ these are small, but they are not zero, and they set a floor on the useful number of tiers: a finer allocation grid buys distortion at the price of launches and of shorter per-bucket loops.

Measured. On one H100 at the Table 5 geometry ($L = 32$, $S = 98$k, $g = 4$), all 32 layers of a decode step take 1.41 ms at 5.2× and 1.30 ms at 11.8×, against 4.59 ms for `flash_attn_with_kvcache` over the fp16 cache, and the kernel matches a dequantise-then-attend reference to relative ℓ_2 error 5×10^{-4}. Three implementation choices carry the result. Both GEMMs run on tensor cores with the queries in the $N = 8$ slot of the MMA, so the $g = 4$ query heads share every dequantised tile. Codes are stored in MMA-fragment order within each 16-token tile—a permutation of the same bytes—and dequantised by byte permutes: a 1-bit pair costs two instructions, and the 8-bit codebook sits in shared memory laid out so that a lookup address is a single byte permute of the code. Each tier runs as its own kernel over an even split of its work, and the tiers run concurrently, since the 1-bit tier is instruction-bound while the 8-bit tier is bound by memory.