AUTOCACHE: AGENTIC SEARCH FOR LLM PREFIX CACHE COMPRESSION

Anonymous authors
Paper under double-blind review

ABSTRACT

Storing and reusing computed states allows LLM serving systems to avoid redundant prefill computation for shared prefixes, reducing time to first token. As context lengths and the number of reusable prefixes grow, however, limited cache capacity forces evictions and expensive recomputation, making storage efficiency critical. This challenge is further complicated by modern hybrid architectures that maintain recurrent and sliding-window states in addition to KV caches, requiring compression strategies that account for model architecture, workload characteristics, and reconstruction cost. We present AUTOCACHE, an agent-driven framework that automatically discovers compression and reconstruction programs for prefix caches. Given a model, dataset, and accuracy target, AUTOCACHE explores a large combinatorial space of compression techniques to compose effective cache-compression recipes, refine existing methods, discover data and model architecture-specific compression opportunities. On the RULER benchmark with 32K context, AUTOCACHE achieves $291\times$ KV-cache compression on Qwen3.5-9B, compared with $16.0\times$ for the strongest evaluated baseline. It does so by composing factorization, quantization, entropy encoding, and protection of high-saliency cache entries in ways that would be difficult to design manually. More importantly, AUTOCACHE can seamlessly extend compression beyond KV caches to recurrent states in hybrid models. By discovering and exploiting a previously unexploited low-rank structure in the recurrent state of linear-attention layers, AUTOCACHE achieves $222\times$ compression of the total cache memory footprint. These results demonstrate the potential of automatically discovering compression algorithms that are tailored to the model architecture and workload, therefore enabling substantially more storage-efficient prefix caching for LLM serving.

1 INTRODUCTION

LLM-powered chatbots, coding assistants, and agents increasingly depend on reusing long contexts across requests. Prefix caching makes this reuse efficient by retaining the model state computed for a shared prefix, allowing subsequent requests to skip repeated computation (Zheng et al., 2024; Kwon et al., 2023). Its effectiveness, however, depends on how much reusable state the server can retain: as contexts and workloads grow, the limited cache storage triggers cache evictions, forcing the server to recompute the states that previously cached (Xie, 2025). Compression can expand effective cache capacity, provided that the retained representation preserves model quality and can be restored efficiently.

Designing such a representation requires decisions that depend on both the model and its workload. A compressor must determine which redundancies to exploit, how to combine compression mechanisms, and where approximation can be tolerated. These decisions become more complex in hybrid architectures, whose *cached state* may combine full-attention KV, sliding-window KV, and recurrent state with different structures and sensitivities (Pan et al., 2025; Jiang et al., 2023). Existing compression leverage particular mechanisms such as quantization (Liu et al., 2024b; Zandieh et al., 2025), low-rank (Chang et al., 2025; 2026) and sparsity Zhang et al. (2024); Li et al. (2024) or combine multiple mechanisms through direct composition (Staniszewski & Łańcucki, 2026)

Only the full-attention KV-cache has been tackled thus far, while other prefix cache components such as the recurrent states in linear attention variants has been largely untouched. (Section 2).

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.

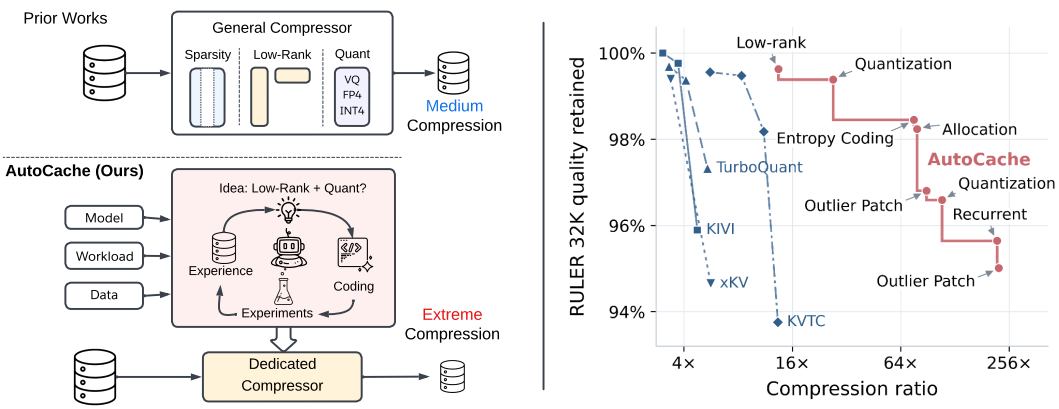

Figure 1: Left: comparison of general compression plugins (top) and agentic search (bottom). Given a model, workload, and objective, the agent iteratively proposes, implements, and evaluates compression programs to produce a dedicated compressor. Right: whole-cache compression versus retained RULER quality for Qwen3.5-9B at 32K context, where labels mark mechanisms added during the search and baselines compress only attention KV.

Also, across different workloads the different data characteristics in the cache itself can lead to different effectiveness of each mechanisms. Finding the different traits and integrating a dedicated compressor for each model and data by hand does not scale. These challenges motivate a broader research question: can we automatically design a compression recipe that adapts to the complete cached state and is tailored to the requirements of the target workload?

We present AUTOCACHE, a framework for automatically designing compression algorithms for reusable LLM state (Figure 1). Given a model, a workload, and accuracy requirements, AUTOCACHE searches for compression algorithms that determine how the prefix state is stored and reconstructed, building on agentic program search (Romera-Paredes et al., 2024; Novikov et al., 2025; Lee et al., 2026). The search can alter cache representations, compose compression mechanisms, and specialize strategies across layers and state components, allowing AUTOCACHE to explore a large combinatorial design space and discover new compression algorithms tailored to the model and data as illustrated in Fig. 1.

AUTOCACHE enables this search through an abstraction that decouples compression design from the serving system. Candidates implement only `compress()` and `decompress()`, while the framework handles architecture-specific state extraction, storage, and restoration within SGLang's prefix-cache path (Zheng et al., 2024). This abstraction allows the search to explore diverse cache representations and evaluate them directly in a high-performance inference setting, measuring both persisted-byte savings and accuracy.

Across four hybrid models and long-context benchmarks, AUTOCACHE discovers compressors adapted to different architectures and workloads. In the controlled KV-only comparison, it outperforms the strongest evaluated baseline on every model–workload pair while retaining at least 95% of uncompressed accuracy, reaching $291\times$ compression on Qwen3.5-9B compared with $16.0\times$ for the strongest human designed baseline KVTC Staniszewski & Łańcucki (2026) which combines low-rank factorization and quantization. Extending the search to recurrent and sliding-window state raises whole-cache compression from to $222\times$ (all metadata, overhead included) on Qwen, yielding a order magnitude higher compression ratios than the hand-tuned baseline which leave the recurrent states untounched. These results demonstrate the value of searching over compression designs tailored to the complete cached state.

We find that AUTOCACHE achieves these high compression rates by discovering sophisticated compression recipes that go well beyond applying existing techniques independently. The discovered recipes compose multiple compression mechanisms in non-obvious ways, specialize compression across layers and state components, and even introduce new algorithms that selectively preserve salient cache entries while aggressively compressing the remainder. Remarkably, AUTOCACHE

also discovers data-dependent signals for identifying salient entries that generalize to RULER and MRCR, despite searching with only a handful of examples. These results suggest that agentic search can uncover compression strategies that would be difficult to design manually, opening a new approach to prefix-cache optimization in modern LLM serving systems.

