STILL: AMORTIZED KV CACHE COMPACTION IN A SINGLE FORWARD PASS

Anonymous authors
Paper under double-blind review

ABSTRACT

A bottleneck for longer-horizon deployments of LLMs is the memory stored in the KV cache. Iterative KV cache compaction can bound the KV cache growth, but existing approaches each have their tradeoffs. Selection and eviction methods are cheap but are bound to a less expressive token space, while synthesis methods utilize the continuous embedding space but often rely on costly per-context optimization. To balance both approaches, we introduce Still: a series of learned compressive maps at each layer of the KV cache, trained on a constructed MCQ dataset. We evaluate our KV compression method on Qwen and other open source models across long-document question answering, mathematical & scientific reasoning, and code generation. Still beats baseline methods at higher compression rates (8x to 200x), and on reasoning tasks can stay within 1.8 points of full-context performance on MATH-500 and GPQA-Diamond whilst using up to 3.8x less accumulated KV cache.

1 INTRODUCTION

The KV cache is the memory bottleneck to deploying large language models (LLMs) on long-horizon tasks (Kwon et al., 2023). As LLMs process longer contexts such as multi-day coding loops or reasoning over the repositories of entire organizations, their key-value (KV) caches require more memory and attention becomes more costly. In removing this bottleneck we are faced only with poor alternatives. Either a lossless KV cache that grows linearly with context length, or a lossy alternative such as fine-tuning or document summaries that abandon the structure of the model's own internal representations.

KV cache compaction has emerged as a core solution for addressing this bottleneck and existing methods fall into two categories. On one hand, selection methods (Zhang et al., 2023; Li et al., 2024; Cai et al., 2025; Kim et al., 2025) score existing cache entries and then select a subset of these tokens (Figure 1a). These methods are however restricted to entries already present in cache. On the other hand, synthesis methods (Eyuboglu et al., 2026; Zweiger et al., 2026) instead construct new keys and/or values, allowing each compact entry to represent information from multiple positions. However, these approaches require a separate optimization for each context and therefore add computation at inference time. One way of amortizing this cost is to train a compactor *across contexts*. KV-Distill (Chari et al., 2025) uses this approach in a selection method, using a learned token scorer and LoRA (Hu et al., 2022) adapters to produce a reduced cache during a single prefill.

We study the underexplored combination of *amortized synthesis of layer-wise KV caches for frozen pretrained language models* (Figure 1b). Training a compactor to synthesize a smaller KV cache at each layer allows the compactor to learn across contexts, meaning that applying it in a new context requires only a forward pass with no further optimization (Figure 1c). Such amortization is a conclusion similarly reached in representation learning, where amortized variational inference (Kingma & Welling, 2014; Rezende et al., 2014) and sparse autoencoders (Makhzani & Frey, 2014; Bricken et al., 2023) both replace repeated per-instance optimization over a fixed structure with a learned encoder.

We introduce *Still*, a KV-cache compactor built from a simplified Perceiver architecture (Jaegle et al., 2021; Alayrac et al., 2022) (Figure 2). Still consists of one module per transformer layer; at each layer being compacted, a bank of latent queries cross-attends to the cache entries being compressed and produces a smaller set of keys and values. Each compaction requires a single forward pass,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

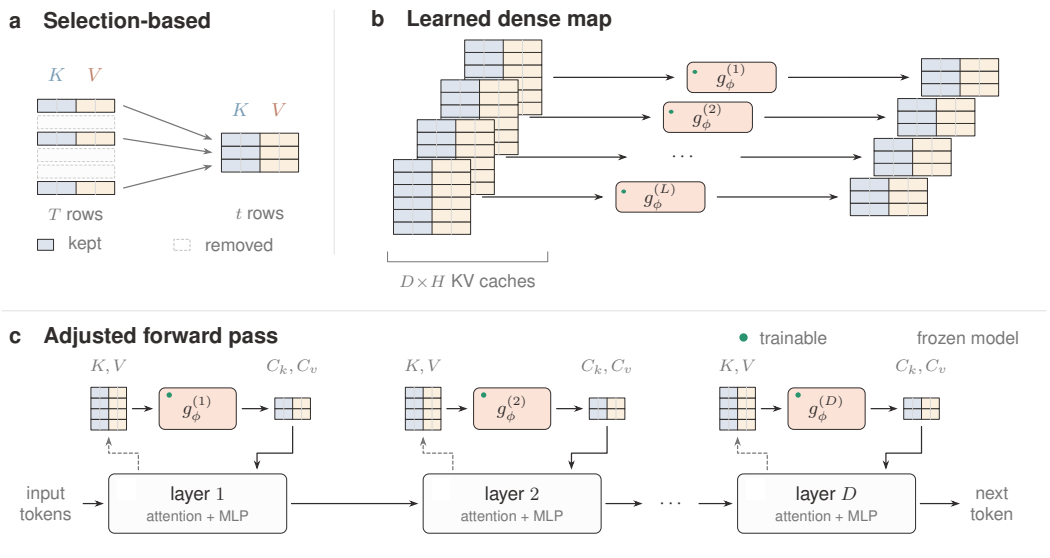


Figure 1: Selection-based and dense KV compaction. **(a)** Selection retains original cache entries (solid) and removes the rest (faded, dashed). **(b)** Still maps each layer/head cache from T entries to t synthesized entries with the same vector width. The diagonal stacks represent $D \times H$ caches, where D is model depth (L in the text) and H is the number of KV-heads. **(c)** Prefix prefill writes each layer's KV cache; Still compacts it, and subsequent tokens read the compact cache in the same frozen layer. Peach boxes denote trainable Still maps.

without collecting reference queries or optimizing the compact cache at inference time. Still can compress a prefilled context or be applied repeatedly to chunks as the context grows.

We first examine how performance changes with compression ratio (§4.1), and evaluate compactors trained separately for different base models (§4.2). We then evaluate Still's task accuracy and KV-cache usage during reasoning and code generation, comparing against uncompressed models and existing methods of managing context (§4.3). Finally, we test repeated compaction on inputs longer than those used during training (§4.4) and provide a simple ablation (§4.5).

A central question to answer with Still is whether the expressivity benefit, from both (a) learning models over dense embeddings rather than just existing tokens and (b) arbitrarily modifying individual layers' and heads' KV cache, allows higher compression ratios. Indeed, we find that Still performs favorably compared to baselines at up to 200 times compression. During reasoning, accuracy decreases by at most 1.8 points compared to full context on MATH-500 and GPQA-Diamond whilst using up to 3.8 times less accumulated KV cache.

By training the compactor across contexts, we make KV cache synthesis a reusable operation that can be applied throughout generation and offer a path toward longer-running models that retain context in significantly smaller caches.

2 Problem Formulation

In this work we learn to replace part of a language model's attention KV cache with a smaller set of keys and values, while preserving predictions on subsequent tokens. We learn a compactor that replaces a selected block of T cache entries with $t < T$ entries, giving a local compression ratio $c = T/t$. We apply this operation either once to an input prefix or repeatedly as the history grows.

Let f_θ be a frozen autoregressive transformer with L layers, H KV-heads per layer, head dimension d, and rotary position embeddings (Su et al., 2023). For each layer ℓ being compacted and KV head h, let $K^{(\ell,h)}, V^{(\ell,h)} \in \mathbb{R}^{T \times d}$ denote the keys and values of the selected block of T cache entries. When this block is the entire cache available to a query $q^{(\ell,h)}$ associated with KV head h, the attention

output is

$$a^{(\ell,h)} = \text{softmax}\left(\frac{q^{(\ell,h)}(K^{(\ell,h)})^\top}{\sqrt{d}}\right) V^{(\ell,h)}.$$

If other entries are retained, attention is computed over their concatenation with this block. We then aim to learn maps g_ϕ^ℓ on each layer across heads directly:

$$g_\phi^{(\ell)} : (K^{(\ell,h)}, V^{(\ell,h)})_{h=1}^H \mapsto (C_k^{(\ell,h)}, C_v^{(\ell,h)})_{h=1}^H,$$

with $C_k^{(\ell,h)}, C_v^{(\ell,h)} \in \mathbb{R}^{t\times d}$, and $t \ll T$. The compact entries replace the block in the cache used by f_θ to process subsequent tokens, while the base model's weights remain unchanged. Akin to Cartridges (Eyuboglu et al., 2026), the expressivity of this model comes from being able to modify individual layers' KV caches so they do not necessarily need to follow the forward pass. Differing from Cartridges, we are learning a full mapping from the uncompressed to compressed KV cache.