2 Background and Motivation

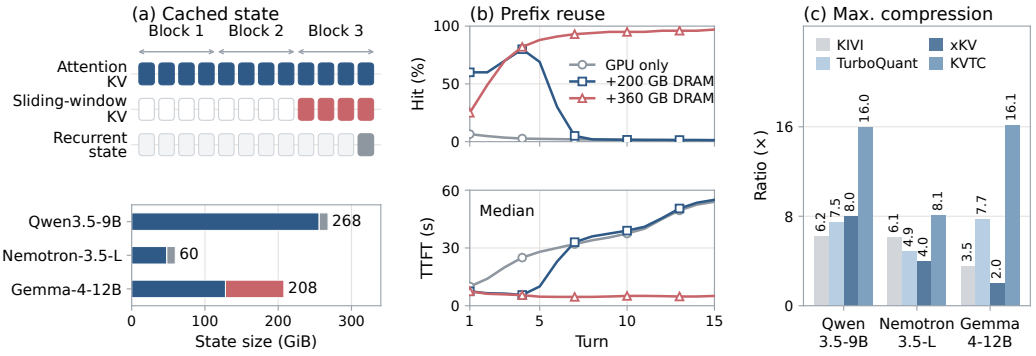

Figure 2: Cached-state composition, prefix reuse, and compressibility. (a) Tokens whose state each component retains (top), and total cached state for 256 prefixes of 32K tokens (bottom). (b) Prefix-cache hit rate and median time to first token (TTFT) per turn for Qwen3.5-9B served with SGLang HiCache (Xie, 2025) on TraceLab traces (Zhu et al., 2026). (c) Highest tested compression ratio of full-attention KV that retains at least 95% of each model's uncompressed RULER-32K accuracy (Hsieh et al., 2024).

2.1 Cached State in Modern LLMs

Prefix caching lets a server reuse computation for a shared prefix, skipping its prefill (Zheng et al., 2024; Pan et al., 2025). We call all information needed to resume from a prefix its *cached state*. In a standard Transformer, this is the KV cache, which grows with prefix length.

Hybrid models combine full attention with sliding-window attention (Jiang et al., 2023) or recurrent layers such as Mamba (Gu & Dao, 2023), Gated DeltaNet (Yang et al., 2025; Team et al., 2025) and Kimi Delta Attention (Team et al., 2025; inclusionAI, 2026). Sliding-window layers retain KV only for recent tokens, while recurrent layers maintain a fixed-size state. Hybrid cached state thus combines tensors with different shapes and statistics, with the mix varying across architectures (Table 1).

Table 1: Hybrid Model Architectures

Model	Architecture
Qwen3.5-9B Qwen Team (2026)	GDN + GQA
Nemotron-3.5-L-30B NVIDIA (2026)[1]	Mamba2 + GQA
Ling-3.0-Tiny (inclusionAI, 2026)	KDA + MLA
Gemma-4-12B-it (Gemma Team, 2026)	SWA (GQA) + global MQA

At 32K tokens, state outside full-attention KV accounts for 4.5% of the cache in Qwen3.5-9B and 38% in Gemma-4-12B-it (Figure 2(a)). If left uncompressed, even a small share can dominate the remaining cache after aggressive compression of full-attention KV. Compression must therefore cover the entire cached state.

2.2 Cache Capacity and Time to First Token

Prefix caching trades memory for computation. Every hit saves a prefill, and every miss pays for one. How often the cache hits, in turn, depends largely on how much state it can hold. Hierarchical caches such as SGLang HiCache (Xie, 2025) expand this capacity by spilling cached state from GPU memory to host memory and storage.

To see how much capacity matters in practice, we served Qwen3.5-9B with HiCache on multi-turn coding-agent traces from TraceLab (Zhu et al., 2026) (Figure 2(b)). With GPU memory alone, the

cache almost never hits, and median TTFT keeps rising as conversations grow. Adding 200 GB of host memory helps for the first few turns, but around turn 5 the conversation histories no longer fit: the hit rate drops to nearly zero, and TTFT climbs back to the GPU-only level. With 360 GB, the histories keep fitting. The hit rate rises above 90%, and median TTFT stays around 5 s, about ten times lower than in the other two settings by turn 15. On this workload, reuse falls off a cliff once the working set outgrows the cache capacity.

The sharp capacity threshold in Figure 2(b) shows that more storage translates directly into more reuse and lower TTFT. Compression offers this extra capacity without adding memory: by shrinking each cached prefix, it lets a fixed memory budget hold more reusable state and sustain prefix reuse for longer. The more compactly a compressor can store the cached state, the more of this TTFT gain it can recover.

2.3 One Compressor, Different Behaviors Across Models

Many KV-cache compressors have been proposed, each built around a particular kind of redundancy. Quantization assumes that keys and values tolerate low precision (Liu et al., 2024b; Zandieh et al., 2025), low-rank projection assumes that they lie close to a small subspace (Chang et al., 2025), and cross-layer methods assume that neighboring layers store similar information (Chang et al., 2026). Composite methods such as GEAR (Kang et al., 2024) and KVTC (Staniszewski & Łańcucki, 2026) go a step further, combining several of these ideas and adding transforms, residual correction, or entropy coding on top.

In practice, we observe that how far a compressor can go depends strongly on the model it is applied to. Figure 2(c) reports the highest compression ratio each method reaches while keeping at least 95% of the model's RULER accuracy. The same method can land in very different places across models: xKV ranks second on Qwen3.5-9B at $8.0\times$ but last on Gemma-4-12B at only $2.0\times$, likely because Gemma already shares KV across layers, leaving little cross-layer redundancy for xKV to remove. The ranking among methods shifts as well: KIVI trails TurboQuant on Qwen3.5-9B and Gemma-4-12B but outperforms it on Nemotron-3.5-L. Combining techniques does not make this go away. KVTC, which chains a transform, adaptive quantization, and entropy coding, is the strongest method on all three models, yet its ratio still drops from $16.0\times$ on Qwen3.5-9B and $16.1\times$ on Gemma-4-12B to $8.1\times$ on Nemotron-3.5-L. Compressibility thus depends on the model as much as on the compressor, and a pipeline fixed in advance suits some models better than others.

3 AutoCache

AutoCache searches for compression designs tailored to a model and workload. It combines a compression design harness, which gives a proposer agent the interface and resources to implement and test candidates, with a controller that allocates the search budget and evaluates finalized designs. Figure 3 summarizes the system.

3.1 Search Objective

Section 2 motivates compressing the complete cached state while preserving task accuracy and fast prefix reuse. For a given model and workload, let ϕ denote a compression design that maps a prefix's cached state to a stored representation and reconstructs it when the prefix is reused. Depending on the architecture, this state includes full-attention KV, sliding-window KV, and recurrent state.

We measure compression ratio $R(\phi)$ and effective decompression bandwidth $B(\phi)$, and maximize compression subject to accuracy and bandwidth floors:

$$R(\phi) = \frac{\text{original bytes}}{\text{stored bytes}}, \qquad B(\phi) = \frac{\text{original bytes}}{\text{decompression time}},$$
$$\phi^\star = \arg\max_{\phi} R(\phi) \quad \text{s.t.} \quad \mathrm{Acc}(\phi) \geq A_{\min}, \qquad B(\phi) \geq B_{\min}. \tag{1}$$

Stored bytes include all metadata required for reconstruction. $\mathrm{Acc}(\phi)$ is the task accuracy when inference resumes from the reconstructed state. The accuracy floor limits quality degradation, while the bandwidth floor constrains the cost of restoring a cached prefix.

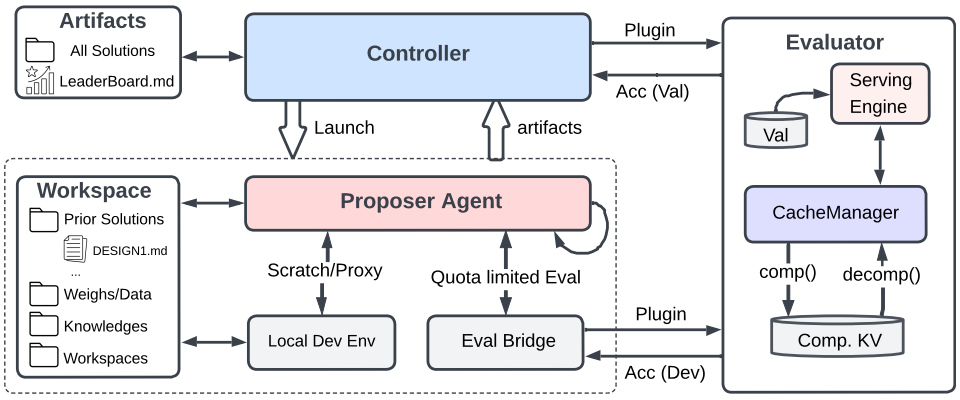

Figure 3: Overview of AUTOCACHE. The proposer develops a compressor in a harness, runs local experiments, and can request a limited number of development evaluations. Candidates implement `compress()` and `decompress()` within the serving engine's cache manager. After each session, the controller evaluates the finalized candidate and records its implementation and measurements for subsequent sessions.

3.2 COMPRESSION DESIGN HARNESS

Compression Interface. The harness isolates compression design from serving-system integration. Each candidate implements two operations: `compress()`, which maps the cached state to a stored representation, and `decompress()`, which reconstructs the state needed to resume inference. The representation may contain tensors of arbitrary shapes and data types, together with reconstruction metadata. Candidates can therefore combine compression mechanisms and specialize them across layers and state components.

We implement this interface in SGLang's cache manager (Zheng et al., 2024). On cache insertion, the manager passes the complete cached state to `compress()` and persists the returned representation. On reuse, it invokes `decompress()` and restores the reconstructed state before inference resumes. The framework handles architecture-specific cache layouts, storage, and interaction with the inference engine.

Workspace Design. Each proposer works in an isolated workspace containing the target model's weights, captured cached states, a small set of development inputs with reference answers removed, and artifacts from previous sessions. Access to the model and these examples allows the proposer to investigate compression opportunities specific to the model and workload, including outliers, low-rank structure, and cross-layer redundancy (Section 2). The proposer can freely write and run GPU programs to inspect these properties, prototype compression mechanisms, measure reconstruction error, and benchmark decompression bandwidth against $B_{\min}$. The harness provides these resources without prescribing a fixed experimental sequence. The proposer chooses which analyses to run and when to request development evaluation based on its current design hypothesis and observed results. Proposers also build their own proxies; one, for example, used the weights to recompute attention queries and weight key error by them.

Quota-limited Development Evaluation. Each proposer receives a limited number of development evaluations to refine its candidate before final submission. The proposer may submit intermediate candidates through the *Eval Bridge* in Figure 3. The evaluator runs them through the same serving path and returns aggregate task accuracy and compression ratio on the development examples. Reference answers and per-example evaluation outputs are withheld from the proposer.

This feedback allows the proposer to compare reconstruction proxies against task accuracy and revise misleading proxies. It also enables adjustment of the compression recipe, including ranks, bit widths, and other design settings. A promising mechanism may appear ineffective under an overly aggressive configuration; development evaluations allow the proposer to calibrate its operating point before assessing the mechanism itself. This reduces the risk of prematurely discarding useful mech-

anisms due to poorly chosen hyperparameters and provides the controller with a more informative basis for selecting promising designs.

3.3 SEARCH CONTROLLER

Session Lifecycle. The controller repeatedly launches proposer sessions, each of which produces one finalized compression candidate. After a session finishes, the controller freezes the implementation, independently measures its decompression bandwidth, and evaluates its task accuracy on validation examples held out from development. It records the implementation and measurements in the search history, which subsequent proposers can inspect. The validation examples and per-example outputs remain accessible only to the controller; the recorded aggregate scores guide later search. Final selection chooses the evaluated candidate with the highest compression ratio that satisfies both the accuracy and bandwidth floors.

Search Policy. The controller assigns sessions to exploration or refinement using an annealed ϵ-greedy policy. With probability ϵ, a session is an *exploration* session: it receives no designated parent and is asked to develop a new compression design, with access to the complete search history. Otherwise, it is a *refinement* session: it starts from a candidate on the current Pareto frontier of accuracy and compression ratio and is asked to improve it. When several refinement sessions run in parallel, the controller distributes them across frontier candidates. As the search progresses, ϵ decreases, shifting the budget toward refining existing designs. The search budget and development-evaluation quota are specified in Section 4.1.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Models and Datasets. We evaluate AUTOCACHE on four models spanning conventional attention and several recent hybrid architectures, as summarized in Table 1. This diversity yields substantially different forms of persistent inference state, allowing us to test whether a single search framework can discover effective compression designs across heterogeneous state representations rather than specializing to conventional KV caches only. We evaluate compression at a context length of 32K tokens using RULER (Hsieh et al., 2024) and MRCR (Vodrahalli et al., 2024). RULER is a synthetic long-context benchmark that stresses retrieval and reasoning over information distributed throughout the context. MRCR evaluates multi-round co-reference resolution and provides a complementary workload in which the model must repeatedly distinguish and recover relevant information from a long conversational context.

Search Configuration. Each proposer session has a development-evaluation quota of three submissions, and each campaign places three development samples with their labels stripped in the proposer workspace. The held-out examples and their per-example outputs are never exposed to the proposer. We use Claude Code with Opus5 as the backend. We set the decompression-bandwidth floor $B_{\min}$ to 10 GB/s in all campaigns, run three proposer sessions in parallel in each iteration, and use four H100 GPUs for the accuracy evaluations within the loop. For each campaign we search for 20 iterations. In each iteration we run 3 proposer session in parallel.

Baselines. We compare against four representative KV-cache compression methods: KIVI (Liu et al., 2024b), TurboQuant (Zandieh et al., 2025), KVTC (Staniszewski & Łańcucki, 2026), and xKV (Chang et al., 2026). These baselines operate on attention key–value caches and do not compress other forms of persistent inference state, such as Mamba state or sliding-window-attention state. Therefore, for a controlled comparison, we evaluate AUTOCACHE in a *KV-only* configuration in this experiment, restricting its search space to the attention KV cache while leaving all other persistent state uncompressed. KIVI applies low-bit quantization to cached keys and values; TurboQuant uses rotation-based low-bit quantization; KVTC combines transform-based compression with quantization; and xKV exploits redundancy across layers through low-rank representations.

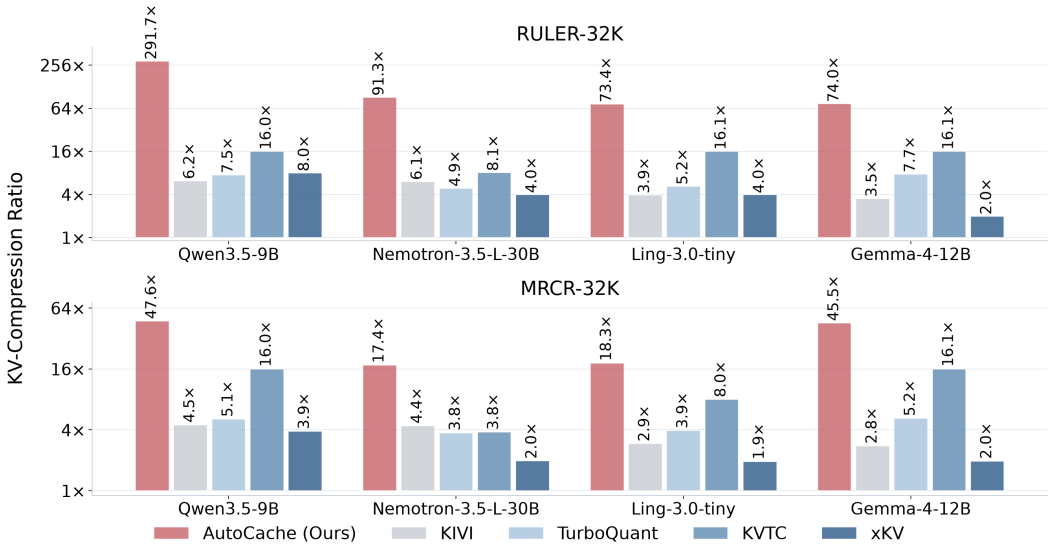

Figure 4: Main KV-Cache Results. Maximum achievable KV-Cache compression ratios subject to retaining at least 95% of each model and task's uncompressed accuracy.

4.2 Main Evaluation Results

Figure 4 reports KV-only compression at 32K context. We restrict AutoCache to the attention KV cache in this comparison so that it is directly comparable with existing KV-only baselines. Across both workloads, AutoCache consistently finds more aggressive compression schemes, while the achievable compression varies substantially across architectures.