3 METHOD

Below we describe the architectural setup and training method for our KV cache compressor. Because we are learning individual models at each layer, the individual models are able to be quite simple while remaining expressive as a whole.

3.1 THE CASE FOR DENSE KV SYNTHESIS

With a fixed memory budget, compaction must decide what to discard. Synthesis, in contrast to pure selection methods that either have to fully retain or forget, instead writes new keys and values, so a single compacted KV pair can combine information from many original tokens. Figure 1 contrasts selection with synthesis and shows where Still acts in the frozen model.

3.2 COMPACTOR ARCHITECTURE AND POSITION HANDLING

Still uses a deliberately small compactor with only one cross-attention block per compacted layer, with no latent self-attention or feed-forward sublayers. We give this configuration the name *Fast Diamond* and depict it in Figure 2. Fast Diamond is simpler than the multi-block Perceiver architecture it is based on, and we later show in §4.5 that this simplification improves accuracy and halves the time of compaction. Simply, Fast Diamond features one compactor per layer and one set of learned latent queries per KV head. The cross-attention projections of the compactor are shared across KV heads (note that attention does not mix heads).

Formally, at each layer the compactor reads the cache as $X = [K; V] \in \mathbb{R}^{T\times 2d}$, with keys un-rotated, and maintains t latent queries $Z \in \mathbb{R}^{t\times 2d}$ per head. One cross-attention block computes

$$Z' = \text{CrossAttn}(Z, X), \tag{1}$$

with a single head: $\tilde{q} = Z$, $\tilde{k} = XW_K + b_K$, $\tilde{v} = XW_V$; queries and keys are both L2-normalized and RoPE is applied, then the cross-attention output is projected back:

$$Z' = \text{softmax}(2d\,\tilde{q}\tilde{k}^\top)\,\tilde{v} \cdot W_O. \tag{2}$$

The compactor learns Z, W_K, b_K, W_V, and W_O. Fixed output projections split the final compacted latents into compact keys and values:

$$C_k = Z'W_{\text{key}}, \qquad C_v = Z'W_{\text{value}}, \tag{3}$$

with compact keys then re-rotated at t evenly spaced positions across the block. W_{key} and W_{value} are fixed truncated identity maps, so C_k and C_v are simply the two halves of Z'.

3.3 SINGLE-PASS AND ITERATIVE COMPACTION

Given such a trained compactor, we can evaluate the compactor in the following two ways:

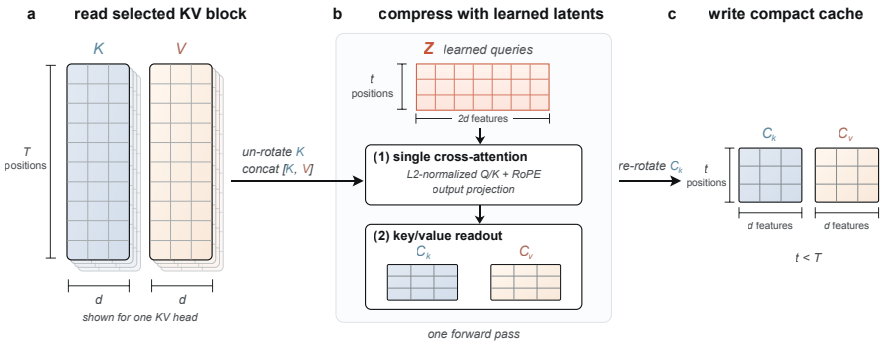


Figure 2: **Fast Diamond compaction in Still. (a)** Given a block of T KV cache entries, remove the base model's rotary embedding from the keys, and concatenate keys and values as new inputs. **(b)** For each KV head, t learned latent queries cross-attend to these entries using the compactor's own rotary encoding. **(c)** We reapply the base model's rotary encoding to the compact keys and output the compacted keys and values.

Single-pass compaction. The most straightforward evaluation: input tokens are prefilled into KV, the compactor consumes the KV and outputs compressed KV, and the base model then reads the smaller cache during generation. Subsequent tokens retain their position indices, so compaction reduces the number of stored entries without shifting RoPE positions.

Iterative chunked compaction. For long-horizon settings where context arrives over time, we alternate prefill and compaction, replacing blocks of T new KV cache entries with t compressed entries at a local compression ratio $c = T/t$. Compacted entries are retained untouched, so the cache grows with inputs at a reduced rate. Appendix B gives the schedules used in the experiments.

3.4 Training

Data. We train on multiple-choice questions from four domains, namely financial filings, Project Gutenberg literature, legal documents, and code. Each example contains an 8,192-token document, a question about a randomly selected passage within it, a set of labeled answer choices, and a response that gives the reasoning followed by the label of the correct choice (e.g., B). We discard questions that the base model answers correctly without the document. For the remaining questions, we retain only responses that select the correct answer option. We then downsample each domain to the same number of examples. For the Qwen models, we generate target answers with Qwen3-4B. Llama-3.1-8B-Instruct uses the same answers, re-tokenized with its own tokenizer. For Phi-4-reasoning-plus and Olmo-3-7B-Think, we generate target answers with the respective base model. Final dataset sizes are given in Appendix C.

Objective. The base model is frozen, and we train the compactor so that the model's predictions after reading the compact cache match its predictions after reading the full cache. For document x, question p, and target answer a, let

$$\ell(x) = \frac{1}{|a|} \sum_{i=1}^{|a|} \mathrm{KL}(f_\theta(\cdot \mid x, p, a_{<i}) \,\|\, f_\theta(\cdot \mid g_\phi(x), p, a_{<i}))\,,$$

computed over the full vocabulary. Let $\ell_0(x)$ be the corresponding KL divergence between full-context and no-document predictions, with both conditioned on the same question and preceding target-answer tokens. We minimize $\mathbb{E}[\ell(x)/\max\{\mathrm{sg}[\ell_0(x)], \epsilon\}]$, where $\epsilon = 10^{-6}$ and sg stops gradients. This measures predictive disagreement under compaction relative to the disagreement caused by removing the document entirely. Gradients pass through the frozen base model into the compact cache; only ϕ is updated.

Training setup. Unless otherwise stated, each training document of 8192 tokens is prefilled at once and independently compacted via eight 1024 token to 128 latent compactions, resulting in a final cache size of 1024 latents. Full training settings are given in Appendix C.

3.5 Experimental setup and comparisons

Reasoning and coding. We evaluate repeated compaction during generation on AIME26 (Dekoninck et al., 2026), MATH-500 (Hendrycks et al., 2021; Lightman et al., 2023), GPQA-Diamond (Rein et al., 2023), and LiveCodeBench v6 (Jain et al., 2024). Each compactor compresses 1,024-token blocks to 128 entries. Lookahead is $\Lambda = 4{,}096$ for AIME26 and LiveCodeBench, $2{,}048$ for GPQA-Diamond, and $1{,}024$ for MATH-500. LiveCodeBench v6 uses single-turn programming tasks evaluated against public and hidden tests by the benchmark evaluator. We set a maximum of $32{,}000$ generated tokens and draw 64 samples per problem for AIME26 and 2 samples for all other benchmarks. We report mean single-sample accuracy. The uncompressed control uses the same model, prompts, and sampling settings with compaction disabled. Appendix E.2 details the generation and scoring protocol.

KV-cache usage. KV cache size is measured in bytes used (at BF16 precision). Per response, accumulated usage is defined as the sum over token generation steps and peak usage is the maximum over those steps. We report r_{AUC} and r_{peak} as Still's mean usage over responses divided by the control's mean (full context), following MEMENTO (Kontonis et al., 2026). As Still and the full context base model generate separate completions, r_{AUC} depends on both retained cache occupancy and completion length. Appendix E.3 gives the layer accounting.

Comparison methods. We compare Still with MEMENTO (Kontonis et al., 2026), a technique which fine-tunes a base model to generate summaries of their reasoning and expels past reasoning after said summary is complete. Still instead trains a separate compactor to modify the cache, keeping base weights fixed. We benchmark against the published MEMENTO results, evaluated with a 32K-token generation limit and reported controls. These controls are separately fine-tuned on reasoning traces without summaries. We report each method's accuracy and KV-cache usage relative to its own control, since the underlying checkpoints differ. Our Phi evaluations use Phi-4-reasoning-plus, whereas MEMENTO uses Phi-4-reasoning.

To assess static document compaction, we compare Still with H2O (Zhang et al., 2023) at a matched retained-cache budget. Appendix D describes the baseline implementations, and Appendix E.1 provides the matched single-pass MCQ comparison.

4 Results

We first examine the effects of compression ratio (§4.1) and base model scale (§4.2), and find the value of Still in highly compressive regimes, going up to 200x compression. We then compare Still with existing methods on reasoning and coding benchmarks (§4.3) and evaluate repeated compaction of long documents (§4.4). Finally, we ablate our simplified compactor to demonstrate the benefit of our architecture choices (§4.5).

4.1 Performance scaling with compression ratio

We first examine how much task performance remains as the compressed cache becomes smaller. On Gemma-3 4B at 16k context (Figure 4), Still both performs close to full-context at 8× compression, and accuracy remains above 40 percent as compression ratios tend toward 200x. This both starts at a higher performance and trends better at higher ratios than the compared baselines. Gemma-3 interleaves sliding-window and global attention layers. Still compacts only the global layers, so its compression ratio applies to those layers.

At a fixed $8\times$ compression on Qwen3-4B, Still achieves 92.48% MCQ accuracy, compared with 98.29% for the full-context control, 90.11% for LCLM (Li et al., 2026), the strongest baseline, and 74.01% for H2O, and compacts each document in 0.05s (Table 3).

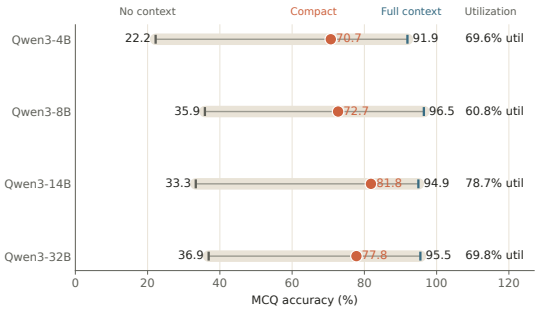


Figure 3: Still performance at a fixed 50× compression. The performance remains robust to base model size. All of the Qwen3 dense models (4B, 8B, 14B, 32B) remain fixed throughout training.

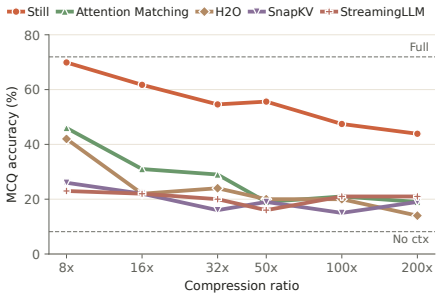


Figure 4: Still vs. baselines as the compression ratio increases. All experiments are on Gemma-3 4B starting from 16k uncompressed context.

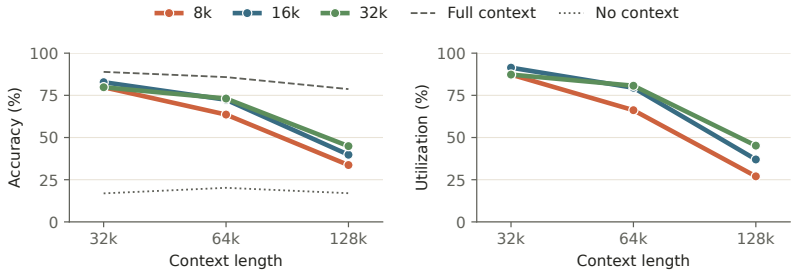


Figure 5: Iterative compaction on Long-MCQ. Color represents the document size in training, all at 8x compression and using 8 compactions (e.g. orange compresses 8192 tokens to 1024 latents via 8 iterative compactions), and the x-axis corresponds to test time document size (so at 32k for orange 8k, 32 sequential chunks are compressed). Left: compact-cache accuracy with full-context and no-context references. Right: normalized utilization.

4.2 Performance scaling with model scale

Holding architecture, training corpus, optimizer, and schedule fixed, Figure 3 reports accuracy at 50× compression across the Qwen3 dense family from 4B to 32B[1]. The compact cache remains useful at every size.

4.3 Comparison to summarization-based methods on reasoning benchmarks

We now ask, when we want to preserve the full performance of the base model, whether the compressed caches generated by Still can both (a) preserve base model performance, and (b) compete with summarization-based methods. We compare with MEMENTO, a learned compaction method that summarizes reasoning blocks before discarding their caches. Table 1 evaluates Still on AIME26, MATH-500, GPQA-D, and LiveCodeBench, alongside results from MEMENTO, across a range of models.

At 8× compaction, Still loses at most 1.8 percentage points relative to its control on MATH-500 and GPQA-D for all four models, with 2.2–3.8× less accumulated KV for Qwen and Phi (1.1–1.2× for Olmo). Losses range from 1.4 to 6.3 points on AIME26 and from 0.8 to 3.4 points on LiveCodeBench. Still uses less accumulated KV than MEMENTO in 7 of 16 comparisons (one tie) and less peak KV in 8 of 16.

Table 1: Reasoning and coding performance at lower compression ratios, compared with full context and summarization-based techniques. Accuracy change (Δ, percentage points) and accumulated KV ratio (r_{AUC}) relative to full context for each method. Still remains competitive with summarization-based techniques at lower compression ratios. Full results can be found in Appendix F.2.

		AIME'26		MATH-500		GPQA-D		LCB v6	
Model	Method	$\Delta\uparrow$	$r_{\text{AUC}}\downarrow$	$\Delta\uparrow$	$r_{\text{AUC}}\downarrow$	$\Delta\uparrow$	$r_{\text{AUC}}\downarrow$	$\Delta\uparrow$	$r_{\text{AUC}}\downarrow$
Qwen3-32B	Still	-3.3	0.45	-1.6	**0.37**	$\mathbf{-0.8}$	**0.46**	-1.9	0.51
	MEMENTO	$\mathbf{-1.5}$	**0.40**	$\mathbf{-0.7}$	0.45	-2.5	0.48	$\mathbf{-1.3}$	**0.34**
Phi-4-reasoning-plus	Still	-6.3	0.47	$\mathbf{-0.9}$	**0.26**	$\mathbf{+0.8}$	**0.36**	-3.4	0.48
	MEMENTO	$\mathbf{-2.2}$	**0.40**	$\mathbf{-0.9}$	0.53	-2.5	0.44	$\mathbf{-3.2}$	**0.35**
Qwen3-8B	Still	$\mathbf{-1.4}$	0.43	-0.9	**0.40**	$\mathbf{+2.5}$	0.44	$\mathbf{-0.8}$	0.55
	MEMENTO	-7.4	**0.34**	$\mathbf{+0.4}$	**0.40**	-2.0	**0.34**	-3.5	**0.29**
Olmo-3-7B-Think	Still	-5.2	**0.80**	-1.8	**0.89**	-1.3	**0.84**	$\mathbf{-2.2}$	0.89
	MEMENTO	$\mathbf{-4.4}$	1.02	$\mathbf{+0.7}$	0.93	$\mathbf{+3.8}$	1.06	-2.8	**0.55**

Methods use their respective checkpoints, controls, and evaluation protocols. MEMENTO values are from its published Table 1 in Kontonis et al. (2026).

4.4 Iterative compaction with Still over growing contexts

In long-horizon settings, context arrives over time and the deployment constraint is a fixed compression ratio: the retained cache grows with the trajectory at rate $1/c$ rather than at full token rate, with bounded per-chunk overhead. Per-context synthesis methods are operationally unattractive in this regime because their per-invocation optimization compounds across the trajectory. With Still, mid-trajectory compaction is a forward pass.

Figure 5 evaluates iterative chunked compaction (§3.3) on Long-MCQ (Appendix E.4), comparing Still checkpoints trained at three iterative horizons (8k, 16k, 32k) deployed at 32k, 64k, and 128k. Every training context contains eight chunks, and all three checkpoints use 8× compression: they compact 1,024-, 2,048-, and 4,096-token chunks to 128, 256, and 512 latents per chunk, respectively. All cells use the lookahead variant (one-chunk raw-KV buffer, Appendix B); the number of stacked compaction passes ranges from 8 (32k-trained at 32k) to 128 (8k-trained at 128k). At each evaluation length, the checkpoints retain the same nominal number of latent slots: 4,096 at 32k, 8,192 at 64k, and 16,384 at 128k, excluding the system prefix. The 8k-trained compactor reaches 79.8% at 32k and 33.7% at 128k, above the no-context accuracy of 17.0%. The 16k-trained checkpoint reaches 39.8% and the 32k-trained checkpoint 44.9% at 128k, with the same pattern reflected in utilization.