KV-Cache Compression Results. On RULER, the largest gains are observed on Qwen3.5-9B and Nemotron-3.5-L-30B: 291.7× and 91.3× are reached by AutoCache, compared with 16.0× and 8.1× for KVTC, the strongest baseline, which combines low-rank projection with quantization. Lower compression is achieved on Ling-3.0-tiny and Gemma-4-12B, since their KV representations are already compact: MLA is used in Ling and KV sharing in Gemma, so less redundancy is left for a KV-only compressor. Even so, AutoCache still improves over the strongest baseline on both models. MRCR is harder to compress than RULER for every method. We attribute this to a difference in dataset characteristics. On RULER, the agents exploit the repetitive haystack, which compresses well, and the needle, which stands out from the haystack as an outlier and is kept at higher precision (Section 4.3). Even on MRCR, which lacks this structure, AutoCache still outperforms the strongest baseline on every model.

Whole-Cache Joint Compression Results. Figure 5 demonstrates the importance of co-designing compression across heterogeneous cache states rather than optimizing the KV cache in isolation. As KV compression becomes more aggressive, previously secondary states can become the new memory bottleneck. On Qwen3.5-9B and Ling-3.0-tiny, leaving the SSM state uncompressed limits whole-cache compression to 19.0× and 10.2× respectively, despite SSM occupying only a small fraction of the original cache. Jointly compressing KV and SSM increases these ratios to 222× and 63.2×, respectively, with near zero accuracy degradation. On Gemma, incorporating sliding window-KV compression increases the compression ratio to 119× with nearly lossless accuracy.

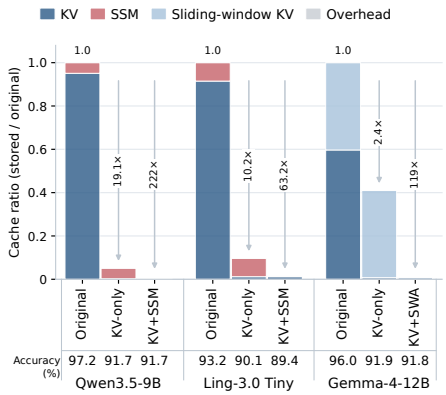

Figure 5: Whole-cache compression ratios, with accuracy (%) under each configuration.

4.3 What Does the Agent Discover and Exploit?

Novel Combinations Over a Wide-Range of Methods. AutoCache discovers compression algorithms that coordinate multiple mechanisms and adapt their settings across components. For Qwen3.5-9B on RULER, low-rank projection reduces KV dimensionality, prediction exploits redundancy across tokens in the projected representation, and entropy coding further compresses the quantized prediction residuals. The compressor selectively allocates extra bits to correct tokens poorly captured by the low-rank representation, while encoding recurrent state through separate low-rank factors. It also selects ranks and precision at a fine granularity, combining aggressive compression with targeted error correction. This example illustrates the sophistication of the discovered design: the agent determines how compression stages fit together and how precision is allocated within the resulting pipeline. (Appendix A.1)

Token Protection (RULER-Specific). On RULER, the search discovers *token protection* policy: it keeps a few tokens entries at higher precision while compressing the rest more aggressively. The Qwen3.5-9B compressor projects the keys and values onto a low-rank basis, and the tokens with the largest projection error receive an extra correction, selected without reading the question/query. Our analysis shows that on RULER's repetitive haystacks these tokens are the needle: the correction covers 0.23% of the context, including 20 of the 23 needle tokens (Appendix A.2). We read this discovery in two ways. First, AutoCache can identify a trait of the dataset and exploit it automatically. Second, it exposes a limitation of synthetic benchmarks for long-context capability: the needle can often be located from statistics of the cache alone.

Low-Rank Recurrent State (Architecture Trait). Figure 5 shows that AutoCache also compresses the recurrent state. The proposers found that each head's recurrent state is low-rank and can be reconstructed with only a few basis vectors (or singular values) (Appendix A.3). This property holds across Qwen3.5-9B, Ling-3.0-tiny, and Nemotron-3.5-L-30B, but the number of basis vectors is model-specific: the median head holds 90% of its energy in one direction on Nemotron-3.5-L-30B and in two to three on the other two models. On Qwen3.5-9B, two RULER compressors keep rank one in all or most heads, because RULER accuracy barely depends on the state. To the best of our knowledge, this is the first work discovers and exploits the low-rank property of the recurrent state.

4.4 Ablation Studies

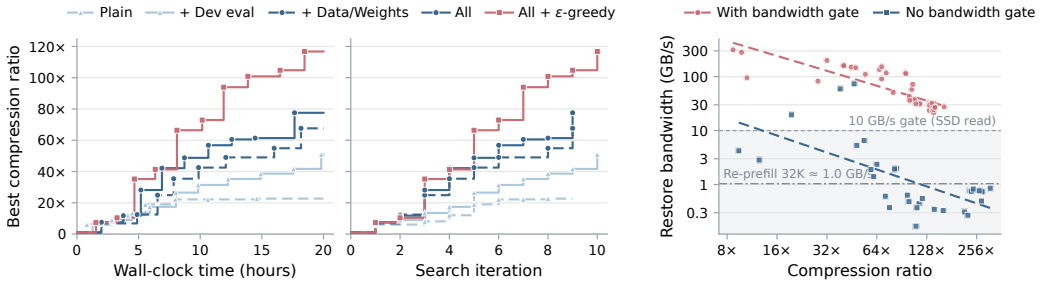

Figure 6: Ablations on Qwen3.5-9B with RULER-32K. Left and middle: best compression ratio over wall-clock time and search iteration as search components are added. Right: restore bandwidth with and without the 10 GB/s bandwidth gate; without the gate, search favors more compact but slower-to-restore designs.

We ablate each search component on Qwen3.5-9B with RULER-32K. *Plain* has neither development evaluations nor model access; + *Dev eval* and + *Weights* add one each; and *All* has both. These four allow proposers choose their own parents, while *All* + *ϵ-greedy* has the controller assign them.

Impact of Different Customized Components in the Agent Loops. Providing model access substantially improves search performance: at 20 hours, the best compression ratio increases from 22.7× to 67.6× without development evaluations, and from 50.9× to 77.6× with them. Development evaluation provides an additional improvement, raising the best ratio from 22.7× to 50.9×

without model access and from $67.6\times$ to $77.6\times$ with it, although the wall-clock curves show the additional evaluation cost. Finally, adding the controller's ϵ-greedy strategy further increases the best ratio to $116.8\times$, compared with $77.6\times$ for *All*. These trends suggest that compression design is strongly model-dependent and that repeatedly refining the current best candidate may miss better design directions. Grounding, development feedback, and explicit exploration therefore play complementary roles in the search.

Decompression Bandwidth Gate. Figure 6 (right) compares the gated *All + ϵ-greedy* campaign with a separate ungated campaign timed under the same protocol. Without an explicit bandwidth constraint, proposers are biased toward designs that maximize memory savings, even when they require expensive reconstruction. Some candidate's decompression speed even drop lower than re-prefill, losing the gain for caching. With the gate, none of 36 candidates falls below 10 GB/s, with a median of 95 GB/s in the plotted range. The fitted trends differ by more than $10\times$ across the range. Thus, the bandwidth gate changes the search preference itself, steering proposers away from compact but computationally inefficient representations toward designs that are both memory-efficient and fast to restore.

5 Related Work

KV-cache Compression. KV-cache compression has been explored along several complementary dimensions. Token-eviction methods reduce the number of cached positions by retaining tokens estimated to be important (Zhang et al., 2024); quantization methods reduce the precision of retained keys and values (Liu et al., 2024b; Zandieh et al., 2025); low-rank methods exploit structure within the hidden dimension (Chang et al., 2025); and cross-layer methods exploit redundancy between layers (Liu et al., 2024a; Chang et al., 2026). Recent work increasingly combines multiple mechanisms. GEAR (Kang et al., 2024), for example, augments quantization with low-rank and sparse error correction (Kang et al., 2024), while KVTC combines a transform, adaptive quantization, and entropy coding (Staniszewski & Łańcucki, 2026). Recent work has begun to automate parts of KV-cache optimization. KVTuner searches layer-wise precision assignments (Li et al., 2025), while EvolKV searches layer-wise cache budgets (Yu & Chai, 2025). Vortex searches programmable sparse-attention and index-selection strategies (Chen et al., 2026), and concurrent CacheCraft uses LLM-guided program evolution to discover executable token-eviction policies (Poudel et al., 2026). AutoCache differs from this prior work in the breadth of its compression-design space. Rather than fixing the compression mechanism in advance, it allows search over how KV tensors are represented in a compact form and how that representation is reconstructed for model execution rather than only over parameters or policies within a predefined family.

Agentic Search. A growing body of work uses LLM-powered agents to propose executable programs and iteratively improve them using feedback from external evaluators (Novikov et al., 2025; Cheng et al., 2025). FunSearch (Romera-Paredes et al., 2024) combines program generation with evolutionary selection, while Meta-Harness (Lee et al., 2026) enables proposers to inspect prior programs, scores, and execution traces. For LLM efficiency, CAKE (Ye et al., 2026) applies agentic search to GPU kernels, while VibeServe (Kamahori et al., 2026) uses coding agents to generate complete serving-system implementations from scratch, assuming the underlying algorithms are fixed. AutoCache extends this paradigm to cache compression, exploring compression algorithms themselves to address the cache storage challenge.

6 Conclusion

We present AutoCache, a framework for automatic designing compression recipes tailored for target architecture and data. Across four models and two long-context workloads, we found that AutoCache is capable of discovering model-specific, and data-specific compression recipes that incorporated combinations of existing and new compression methods. We believe that the discovered methods would've been impossible to design by hand, and we hope that AutoCache framework and out studies can forth a new paradigm of prefix cache compression in LLM serving systems.

AI USE STATEMENT

Generative AI was used both as a component of the research methodology and as an assistant during the development and preparation of this work.

As part of the methodology evaluated in this paper, AUTOCACHE uses coding agents based on Claude Opus 5 through Claude Code to propose, implement, analyze, and iteratively refine cache-compression programs. The agents were provided with controlled access to model artifacts, local analysis tools, and a quota-limited development evaluator. Candidate compressors produced by the agents were executed and evaluated by our external evaluation harness; reported accuracy, compression ratio, and decompression performance were obtained from these executions rather than from the agents' own assessments.

Generative AI tools, including ChatGPT and Claude, were also used by the authors during research development for brainstorming and refining research ideas, providing feedback on experimental methodology, assisting with software development and debugging, analyzing and interpreting experimental results, drafting and editing portions of the manuscript, improving presentation and organization, and assisting with the creation and refinement of figures.

The authors reviewed and revised all AI-assisted research artifacts, code, analyses, figures, and manuscript text used in the final submission. Experimental claims and quantitative results were verified using the authors' evaluation infrastructure. The authors take responsibility for the final content of the paper and for all claims and artifacts produced with the assistance of generative AI.

REFERENCES

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S. Abdelfattah, and Kai-Chiang Wu. Palu: KV-cache compression with low-rank projection. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=LWMS4pk2vK.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Hung-Yueh Chiang, Yash Akhauri, Xilai Dai, Huiqiang Jiang, Yucheng Li, Luis Ceze, Kai-Chiang Wu, and Mohamed S. Abdelfattah. xKV: Cross-layer KV-cache compression via aligned singular vector extraction. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2503.18893.

Zhuoming Chen, Xinrui Zhong, Qilong Feng, Ranajoy Sadhukhan, Yang Zhou, Michael Qizhe Shieh, Zhihao Jia, and Beidi Chen. Vortex: Efficient and programmable sparse attention serving for AI agents, 2026. URL https://arxiv.org/abs/2606.06453.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Shubham Agarwal, Mert Cemri, Bowen Wang, Alexander Krentsel, Tian Xia, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Ashwin Naren, Shulu Li, Ruiying Ma, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Let the barbarians in: How AI can accelerate systems performance research, 2025. URL https://arxiv.org/abs/2512.14806.

Gemma Team. Gemma 4 Technical Report, 2026. URL https://arxiv.org/abs/2607.02770.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2023. URL https://arxiv.org/abs/2312.00752.

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? *arXiv preprint arXiv:2404.06654*, 2024.

inclusionAI. Ling-3.0-tiny, 2026. URL https://huggingface.co/inclusionAI/Ling-3.0-tiny. Model checkpoint.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasikci. VibeServe: Can AI agents build bespoke LLM serving systems?, 2026. URL https://arxiv.org/abs/2605.06068.

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, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Xing Li, Zeyu Xing, Yiming Li, Linping Qu, Hui-Ling Zhen, Wulong Liu, Yiwu Yao, Sinno Jialin Pan, and Mingxuan Yuan. KVTuner: Sensitivity-aware layer-wise mixed-precision KV cache quantization for efficient and nearly lossless LLM inference. In *International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2502.04420.

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 *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=poE54GOq2l.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: KV cache compression in depth dimension for large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a. URL https://openreview.net/forum?id=sgVOjDqUMT.

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. *arXiv preprint arXiv:2402.02750*, 2024b.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

NVIDIA. NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16, 2026. URL https://huggingface.co/nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16. Hugging Face model card. Accessed September 26, 2026.

Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid LLMs. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

Pratik Poudel, Yanzhao Wu, Sumit Jha, and Jason Liu. Discovering KV cache eviction policies via LLM-guided program evolution, 2026. URL https://arxiv.org/abs/2608.14555.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL https://qwen.ai/blog?id=qwen3.5.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Konrad Staniszewski and Adrian Łańcucki. Kv cache transform coding for compact storage in llm inference, 2026. URL https://arxiv.org/abs/2511.01815.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Kiran Vodrahalli, Santiago Ontanon, Nilesh Tripuraneni, Kelvin Xu, Sanil Jain, Rakesh Shivanna, Jeffrey Hui, Nishanth Dikkala, Mehran Kazemi, Bahare Fatemi, Rohan Anil, Ethan Dyer, Siamak Shakeri, Roopali Vij, Harsh Mehta, Vinay Ramasesh, Quoc Le, Ed Chi, Yifeng Lu, Orhan Firat, Angeliki Lazaridou, Jean-Baptiste Lespiau, Nithya Attaluri, and Kate Olszewska. Michelangelo: Long context evaluations beyond haystacks via latent structure queries, 2024. Introduces the MRCR (multi-round co-reference resolution) benchmark.

Zhiqiang Xie. SGLang HiCache: Fast hierarchical KV caching with your favorite storage backends. LMSYS Org blog, September 2025. URL https://www.lmsys.org/blog/2025-09-10-sglang-hicache/.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2412.06464.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. CAKE: Compiler-agent co-design for frontier kernel evolution, 2026. URL https://arxiv.org/abs/2608.12629.

Bohan Yu and Yekun Chai. EvolKV: Evolutionary KV cache compression for LLM inference, 2025. URL https://arxiv.org/abs/2509.08315.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. TurboQuant: Online vector quantization with near-optimal distortion rate, 2025. URL https://arxiv.org/abs/2504.19874.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37: 62557–62583, 2024.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. TraceLab: Characterizing coding agent workloads for LLM serving, 2026. URL https://arxiv.org/abs/2606.30560.

A AGENT-DISCOVERED FINDINGS

This section details the three findings of Section 4.3: how the Qwen3.5-9B RULER compressor is composed (Appendix A.1), how it protects the needle (Appendix A.2), and how the compressors store the recurrent state (Appendix A.3).

A.1 ANATOMY OF THE QWEN3.5-9B RULER COMPRESSOR

Figure 7 shows the data flow of the reported Qwen3.5-9B RULER compressor (bold row of Table 3). It removes redundancy across layers and across tokens, then corrects the few tokens it reconstructs worst. Its thresholds, caps and weights are fixed settings chosen by the search agent, the same for every request.