4.5 Simplifying the compactor

Finally, we ask whether Fast Diamond's simplifications cost accuracy. The natural design for a latent-query compressor is a Perceiver-style resampler (Jaegle et al., 2021; Alayrac et al., 2022), which stacks several attention blocks with residual updates. We therefore compare Fast Diamond with a two-block Perceiver compactor with cross-attention and latent self-attention (Appendix G). In MCQ evaluation using Qwen3-4B at $8\times$ compression, Fast Diamond achieves 92.48% accuracy, compared with 88.26% for the two-block compactor with the baseline training method, and compacts each document in 0.05 s versus 0.11 s (Table 3). In this regime, we note that "Fast" indicates the one-block architecture and "Diamond" our training method, which trains for 3,000 steps (rather than 1,500 steps) and normalizes the KL by the no-context KL. Starting from the two-block compactor with the baseline training method, changing only the architecture raises accuracy from 88.26% to 91.56%, whilst changing only the training method raises it to 89.58%. Most of the improvement therefore comes from the architecture.

[1]Qwen3 checkpoints are released under Apache-2.0 and Gemma-3 checkpoints under the Gemma license.

5 Related Work

Cache selection and synthesis. KV-cache compaction methods differ in how they construct the compact cache and when they perform the computation. *Selection* retains existing cache entries, while *synthesis* constructs new keys or values. Both can involve computation specific to each context or a compression rule learned across contexts. Here, Still combines synthesis with *amortization*, being a compactor trained across contexts that produces new cache entries in one forward pass.

H2O, SnapKV, PyramidKV, StreamingLLM, and KVZip retain entries using attention statistics or positional rules (Zhang et al., 2023; Li et al., 2024; Cai et al., 2025; Xiao et al., 2024; Kim et al., 2025). Cartridges optimizes compact keys and values for a source corpus and reuses them across queries (Eyuboglu et al., 2026). Attention Matching combines selected keys with fitted values and attention-mass biases (Zweiger et al., 2026). These methods illustrate that selection and synthesis can be combined within the same compactor.

Amortized KV compression. KV-Distill learns which token positions to retain and adapts their representations during encoding (Chari et al., 2025). Still removes the constraint caused by token-position selection, as each output slot combines information from across the cache block through cross-attention. Other methods construct compact KV states through special compression tokens. For example, CCM trains adapters that activate only on these tokens and uses their keys and values to construct and update compressed memory (Kim et al., 2024). Activation Beacon inserts special tokens into successive context chunks and learns projections that encode contextual information into their retained KV entries (Zhang et al., 2025). Still goes one step further, learning separate modules that map existing cache blocks directly to new per-layer keys and values, without inserting compression tokens or selecting token positions.

Learned representations and architectures. Learning a reusable compressor extends beyond KV cache compaction to methods that represent context through special tokens or compact embeddings. These include Gisting, which represents prompts through gist-token KV states, and AutoCompressors and ICAE, which encode context into compact vectors consumed by a language model (Mu et al., 2023; Chevalier et al., 2023; Ge et al., 2024). LCLM jointly trains an encoder and decoder around compressed latent embeddings (Li et al., 2026). Perceiver and Flamingo's Perceiver Resampler introduce the latent-query cross-attention pattern that Still uses to aggregate KV cache entries into compacted latents (Jaegle et al., 2021; Alayrac et al., 2022).

Variants of attention address KV cache cost differently. GQA shares KV heads, while multi-head latent attention (MLA) compresses and stores lower-dimensional keys and values (Ainslie et al., 2023; DeepSeek-AI, 2024). Dynamic Memory Compression learns to merge KV entries through continued pretraining (Nawrot et al., 2024), whilst Native Sparse Attention and DeepSeek-V4 combine compression with sparse attention (Yuan et al., 2025; DeepSeek-AI, 2026). Still applies its compression through a separate learned module that compacts existing KV states, all leaving the base-model frozen.

Compressing during generation. Long reasoning traces grow the KV cache during generation. MEMENTO (Kontonis et al., 2026) fine-tunes the model to write a summary after each reasoning block and then discards the block's cache entries. ThinKV (Ramachandran et al., 2026) combines quantization with eviction during reasoning, and RLKV (Du et al., 2026) learns which attention heads need access to the full history. Still also works by compacting the cache during generation, but in contrast it synthesizes new entries with a separate compactor, leaving the weights of the model unchanged and ensuring no tokens are spent writing summaries. Surprisingly, although Still is thus not designed as a summarization method, it reaches an accuracy–memory trade-off comparable to MEMENTO's at the roughly 2–4$\times$ KV reduction that MEMENTO achieves (Table 1).

6 Discussion

Still amortizes KV-cache synthesis across contexts. A compactor is trained once for each frozen base model and each subsequent compaction is a single forward pass over a block of cache entries with no per-context optimization. This means compaction is cheap enough to apply repeatedly during generation. The clearest benefit of synthesis is shown in single-pass compaction at high compression

ratios, where each compact entry must summarize many original entries (Figures 3 and 4). During generation at $8\times$, Still gives an accuracy–memory tradeoff broadly comparable to MEMENTO's (Table 1) while leaving the base model's weights unchanged, whereas MEMENTO fine-tunes the model to write summaries. The tradeoff varies by model and benchmark: accuracy losses are small on MATH-500 and GPQA-D and largest on average on AIME26. These accuracies reflect reasoning over compacted caches: across the four models and three reasoning benchmarks, 51–97% of responses were compacted at least once, with 2.6–14.3 compactions per response on average (Appendix F.2).

There are several directions we wish to investigate in future. First, we want to understand the difference between training and deployment. During training each block's keys and values are computed while attending to uncompressed history, and blocks are compacted in isolation without lookahead. In contrast, during generation, the keys and values are computed while attending to earlier compact entries, and a raw lookahead buffer is retained. Lookahead affects both accuracy and realized KV savings, and whether training under the same settings as deployment would narrow the remaining gap on AIME26 is an open question. The second area of investigation is how errors may accumulate as compactions stack over longer histories. On Long-MCQ (Figure 5) we see that, while performance decay rates grow slightly with the number of compactions, decay rate with respect to total tokens is much more important and still decays more rapidly than the full context performance. Finally, the method we utilize appends blocks such that the cache still grows linearly with trajectory; we wish to examine variants of this method that hold a compact budget by merging or reusing slots.

Limitations. Still is, after all, a compression algorithm and thus lossy. Currently its accuracy loss depends on the model as well as the benchmark and compaction schedule, and the relationship between these inputs and accuracy loss is unclear. Under the append schedule we report the retained cache grows linearly with trajectory length at rate $1/c$ and hence is not constant size memory. Additionally, in single-pass mode Still runs after a full prefill of the input, and hence can't reduce the cost of the initial forward pass (despite reducing the retained cache and later attention cost). Further, each base model needs its own trained compactor which adds parameters, training data and training cost. Furthermore we only estimate cost savings in terms of KV cache size and don't take into consideration or measure GPU memory, latency or throughput. Finally, our results use a single training seed, and the Qwen3-8B reasoning results use an earlier checkpoint (step 2,500 of 3,000) selected on benchmark accuracy (Appendix C).

7 Conclusion

Still compacts a model's KV cache in a single forward pass using one cross-attention block per layer. On static documents it outperforms selection, optimization and other learned compression baselines and degrades gradually at higher compression. During generation, it reaches an accuracy–memory tradeoff comparable to MEMENTO. We find that amortizing compaction in this way turns cache compression into a cheap operation that can run repeatedly during inference.

Reproducibility statement

We will open source all code and data when ready. We include both hyperparameters for training and compute details in Appendix C. We hope these details help future researchers to properly evaluate and build upon our work.

AI use statement

We used generative AI tools to draft and edit parts of the text and to search for related literature that we were unaware of. We used AI coding assistants (Codex and Claude) to help implement experiment code, which we tested and checked against our logged runs. The questions we used for training and evaluation, and some optimal/target responses were generated with open-weight language models as part of the data pipeline described in §3.4 and Appendix E.4, and filtered as described there. The authors reviewed all AI-assisted content and take responsibility for the final paper.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://doi.org/10.18653/v1/2023.emnlp-main.298.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: A visual language model for few-shot learning. In *Advances in Neural Information Processing Systems*, 2022. URL https://doi.org/10.52202/068431-1723.

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E. Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. Towards monosemanticity: Decomposing language models with dictionary learning, 2023. Transformer Circuits Thread, https://transformer-circuits.pub/2023/monosemantic-features/index.html.

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. In *Second Conference on Language Modeling*, 2025.

Vivek Chari, Guanghui Qin, and Benjamin Van Durme. KV-Distill: Nearly lossless learnable context compression for LLMs, 2025. URL https://doi.org/10.48550/arXiv.2503.10337.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://doi.org/10.18653/v1/2023.emnlp-main.232.

DeepSeek-AI. DeepSeek-V3 technical report, 2024. URL https://doi.org/10.48550/arXiv.2412.19437.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. arXiv preprint arXiv:2606.19348.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: MathArena as an evaluation platform for mathematics with LLMs, 2026. URL https://doi.org/10.48550/arXiv.2605.00674.

Wenjie Du, Li Jiang, Keda Tao, Xue Liu, and Huan Wang. Which heads matter for reasoning? RL-guided KV cache compression. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://doi.org/10.48550/arXiv.2510.08525.

Sabri Eyuboglu, Ryan Ehrlich, Simran Arora, Neel Guha, Dylan Zinsley, Emily Liu, Will Tennien, Atri Rudra, James Zou, Azalia Mirhoseini, and Christopher Ré. Cartridges: Lightweight and general-purpose long context representations via self-study. In *International Conference on Learning Representations*, 2026. URL https://doi.org/10.48550/arXiv.2506.06266.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model, 2024. URL https://doi.org/10.48550/arXiv.2307.06945.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset, 2021. URL https://doi.org/10.48550/arXiv.2103.03874.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://doi.org/10.48550/arXiv.2106.09685.

Andrew Jaegle, Felix Gimeno, Andrew Brock, Oriol Vinyals, Andrew Zisserman, and João Carreira. Perceiver: General perception with iterative attention. In *Proceedings of the 38th International Conference on Machine Learning*, 2021. URL https://proceedings.mlr.press/v139/jaegle21a.html.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://doi.org/10.48550/arXiv.2403.07974.

Jang-Hyun Kim, Junyoung Yeom, Sangdoo Yun, and Hyun Oh Song. Compressed context memory for online language model interaction. In *International Conference on Learning Representations*, 2024. URL https://doi.org/10.48550/arXiv.2312.03414.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. KVzip: Query-agnostic KV cache compression with context reconstruction. In *Advances in Neural Information Processing Systems*, 2025. URL https://doi.org/10.52202/085713-5585.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014. URL https://doi.org/10.48550/arXiv.1312.6114.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of permissively licensed source code, 2022. URL https://doi.org/10.48550/arXiv.2211.15533.

Vasilis Kontonis, Yuchen Zeng, Shivam Garg, Lingjiao Chen, Hao Tang, Ziyan Wang, Ahmed Awadallah, Eric Horvitz, John Langford, and Dimitris Papailiopoulos. MEMENTO: Teaching LLMs to manage their own context, 2026. URL https://doi.org/10.48550/arXiv.2604.09852.

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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Jingdi Lei, Di Zhang, Junxian Li, Weida Wang, Kaixuan Fan, Xiang Liu, Qihan Liu, Xiaoteng Ma, Baian Chen, and Soujanya Poria. δ-mem: Efficient online memory for large language models. *arXiv preprint arXiv:2605.12357*, 2026. URL https://doi.org/10.48550/arXiv.2605.12357.

Ang Li, Sean McLeish, Haozhe Chen, Nimit Kalra, Zaiqian Chen, Artem Gazizov, Venkata Anoop Suhas Kumar Morisetty, Bhavya Kailkhura, Harshitha Menon, Zhuang Liu, Brian R. Bartoldson, Tom Goldstein, Sanae Lotfi, Micah Goldblum, and Pavel Izmailov. End-to-end context compression at scale. *arXiv preprint arXiv:2606.09659*, 2026. URL https://doi.org/10.48550/arXiv.2606.09659.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, 2024. URL https://doi.org/10.52202/079017-0722.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://doi.org/10.48550/arXiv.2305.20050.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://doi.org/10.48550/arXiv.1711.05101.

Alireza Makhzani and Brendan Frey. k-Sparse autoencoders, 2014. URL https://doi.org/10.48550/arXiv.1312.5663.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. In *Advances in Neural Information Processing Systems*, 2023. URL https://doi.org/10.52202/075280-0848.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M. Ponti. Dynamic memory compression: Retrofitting LLMs for accelerated inference. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 37396–37412, 2024. URL https://proceedings.mlr.press/v235/nawrot24a.html.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. ThinKV: Thought-adaptive KV cache compression for efficient reasoning models. In *International Conference on Learning Representations*, 2026. URL https://doi.org/10.48550/arXiv.2510.01290.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level Google-proof Q&A benchmark, 2023. URL https://doi.org/10.48550/arXiv.2311.12022.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning*, 2014. URL https://proceedings.mlr.press/v32/rezende14.html.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding, 2023. URL https://doi.org/10.48550/arXiv.2104.09864.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://doi.org/10.48550/arXiv.2309.17453.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 23078–23097, 2025. URL https://doi.org/10.18653/v1/2025.acl-long.1126.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with Activation Beacon. In *International Conference on Learning Representations*, 2025. URL https://doi.org/10.48550/arXiv.2401.03462.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, 2023. URL https://doi.org/10.52202/075280-1506.

Adam Zweiger, Xinghong Fu, Han Guo, and Yoon Kim. Fast KV compaction via attention matching, 2026. URL https://doi.org/10.48550/arXiv.2602.16284.

APPENDIX

Appendices A–C describe the compactor, compaction pipeline, and training. Appendices D and E give details of other baselines and evaluation protocols, Appendix F reports additional results, and Appendix G specifies the two-block compactor compared in §4.5.

A COMPACTOR ARCHITECTURES AND IMPLEMENTATION

Appendices A.1–A.4 specify the Fast Diamond compactor introduced in §3.2. Appendix B details our compaction procedures.

A.1 INPUTS AND PARAMETER SHARING

Let d be the frozen base model's KV-head dimension, $D = 2d$ the compactor width, H the number of base-model KV heads in one transformer layer, T the source-cache length, and t the compact length. For KV head h, the base model's rotary phase is first removed from the cached key $K^{(h)}$, giving $\bar{K}^{(h)}$; values are unchanged. The compactor input is

$$X^{(h)} = [\bar{K}^{(h)}; V^{(h)}] \in \mathbb{R}^{T \times D}, \qquad h = 1, \ldots, H, \tag{4}$$

where $[\cdot;\cdot]$ denotes concatenation along the feature axis.

Every compacted transformer layer has an independent Fast Diamond module. Within that module, each base-model KV head has its own learned query bank,

$$Z^{(h)} \in \mathbb{R}^{t \times D}, \tag{5}$$

while the key, value, and output projections are shared across the H KV heads within each layer. Each head attends independently to its own cache entries. In mixed-attention models, we compact only the global-attention layers. Sliding-window layers retain their original caches.

A.2 SINGLE CROSS-ATTENTION COMPUTATION

For each base-model KV head, Fast Diamond uses a single cross-attention block with one attention head. The learned latents $Z^{(h)}$ serve directly as queries, without a learned query projection. We apply L2 normalization followed by the compactor's rotary encoding to both the queries and projected keys. For KV head h,

$$Q^{(h)} = \mathrm{RoPE}_c\Big(\mathrm{L2Norm}(Z^{(h)}), p_q\Big), \tag{6}$$

$$K_c^{(h)} = \mathrm{RoPE}_c\Big(\mathrm{L2Norm}(X^{(h)}W_K + b_K), p_k\Big), \tag{7}$$

$$Y^{(h)} = \mathrm{softmax}\Big(DQ^{(h)}K_c^{(h)\top}\Big)\, X^{(h)}W_V W_O. \tag{8}$$

Here $W_K, W_V, W_O \in \mathbb{R}^{D \times D}$ are learned projection matrices, and $b_K \in \mathbb{R}^D$ is the learned key-projection bias, broadcast across the T source rows. $\mathrm{L2Norm}$ acts row-wise. The attention logits are cosine similarities scaled by D, and the softmax is taken over the T source entries for each compact query. Appendix C gives the compactor's rotary parameter θ_c for each experimental configuration.

We split the output $Y^{(h)} \in \mathbb{R}^{t \times D}$ along its feature dimension to obtain the unrotated compact keys and compact values:

$$\bar{C}_k^{(h)} = Y^{(h)}_{:,1:d}, \qquad C_v^{(h)} = Y^{(h)}_{:,d+1:2d}. \tag{9}$$

A.3 POSITION HANDLING AND CACHE INSERTION

Fast Diamond uses two rotary coordinate systems for different purposes. Before compaction, cached keys are unrotated into $\bar{K}$ so that X contains key content in an unrotated frame. The compactor applies RoPE_c only to the normalized attention queries and projected attention keys in Equations 6–7; it never rotates the value stream.