Shared Structure Across Layers. Qwen3.5-9B has 32 layers. Eight are full-attention layers, which keep a KV cache; the other 24 are linear-attention layers, which keep a recurrent state instead (see State Preservation below). For each token, every full-attention layer stores a key vector and a value vector of 1,024 numbers each. A token therefore has $8 \times 2 = 16$ such vectors in the cache. We call each one a block, and each of its 1,024 numbers a channel.

The compressor first puts the 16 blocks on a common scale. It subtracts each channel's mean over the request's tokens (the means are stored) and divides each block by its root-mean-square, the square root of its average squared value. It then multiplies each block by a fixed weight. The weights were measured once, offline, by adding noise to one block at a time and recording how much the model's output changed; a block's squared weight is proportional to that change.

The 16 normalized, weighted blocks of a token are joined into one row of $16 \times 1{,}024 = 16{,}384$ numbers. Each row is projected onto a basis, a small set of directions shared by all tokens of the request, so each token becomes a short list of numbers (coefficients), one per direction. The compressor tries three bases and keeps one: (i) the request's own directions of largest variation, each stored as a full row of 16,384 numbers; (ii) directions from a fixed dictionary of 1,024 directions, computed offline from sample requests and stored inside the compressor, taken in order of how much the request varies along them, so only their positions in the dictionary are stored; and (iii) the request's directions of largest variation within the space the dictionary spans, each stored as one weight per dictionary direction. Stored basis values are rounded to integers. A basis is ruled out if its error is far worse than the best of the three: above 10% of the rows' total energy (sum of squares) and more than 5% above the smallest error. Of the rest, the compressor keeps the one with the lowest error plus λ times bits (Equation 2 below).

Predictability Across Tokens. Each coefficient is predicted from coefficients already stored, and only the difference from the prediction is kept. The prediction combines the same coefficient in the previous two tokens and in two matched earlier tokens: the earlier tokens that best explain what the previous two miss. For each matched token, the compressor stores how many tokens back it lies. The prediction weights are fitted per direction by least squares, which chooses the weights that bring the predictions closest to the actual values. Predictions are made from the values as they will be restored, not from the originals, so compression and restoration make the same prediction.

Each difference is then quantized: rounded to a whole multiple of a step and stored as an integer. Each direction has its own step, picked from a fixed list of 15 step sizes, each a multiple of that direction's spread (standard deviation); a larger step needs fewer bits but adds more error. Last, rANS, a lossless entropy coder that gives frequent integers shorter codes than rare ones, packs all stored integers: the coefficients, the basis values, the matched-token distances and the outlier patch (below). It keeps separate code tables for groups of values with different spreads, called context classes.

Selective Error Correction. After quantization, the compressor scores how well each token is reconstructed. A token's score is its reconstruction error: the difference between its original row and the row as restored before the patch is added, on the original scale and weighted by the squared block weights. A token is patched when its score is far above a typical token's, more than 12 times the median score. At most 1.25% of the tokens are patched (384 when 30,720 tokens are compressed), highest scores first; if no token qualifies, no patch is stored. For each patched token, the outlier

patch stores its position, one step, and its remaining error rounded to that step. The step is set so the patch is about as accurate as rounding the token's remaining error at 4 bits per number (the reference quantizer). Appendix A.2 describes the patch and the tokens it selects.

Restoring the Cache. Restoration reverses these steps and runs four operations in order. First, each nonzero stored integer (a coefficient, basis value or patch value) is shifted slightly toward zero, by a fraction of a step fitted per context class, so it stands for the average of the values that were rounded to it rather than the middle of its rounding interval (conditional-mean levels). Second, each block is rescaled so its energy matches the original block's (per-block gain). Third, the outlier patch is added. Last, the key-norm fix moves each restored key's length partway toward a stored target: the request's root-mean-square key length for that layer and head.

Balancing Reconstruction and Storage. Three of the choices above are made anew for every request: the basis, the rank (the number of directions kept), and the precision (each direction's step). All three follow one rule that balances reconstruction error against stored size. For a candidate choice c, the compressor minimizes

$$\mathcal{J}_\lambda(c) = E(c) + \lambda B(c), \qquad \lambda \geq 0, \tag{2}$$

where $E(c)$ is the reconstruction error (for the KV cache, measured on the normalized, weighted blocks) and $B(c)$ is the number of stored bits. The trade-off weight λ is the minimum error reduction required per extra bit: spending ΔB more bits is worthwhile only if the error falls by more than $\lambda \Delta B$. A larger λ gives a smaller output, with fewer directions and coarser steps; a smaller λ gives lower error. The same λ is used for every request; it is a fixed setting of the compressor. What differs between requests is the outcome: each request's data yield their own basis, rank and steps under that one λ.

E and B are measured in two ways. For the basis, both are measured on the finished KV payload under each basis: its bits, including the basis itself and the outlier patch, and the error left after the patch (the patched tokens counted at the reference quantizer's error). For the rank and steps, both are read from a fixed table of error and bits per step size for bell-shaped (Gaussian) values, with the error scaled by each direction's variance. The shaded boxes in Figure 7 mark the choices set this way. The outlier patch follows the fixed rules above; λ also decides whether the reference quantizer should use 2 or 8 bits instead of 4, and at the reported λ it keeps 4. The rule uses reconstruction error only; task accuracy is measured separately.

State Preservation. Each of the 24 linear-attention layers keeps a recurrent state instead of a KV cache: a fixed-size 128×128 memory matrix per head. The compressor stores each head's matrix in low-rank form, as the product of two thin matrices; their number of columns, the rank, is at most 48 of a possible 128 (Appendix A.3). Rank and precision minimize $E_{\text{state}} + \lambda_{\text{state}} B_{\text{state}}$ with $\lambda_{\text{state}} = \lambda / 20.35$, where the KV and state errors are each measured relative to their own mean square. With this smaller weight, a bit spent on the state needs only $1/20.35$ of the error reduction that a bit spent on the KV cache needs, so the state is kept more precisely. Unlike λ, the state weight can change from request to request. The compressor compares the two parts' relative errors (the square root of the mean squared error over the mean square) and keeps the state's between 0.12 and 0.40 times its estimate for the KV part, that is, between about 2.5 and 8 times more accurate. If the state falls outside this band, bisection (repeatedly halving a search range) adjusts the state weight for that request until it is back inside. The convolution state, a small buffer of recent tokens that these layers also keep, is stored raw.

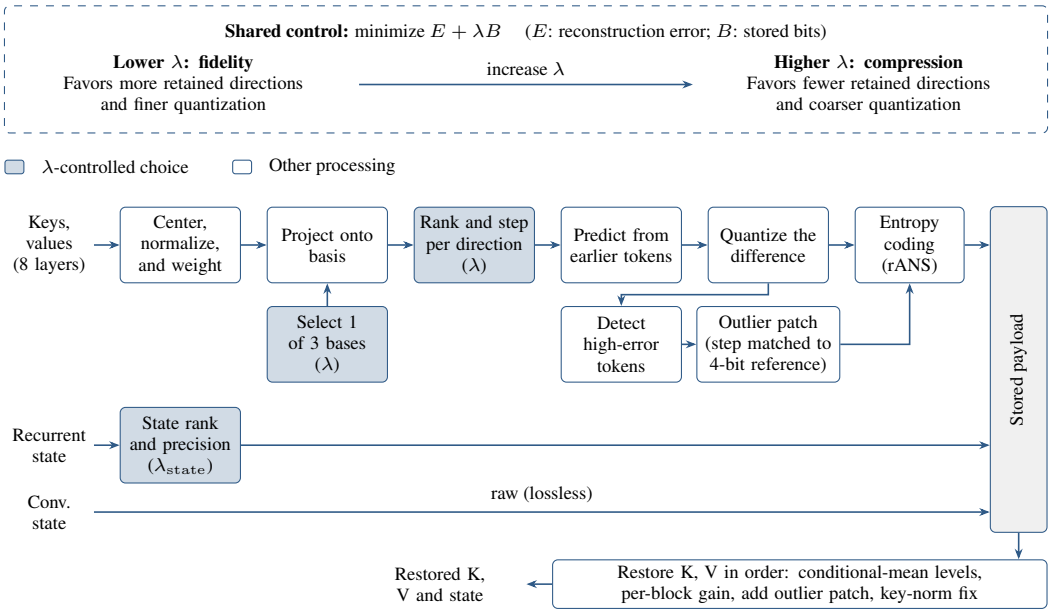

Figure 7: Qwen3.5-9B RULER compression pipeline. Shaded boxes choose the representation or precision by minimizing $E + \lambda B$; increasing λ requires a larger error reduction per extra bit. The recurrent state uses $\lambda_{\text{state}} = \lambda/20.35$, adjusted by bisection to keep its relative error in a band. The outlier patch follows fixed rules.