In a standard single-pass, the compactor's keys keep their original positions (p_k), and its t queries are evenly spaced between the first and last source position (p_q), which may be fractional. The compact entries produced by Equation 9 are inserted into the base model's cache, so we spread t points evenly across the same span and round each down to obtain the output positions p_{out}. Only the compact keys are rotated with the base model's RoPE at these positions,

$$C_k^{(h)} = \text{RoPE}_{\text{base}}\left(\bar{C}_k^{(h)}, p_{\text{out}}\right), \tag{10}$$

while $C_v^{(h)}$ is inserted without rotation.

The compact cache's t entries are distributed uniformly across the full original prefix's logical length (plus any explicitly preserved entries). Continuation tokens therefore receive positions as if the uncompressed prefix were still present.

A.4 Initialization

Let $W_{K,0}$ denote the initially sampled key-projection matrix in the row-vector convention of Equation 7. The direct-query initialization constructs

$$u = \text{L2Norm}\left(W_{K,0}^\top \mathbf{1}\right) \in \mathbb{R}^D. \tag{11}$$

It then gives every row of every per-head query bank this same direction, $Z_{i,:}^{(h)} = u$, before setting $W_K = 0$, $b_K = 10u$, and $W_V = W_O = I_D$. The fixed terminal readouts implement Equation 9.

A.5 Model size

The trainable parameters for one compacted layer are Z, W_K, b_K, W_V, and W_O, for a total of

$$HtD + 3D^2 + D. \tag{12}$$

The fixed terminal K/V selectors store an additional $2Dd = D^2$ parameters per compacted layer, so after summing over all layers we arrive to a total of $L(HtD + 4D^2 + D)$ parameters for the Still model.

B Compaction pipeline details

This appendix expands §3.3.

Single-pass compaction.

1. Run the frozen base model on the prefix to obtain the uncompressed KV cache.
2. Remove the base model's positional encoding from the cached keys.
3. Concatenate keys and values into $X = [\bar{K}; V]$ and apply the per-layer compactor.
4. Apply the position encoding back to the now-compacted keys, then finally return.

For single-pass evaluations, the full prefill KV cache is compacted during training and the question or continuation is placed after the compacted cache.

Iterative compaction with one-block lookahead. When context arrives over time, we alternate prefill and compaction in blocks of $T = ct$ tokens, compressing each block to t entries at a local compression ratio c. We delay each compaction by one block, so that after n complete blocks, each compacted layer holds $n - 1$ compact blocks and one uncompressed block, or

$$L_n = (n-1)t + T \tag{13}$$

entries, excluding any prefix entries kept uncompacted. The cache then grows linearly with input length at rate $1/c$. During generation, a block is instead compacted once Λ further tokens have been computed, so one-block lookahead corresponds to $\Lambda = T$.

Bias semantics in attention. When the optional bias channel is enabled and the frozen model attends over a compact prefix followed by any uncompacted suffix, β is added to the compact-prefix logits:

$$\mathrm{Attn}\left(q;\begin{bmatrix}C_k\\K_{\mathrm{fix}}\end{bmatrix},\begin{bmatrix}C_v\\V_{\mathrm{fix}}\end{bmatrix},\begin{bmatrix}\beta\\\mathbf{0}\end{bmatrix}\right)=\frac{\sum_{j=1}^{t}\exp\left(qC_{k,j}^{\top}/\sqrt{d}+\beta_j\right)C_{v,j}+\sum_j\exp\left(qK_{\mathrm{fix},j}^{\top}/\sqrt{d}\right)V_{\mathrm{fix},j}}{\sum_{j=1}^{t}\exp\left(qC_{k,j}^{\top}/\sqrt{d}+\beta_j\right)+\sum_j\exp\left(qK_{\mathrm{fix},j}^{\top}/\sqrt{d}\right)}. \tag{14}$$

Non-compact entries carry $\beta = 0$. In the Qwen implementation this is realized by extending the cache with a per-token bias channel and adding the bias inside attention for `CompactedPrefixCache`. Multi-token prefill uses a FlexAttention score modifier when available; generation-length queries use an equivalent SDPA fallback based on dimension augmentation.

Memory footprint. In single-pass compaction, the frozen model first prefills the full prefix, and Still then replaces the cache with a compact one. Peak memory during this initial step therefore includes all model weights, both caches, and the compactor's weights and activations. Once the compactor has been built only the compact cache and any generated tokens are used.

Formally, for a model with L attention layers, H_{kv} KV heads, head dimension d, and b bytes per element, a cache with S entries per KV head occupies

$$M_{\mathrm{KV}}(S) = 2LH_{\mathrm{kv}}dbS \tag{15}$$

bytes for keys and values. For Qwen3-4B ($L = 36$, $H_{\mathrm{kv}} = 8$, $d = 128$, bf16), this is 144 KiB per token, meaning that a full 16k-token cache occupies 2.25 GiB. At $50\times$ compression, our compact cache occupies about 46 MiB. The matching Fast Diamond compactor ($t = 328$) adds about 31.3M trainable parameters (Equation 12) which equates to 60 MiB in bf16.

C Training details

C.1 Generation training

Each example contains a document, a multiple-choice question with labeled options, and a target response containing reasoning and the answer label. We construct questions from random 1k-token passages and retain an item only when the generated response is correct with the document and incorrect without it. The Qwen models share Qwen-generated target responses. Llama-3.1-8B-Instruct reuses those strings after retokenization; Phi-4-reasoning-plus and Olmo-3-7B-Think use targets generated by their respective base models.

For the 8k-context Qwen3-4B training set, we cap each of the Financial, Project Gutenberg, Code, and Legal domains at 30,136 items. This leaves 120,544 examples, or approximately 1.0B source-context tokens per pass.

We freeze the base model and train only the compactor. We split each 8,192-token example into eight 1,024-token blocks, each compacted independently to 128 entries without lookahead. The loss is full-vocabulary forward KL averaged over target-response positions; prompt and padding positions are masked. We normalize it per example by $\max(\mathrm{stopgrad}(\ell_0), 10^{-6})$, where ℓ_0 is the no-context KL defined in §3.4. Thus training uses uncompacted source blocks, while generation conditions later blocks on earlier compact states and may use lookahead $\Lambda > 0$.

We use AdamW (Loshchilov & Hutter, 2019) with $\beta = (0.9, 0.999)$, peak learning rate 3×10^{-4}, 100 warmup updates, cosine decay to 0.1 times the peak, weight decay 0.01 on matrix parameters, gradient clipping at norm 1, effective batch size 64, seed 42, and compactor RoPE base $\theta_c = 5 \times 10^6$. Generation models train for approximately 3,000 updates. For Qwen3-8B, the reasoning and code results use an earlier checkpoint, taken after 2,500 updates and selected on benchmark accuracy.

C.2 Static MCQ training

The matched single-pass MCQ comparison in Appendix E.1 uses the Qwen3-4B training set above and the following Fast Diamond recipe.

Table 2: Fast Diamond hyperparameters for the matched single-pass MCQ comparison.

Hyperparameter	Value
Base model	Qwen3-4B, frozen; BF16; FlashAttention-2
Compaction	8,192 to 1,024, single pass
Architecture	Appendix A.2; one cross-attention block, one head, width 256; no self-attention or residual
Latent queries	direct learned queries
Initialization	structured identity initialization (Appendix A.4)
K/V output projections	frozen
Compactor RoPE base	10
Training data	Financial, Gutenberg, Code, Legal
Objective	full-vocabulary forward KL averaged over target-response positions
KL normalization	per-example no-context KL, $\varepsilon = 10^{-6}$
Optimizer	AdamW, $\beta = (0.9, 0.999)$, peak LR 3×10^{-4}
Schedule	100-update warmup; cosine to 0.1 times peak
Training budget	3,000 updates; seed 42
Batch / hardware	microbatch 8, accumulation 1, 8× H200
Effective batch size	64

C.3 Compute

We estimate approximately 2,050 H100, 1,250 H200, 410 B200, and 100 B300 GPU-hours for training, data preparation, evaluation, and development, including failed attempts and retries.

D Baseline Implementation Details

We provide the details of the baseline compression methods used in Table 3 and Figure 4. Throughout, K denotes the retained KV-cache budget. H2O, SnapKV, and our baseline that keeps initial and recent tokens retain entries from the original cache without adding an attention-logit bias.

H2O (Zhang et al., 2023). Our static adaptation ranks source positions by cumulative attention. In Figure 4, we collect proxy queries by feeding the context a second time against its full cache, reserve the 64 most recent positions, and fill the remaining $K - 64$ slots with the highest-scoring older positions.

SnapKV (Li et al., 2024). For Figure 4, we score source positions using attention from the final 64 proxy queries. We smooth these scores using a moving average over five positions, retain the 64 most recent positions, and fill the remaining $K - 64$ slots with the highest-scoring older positions.