A.2 Token Protection on RULER

The Qwen3.5-9B RULER compressor stores an extra correction, the outlier patch, for the few tokens with the largest reconstruction error.

How the Patch Works. Let $\mathbf{u}_{t,b}$ be the cached vector of token t in block b, the keys or values of one of the 8 full-attention layers. The tokens t run over the compressed span, the part of the prompt whose cache is compressed. The compressor subtracts each channel's mean over the span's tokens (the vector $\boldsymbol{\mu}_b$, which is stored), divides each block by its root-mean-square s_b, and multiplies it by a weight w_b. It concatenates each token's blocks into one vector (row) $\mathbf{x}_t$ of length $D = 16{,}384$ and projects it onto a basis $\hat{\mathbf{V}}$, a small set of directions shared by all tokens. The basis is taken exactly as it is rebuilt from storage: the stored integers, shifted toward zero by their conditional-mean offsets (described below), serve as channel values or as mixing weights over the dictionary (the option that keeps dictionary directions stores only their positions), and each direction is scaled to unit length. The resulting coefficients, one number per direction, are quantized with Q. The basis is the one kept for the request, the winner of the three-basis trial of Appendix A.1; the patch is computed inside each trial:

$$\mathbf{x}_{t,b} = \frac{w_b}{s_b}\big(\mathbf{u}_{t,b} - \boldsymbol{\mu}_b\big), \qquad \hat{\mathbf{x}}_{t,b} = g_b\big[\, Q\big(\mathbf{x}_t \hat{\mathbf{V}}^\top\big)\, \hat{\mathbf{V}}\,\big]_b, \qquad \mathbf{r}_t = \mathbf{x}_t - \hat{\mathbf{x}}_t. \tag{3}$$

Here Q stands for all the operations applied to the coefficients (prediction from earlier tokens, quantization, and the conditional-mean offsets). The per-block gain g_b is one factor per block that rescales the reconstructed block so its energy matches the original's, and $\hat{\mathbf{x}}_t$ is the reconstructed row before the patch. When the cache is restored, the patch is added after this gain is applied and before the key-norm fix, which moves each restored key's length partway toward a stored target length. The squared weight w_b^2 is proportional to the block's sensitivity, measured offline as the change in model output when noise is added to that block alone, and floored at a tenth of the largest. Each token's score is its reconstruction error e_t ("residual energy" in Figure 8) before the patch and the key-norm fix: each block's squared residual, multiplied by s_b^2, summed over blocks. This equals the w_b^2-weighted squared error of its reconstructed keys and values:

$$e_t = \sum_b s_b^2 \,\|\mathbf{r}_{t,b}\|^2 = \sum_b w_b^2 \,\|\mathbf{u}_{t,b} - \hat{\mathbf{u}}_{t,b}\|^2, \tag{4}$$

where $\hat{\mathbf{u}}_{t,b} = \boldsymbol{\mu}_b + (s_b/w_b)\,\hat{\mathbf{x}}_{t,b}$ is the reconstructed block before the patch. The patched set $\mathcal{P}$ holds the tokens scoring above $\kappa = 12$ times the median over the span's T tokens, capped at the $N = \mathrm{round}(\rho T)$ highest, with $\rho = 1.25\%$:

$$\mathcal{P} = \mathrm{top}_N\big\{\, t : e_t > \kappa \cdot \mathrm{median}_s\ e_s \,\big\}. \tag{5}$$

For each $t \in \mathcal{P}$, the compressor stores the position t, a step Δ_t, and the integer residual $\mathbf{q}_t$. The integers are packed by rANS, the lossless entropy coder also used for the coefficients and the basis, which gives frequent integers shorter codes than rare ones, with a separate code table for each context class. The class of each 256-channel group of $\mathbf{q}_t$ is set by the spread of the residual in that group relative to Δ_t and is stored with the patch. When the cache is restored, the patch is added back to $\hat{\mathbf{x}}_t$:

$$\mathbf{q}_t = \mathrm{round}\big(\mathbf{r}_t/\Delta_t\big), \qquad \Delta_t = c\,\frac{\|\mathbf{r}_t\|}{\sqrt{D}}, \qquad \tilde{\mathbf{x}}_t = \hat{\mathbf{x}}_t + \Delta_t \mathbf{q}_t. \tag{6}$$

The constant c starts at the value for which the standard estimate for evenly spaced rounding (an average squared error of one twelfth of the squared step per value) makes the patch error $\sum_{t\in\mathcal{P}} \|\Delta_t \mathbf{q}_t - \mathbf{r}_t\|^2$ equal ϵ_b. Here ϵ_b is the total error that a b-bit reference quantizer makes on the same residuals $\mathbf{r}_t$, $t \in \mathcal{P}$; this quantizer rounds each value to one of 2^b evenly spaced levels, with its own step for each group of 256 channels of a token. The compressor shrinks c by 10% at a time only if the measured patch error exceeds ϵ_b, which never happens on the 11 saved development caches. The reference width is $b = 4$ bits. The compressor also tests 2 and 8 bits with the trade-off weight λ of Appendix A.1. For each patched token and width, it compares the error reduction, $\|\mathbf{r}_t\|^2$ minus the reference quantizer's error on $\mathbf{r}_t$ (both on the normalized row, without the factor s_b^2 of e_t), with λ times the size of a plain fixed-width field: b bits per channel plus a 16-bit step per 256 channels, not the bits the patch takes after rANS packing. At the reported λ no patched token's reduction exceeds that amount at any width, so b stays 4. When the cache is restored, each nonzero integer of $\mathbf{q}_t$ is also shifted toward zero by a fitted fraction of a step (less than one half), one fraction per context class, approximating the average of the values rounded to it (conditional-mean levels); the equations above omit this shift. If no token clears the threshold, the request carries no patch. The score reads only the cached keys and values of the compressed span; the question lies in the uncompressed remainder of the prompt.

The Patch Lands on the Needle. Figure 8 shows one `niah_single_1` example. The haystack repeats five sentences, and the needle is the only line that breaks the pattern. The patch admits 72 of 30,720 tokens (0.23%) and covers 20 of the 23 needle tokens, including the key and every digit of the answer. The three it misses ("is", a space, and the final period) score just under the threshold. The best needle token ranks 6th. Of the other 52 patched tokens, 38 lie in the system prompt and instructions at the start of the span, 5 in the first two haystack lines right after them, 4 within about 110 tokens after the needle, and 5 elsewhere in the haystack.