StreamingLLM (Xiao et al., 2024). For Figure 4, we retain the first $K - 64$ source positions and the final 64, preserving their original rotary positions. The original work retains four initial tokens as attention sinks by default, alongside a rolling KV cache of recent tokens and assigns positions within the cache.

Attention Matching (Zweiger et al., 2026). We use highest-attention-keys (HAK), obtaining reference queries by repeated prefill through the frozen model and selecting keys by the root mean square of their attention weights across queries. We match attention mass using bounded least squares, with $e^{-3} \leq w_j \leq e^3$ and two projected gradient steps, then set $\beta_j = \log w_j$. Compacted values are fitted by ordinary least squares to match full-cache attention outputs. In the static MCQ comparison, "off-policy" queries are collected using full caches at every layer whereas "on-policy" queries are collected layer by layer with earlier layers already compacted.

Cartridges (Eyuboglu et al., 2026). We optimize 1,024 learned KV slots per input document through a frozen Qwen3-4B model. We train with KL to the full-context teacher over synthetic self-study queries generated by the same model. As we evaluate in the online setting, we ablated training on 128 and 512 queries per document. Appendix E.1 gives the evaluation controls.

LCLM (Li et al., 2026). The reported $8\times$ compression of LCLM is a jointly-trained mean-pooled encoder and decoder. LCLM is evaluated with its own trained decoder, whereas Still uses the frozen base model.

δ-mem (Lei et al., 2026). We test three variations: *state-only*, which retains 64 scalars per layer, *hybrid*, which combines the δ-mem state with a $1{,}024$-token raw cache, and *recent-only*, which retains only the $1{,}024$-token raw cache.

E Evaluation protocols and datasets

E.1 Static MCQ evaluation

The matched single-pass comparison uses a separate held-out cohort of 256 medical transcripts (outside of the training domains) and 758 extractive MCQs. Each transcript contains 8,192 tokens, which each method compresses to a nominal budget of 1,024 cache positions. All compared methods are built off of Qwen3-4B.

We evaluate every method on the same 256 documents and 758 questions with Qwen3-4B using greedy decode. Full-context and no-context inference provide the upper and lower controls.

We report normalized utility, called *utilization* or *utility* elsewhere in our experiments as

$$U(m) = 100\frac{A_m - A_{\text{no context}}}{A_{\text{full context}} - A_{\text{no context}}}, \tag{16}$$

where A_m is the MCQ accuracy of method m. Full context and no context have utility 100% and 0%.

E.2 Reasoning and code generation

We evaluate AIME26, MATH-500, GPQA-Diamond, and LiveCodeBench v6 with temperature 0.6, top-$p = 0.95$, and top-$k = 20$, and cap generation at 32,000 tokens. Still compacts each block of $T = 1{,}024$ tokens to $t = 128$ entries, with lookahead $\Lambda = 4{,}096$ for AIME26 and LiveCodeBench, $2{,}048$ for GPQA-Diamond, and $1{,}024$ for MATH-500 (Appendix B). Only complete blocks are compacted. We draw 64 responses per AIME26 problem and 2 per problem for the other benchmarks, reporting the mean accuracy of individual responses.

E.3 Retained KV accounting

We compute retained KV usage from the recorded compaction events rather than measuring GPU memory. For a prompt of P tokens, after τ generated tokens the uncompressed control holds $P + \tau$ entries per KV head in each full (global) attention layer, whereas each compacted layer in Still holds

$$S_{\text{Still}}(\tau) = P + \tau - (T - t)\,\kappa(\tau) \tag{17}$$

entries per KV head, where $\kappa(\tau)$ is the number of compactions completed so far. A layer with H_{kv} KV heads, head dimension d, and b bytes per element stores S entries per KV head in $2H_{\text{kv}}dbS$ bytes, and we use $b = 2$ for BF16.

For each response i, we sum these bytes over the counted layers and all generated tokens to obtain the accumulated usage $M^{(i)}_{\text{AUC}}$, and take the maximum over generated tokens to obtain the peak usage $M^{(i)}_{\text{peak}}$. As in MEMENTO, we report ratios of means over responses,

$$r_{\text{AUC}} = \frac{\mathbb{E}_i[M^{(i)}_{\text{AUC,Still}}]}{\mathbb{E}_i[M^{(i)}_{\text{AUC,control}}]}, \qquad r_{\text{peak}} = \frac{\mathbb{E}_i[M^{(i)}_{\text{peak,Still}}]}{\mathbb{E}_i[M^{(i)}_{\text{peak,control}}]}. \tag{18}$$

Because Still and the control generate separate completions, both ratios reflect completion length as well as the retained cache. We count only attention KV entries. In hybrid models, we compact only the full-attention layers, and the other layers keep their native state.

E.4 LONG-MCQ

Long-MCQ contains generated extractive multiple-choice questions over long Financial, Gutenberg, Legal, and Code documents at 32k, 64k, and 128k tokens, measured with the Qwen3-4B-Instruct-2507 tokenizer. A Qwen teacher generates the questions, which are kept only after no-context filtering and teacher verification. Each domain and length targets 50 documents, and the Code split uses a Stack-v1 (Kocetkov et al., 2022) long-source regeneration to avoid auto-generated, length-biased code rows.[2]

F ADDITIONAL RESULTS

F.1 STATIC MCQ COMPARISON

Table 3 reports every method in the static MCQ comparison of §4.1, using the protocol in Appendix E.1. Appendix G defines the Gold, Fast Gold, Diamond, and Fast Diamond–KVID variants. For important notes on baseline methods:

1. LCLM uses its own trained decoder and not the vanilla Qwen decoder.
2. Long-Cartridges accuracy and utility use its native FlexQwen evaluator and its own full- and no-context baselines (746/758 and 97/758), the 9.65 s/question generation time is measured via common-H100 timing-only pass over the same 256 trained caches. We evaluate on 128 and 512 self-study examples, as we are evaluating in the online compression setting.
3. For δ-mem variations please see Appendix D, and for the baseline method see Lei et al. (2026).

Table 3: Baseline comparisons on Qwen3-4B at $8\times$ compression. For H2O, HAK, and Still we measure online compaction time, for LCLM we measure encoder and adapter compaction time, and for Cartridges we measure both synthesis and downstream optimization time.

Method	Accuracy (%)	Normalized utility (%)	Cache / compaction (s/document)	Generation (s/question)
Full-context control	98.29	100.00	0.00	8.44
Fast Diamond (one block)	92.48	93.20	0.05	10.89
Fast Gold	91.56	92.12	0.05	10.05
LCLM mean-pooling $8\times$	90.11	90.42	0.07	1.45
Fast Diamond–KVID (weight-trained)	89.31	89.49	0.06	11.88
Diamond (two blocks)	89.58	89.80	0.13	12.23
Gold (two blocks)	88.26	88.25	0.11	10.08
HAK on-policy	83.38	82.53	56.26	16.18
Cartridges–512 (long)	80.61	79.20	947.19	9.65
H2O	74.01	71.56	0.43	10.61
Cartridges–128 (short)	73.22	70.64	244.16	11.41
HAK off-policy	71.24	68.32	21.88	17.37
δ-mem + recent-1024 KV	25.59	14.84	0.42	3.11
Recent-1024 KV only	22.43	11.13	0.24	1.84
δ-mem state only	12.66	-0.31	0.64	3.53
No-context control	12.93	0.00	0.00	10.13

[2]Underlying books and code retain their source-specific terms, including the Project Gutenberg License and The Stack's terms of use and original file licenses.

F.2 Reasoning and code generation

Table 4 contains the full results from Table 1, additionally including Qwen3.8-27B and Llama models. Both Llama-3.1-8B and Llama-3.2-3B are paired with MEMENTO trained models for comparison. Table 5 reports how often Still compacted during these generations.

Table 4: Accuracy and KV usage on AIME'26, MATH-500, GPQA-D, and LiveCodeBench v6 (LCB). Still uses $8\times$ compression and generates at temperature 0.6, with lookahead $\Lambda = 4{,}096$, $1{,}024$, $2{,}048$, and $4{,}096$ tokens, respectively. Deltas are computed between the compacted accuracy and each method's own uncompacted accuracy (as MEMENTO trains base model weights).

		Still			MEMENTO		
Model	Benchmark	Accuracy / raw (Δ)	r_{AUC}	r_{peak}	Accuracy / control (Δ)	r_{AUC}	r_{peak}
Qwen3.8-27B	AIME'26	83.5 / 84.1 (-0.6)	0.46	0.48	—		
	MATH-500	97.8 / 98.6 (-0.8)	0.39	0.60	—		
	GPQA-D	83.3 / 82.1 ($+1.3$)	0.51	0.54	—		
	LCB v6	62.1 / 62.8 (-0.7)	0.47	0.43	—		
Qwen3-32B	AIME'26	69.8 / 73.2 (-3.3)	0.45	0.43	72.6 / 74.1 (-1.5)	0.40	0.44
	MATH-500	95.9 / 97.5 (-1.6)	0.37	0.49	91.1 / 91.8 (-0.7)	0.45	0.47
	GPQA-D	66.7 / 67.4 (-0.8)	0.46	0.49	62.1 / 64.6 (-2.5)	0.48	0.44
	LCB v6	50.6 / 52.5 (-1.9)	0.51	0.45	74.0 / 75.3 (-1.3)	0.34	0.37
Phi-4-reasoning-plus	AIME'26	67.9 / 74.2 (-6.3)	0.47	0.41	67.6 / 69.8 (-2.2)	0.40	0.38
	MATH-500	92.9 / 93.8 (-0.9)	0.26	0.32	89.7 / 90.6 (-0.9)	0.53	0.49
	GPQA-D	67.4 / 66.7 ($+0.8$)	0.36	0.37	61.6 / 64.1 (-2.5)	0.44	0.38
	LCB v6	37.9 / 41.3 (-3.4)	0.48	0.37	61.8 / 65.0 (-3.2)	0.35	0.35
Qwen3-8B	AIME'26	65.0 / 66.4 (-1.4)	0.43	0.40	57.3 / 64.7 (-7.4)	0.34	0.39
	MATH-500	95.1 / 96.0 (-0.9)	0.40	0.43	90.1 / 89.7 ($+0.4$)	0.40	0.47
	GPQA-D	60.4 / 57.8 ($+2.5$)	0.44	0.42	55.8 / 57.8 (-2.0)	0.34	0.35
	LCB v6	44.5 / 45.3 (-0.8)	0.55	0.45	66.5 / 70.0 (-3.5)	0.29	0.32
Olmo-3-7B-Think	AIME'26	62.7 / 67.9 (-5.2)	0.80	0.64	55.4 / 59.8 (-4.4)	1.02	0.91
	MATH-500	95.0 / 96.8 (-1.8)	0.89	0.78	91.1 / 90.4 ($+0.7$)	0.93	0.85
	GPQA-D	48.5 / 49.8 (-1.3)	0.84	0.65	49.5 / 45.7 ($+3.8$)	1.06	0.93
	LCB v6	40.4 / 42.6 (-2.2)	0.89	0.67	56.0 / 58.8 (-2.8)	0.55	0.69
Llama-3.1-8B	AIME'26	1.0 / 1.1 (-0.1)	0.33	0.31	5.3 / 1.1 ($+4.2$)†	0.95	1.07
	MATH-500	50.4 / 50.3 ($+0.1$)	0.20	0.28	63.1 / 50.3 ($+12.8$)†	0.81	1.23
	GPQA-D	28.5 / 27.5 ($+1.0$)	0.21	0.32	22.7 / 27.5 (-4.8)†	1.87	2.31
	LCB v6	12.1 / 11.8 ($+0.3$)	0.58	0.98	22.9 / 11.8 ($+11.1$)†	77.8‡	5.7
Llama-3.2-3B	AIME'26	0.8 / 0.8 (± 0.0)	0.34	0.34	3.9 / 0.8 ($+3.1$)†	1.78	1.84
	MATH-500	45.4 / 45.3 ($+0.1$)	0.20	0.36	62.0 / 45.3 ($+16.7$)†	3.12	3.59
	GPQA-D	8.1 / 7.6 ($+0.5$)	0.14	0.41	24.2 / 7.6 ($+16.7$)†	5.22	4.87
	LCB v6	9.1 / 7.4 ($+1.7$)	0.21	0.90	8.0 / 7.4 ($+0.6$)†	34.4‡	5.6

MEMENTO values are from its published Table 1 Kontonis et al. (2026). except rows marked †. For such rows, we provide our own evaluation of MEMENTO-SFT checkpoints (Llama-3.1-8B, step 174; Llama-3.2-3B, stage 2 step 100) at temperature 0.6, using the same datasets, seeds, harness, scorer, and control as the Still Llama rows. ‡ r_{AUC} can be dominated by occasional very long generations.

Table 5: Compaction exposure of the Still generations in Table 1. For each benchmark, % is the share of responses compacted at least once, and Mean is the average number of compactions per response.

	AIME'26		MATH-500		GPQA-D	
Model	%	Mean	%	Mean	%	Mean
Qwen3-32B	88.1	8.39	76.3	2.60	75.8	3.80
Phi-4-reasoning-plus	79.4	11.30	51.4	6.95	75.5	8.02
Qwen3-8B	93.3	9.81	81.6	3.87	90.9	5.71
Olmo-3-7B-Think	97.4	14.31	88.2	5.05	95.2	12.79

G ALTERNATIVE "TWO-BLOCK" STILL

Section 4.5 compares Fast Diamond with an alternative two-block compactor in the standard Perceiver form (Jaegle et al., 2021). Here we detail the architecture of this compactor and the variants used in Table 3.

Architecture. The compactor takes the same input $X = [\bar{K}; V]$ as Fast Diamond and also shares parameters across KV heads. Unlike Fast Diamond, it has two Perceiver blocks and uses learned query projections, residual updates, latent self-attention, and learned output projections for the compact keys and values. Writing Z_0 for the learned latent bank, block $b \in \{1, 2\}$ computes

$$U_b = Z_{b-1} + \mathrm{CrossAttn}_b(\mathrm{RMSNorm}(Z_{b-1}), X)\,, \tag{19}$$

$$Z_b = U_b + \mathrm{SelfAttn}_b(\mathrm{RMSNorm}(U_b))\,. \tag{20}$$

Cross-attention L2-normalizes its projected queries and keys, applies the compactor's rotary encoding, and scales the logits by the head width. Self-attention uses the same pre-RMSNorm residual form and query/key normalization however has no rotary encoding, and also no feed-forward sublayer (in either block). After the second block, learned output projections map Z_2 to the compact keys and values. In our experiments, the latent width is $d_\ell = 256$, each block has one cross-attention head and one self-attention head, and the compactor's RoPE base is $\theta_c = 10$. Like Fast Diamond, it adds no attention bias ($\beta = 0$).

Initialization. The compactor starts close to an identity map. With $d_\ell = 2d$ and one cross-attention head, the value and cross-attention output projections are initialized as identity maps over $[K; V]$, and the output projections extract the key and value halves, $W_{\mathrm{key}} = [I_d\ 0]$ and $W_{\mathrm{val}} = [0\ I_d]$. At initialization, the latent bank and the cross-attention key projection are set to zero, and the query and key biases are set to the same unit vector. Every query then matches every key equally, so attention depends only on the compactor's rotary position encoding. With a small RoPE base of 10, each latent attends to a contiguous window of nearby source entries, so each compact entry starts as a weighted average of the cache entries it replaces. The output projections of both self-attention sublayers and of the second block's cross-attention start at zero, so these sublayers initially leave the latents unchanged. At initialization, the compactor therefore reduces to the first block's cross-attention, and the other sublayers take effect only as training moves their output projections away from zero.

Parameter count. A compacted layer with latent width d_ℓ, B blocks, and n_c cross-attention and n_s self-attention heads per block has

$$Htd_\ell + B\left(2n_c d_\ell^2 + 4dn_c d_\ell\right) + B\left(4n_s d_\ell^2\right) + (2d_\ell d + d_\ell) \tag{21}$$

trainable parameters. For Qwen3-4B in Table 3 ($t = 1{,}024$, $d_\ell = 256$, $B = 2$, $n_c = n_s = 1$), this gives 116M parameters across 36 layers, contrasting to 83M for Fast Diamond (Equation 12).

Variants. We detail the variants of Table 3. Here, Gold is the two-block compactor trained with the baseline method, which uses 1,500 updates on the KL loss $\ell(x)$ of §3.4 without dividing by the no-context KL $\ell_0(x)$. *Fast* Gold uses the one-block Fast Diamond architecture with the same method. Diamond uses the two-block compactor with our method, which uses 3,000 updates on the normalized objective $\ell(x)/\ell_0(x)$ with a learning rate that decays to 0.1 times its peak. Fast Diamond combines the one-block architecture with this method. Fast Diamond–KVID uses KVID initialization, which keeps the identity K/V path but leaves the latent bank random and drops the diagonal query/key anchor.