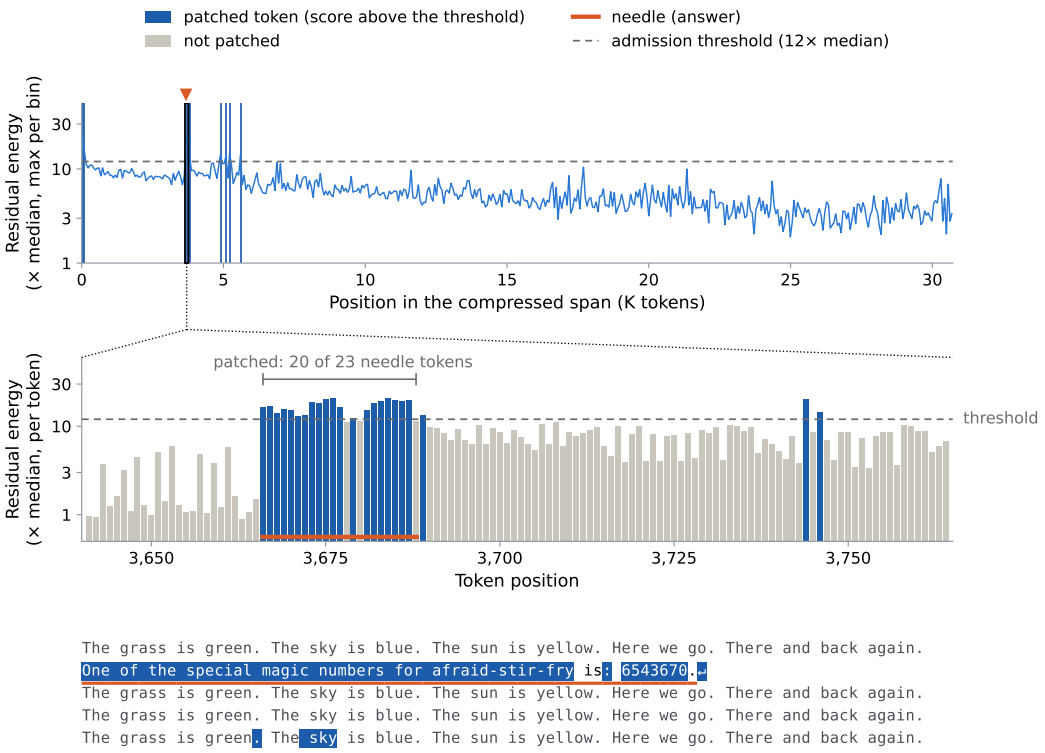

Figure 8: The outlier patch on one `niah_single_1` example. Top: the whole span. Middle: token scores around the needle. Bottom: the same tokens as text, with patched tokens filled and the needle underlined.

A.3 Low-Rank Recurrent State

Three of the four models (Table 1) keep a recurrent state, a fixed-size memory matrix per head, beside the KV cache: Mamba2 in Nemotron-3.5-L-30B (64×128 per head), Gated DeltaNet (Yang et al., 2025) in Qwen3.5-9B, and Kimi Delta Attention (KDA) (Team et al., 2025) in Ling-3.0-tiny (both 128×128 per head). KV-only compressors leave this state uncompressed, which caps whole-cache compression (Figure 5).

Prevalence. The discovered compressors often store each head's state in low-rank form, as the product of two thin matrices. This holds for the best compressor of each Nemotron-3.5-L-30B campaign that met its accuracy floor, of each of the six Qwen3.5-9B campaigns we inspected, and of the two Ling-3.0-tiny campaigns that compress the state most. Across these compressors, the mean stored rank (directions kept per head) ranges from about one to 16; individual heads keep between zero and 64.

Spectrum. Figure 9 shows how much of each head's energy its top k singular directions hold: a head's energy is the sum of its squared singular values, and its top k singular directions are the k directions of the state matrix that hold the most of it. In Nemotron-3.5-L-30B, one direction holds 97.5% of the median head's energy and two hold 99%. In Qwen3.5-9B and Ling-3.0-tiny, one direction holds 70–82%; the median head needs two to three directions to reach 90% and 10–13 to reach 99%. Each model's RULER and MRCR curves differ by at most 6.3 points.

Effect of Rank on Accuracy. In Figure 10, only the state rank changes. On 48 MRCR development examples, we run the prompt to the end of the span the serving engine compresses, replace each head's state by its best rank-k approximation (the closest matrix built from its top k singular directions, $k \in \{1, 2, 4, 8, 16\}$) or by zeros, then process the rest of the prompt with the KV cache

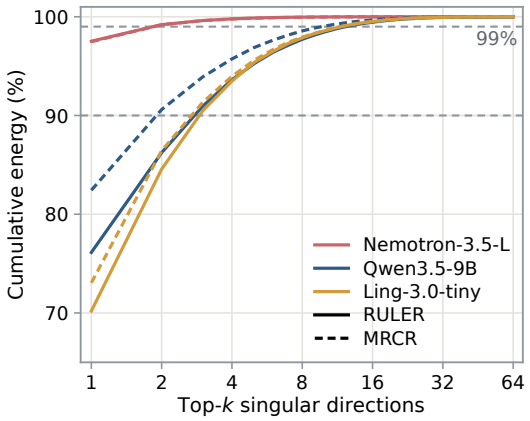

Figure 9: Share of each head's recurrent-state energy held by its top k singular directions, median over heads. Every median exceeds 99.9% at $k = 64$.

and convolution state exact and generate the answer greedily. Each p-value comes from a two-sided paired sign-flip test, which randomly flips the signs of the per-example changes and reports how often the mean change is at least as large, in absolute value, as the observed one.

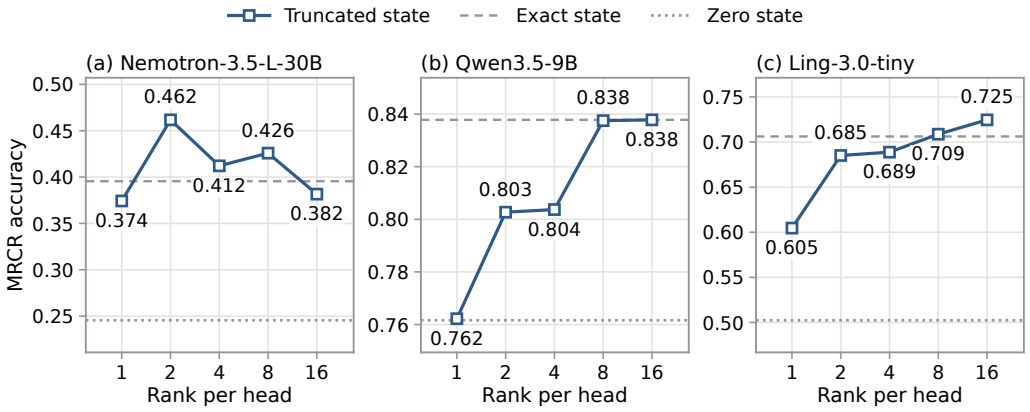

Figure 10: MRCR accuracy when only the recurrent state is truncated to rank k.

A zero state lowers accuracy by 0.15 on Nemotron-3.5-L-30B and 0.20 on Ling-3.0-tiny (both $p < 0.001$), and by 0.08 on Qwen3.5-9B ($p = 0.09$). Rank one lowers it by 0.08 on Qwen3.5-9B, 0.10 on Ling-3.0-tiny (both $p = 0.06$), and 0.02 on Nemotron-3.5-L-30B ($p = 0.61$). From rank two on, every change lies between -0.04 and $+0.07$ (all $p > 0.05$). In separate RULER runs, a zero state changes Ling-3.0-tiny's accuracy by $+0.01$ ($p = 0.52$, 176 development examples), and five Qwen3.5-9B compressors with the same KV payload (within 0.012%) and states ranging from one value per head to a mean rank of 8 score within 0.014 of each other on the 704 validation requests.

B Compression Ratio Accounting

Every compression ratio in this paper is original bytes divided by stored bytes, summed over all prefixes cached during an evaluation run. Stored bytes include the compressed payload and everything needed to reconstruct it: per-prefix metadata, tensors shared across prefixes (such as a fitted basis), and the persistent decompression workspace. Shared tensors and the workspace are charged once per run. We report two ratios.

KV Ratio. Original full-attention KV bytes divided by the KV payload plus all metadata, shared tensors, and workspace. Recurrent and sliding-window state are left out of both sides, but all shared overhead is charged to KV, including any part that serves other state. This is the ratio in Figures 4 and 11, and the one comparable with the baselines, which compress only KV.

Whole-Cache Ratio. All cached state divided by all stored bytes. Both sides add the recurrent state, the sliding-window KV, and the convolution state of hybrid models. The framework keeps the convolution state losslessly and never passes it to `compress()`, so it is stored at $1\times$. This is the ratio in Figure 5.

Why the Whole-Cache Ratio Is Lower. Table 2 works through the Qwen3.5-9B RULER compressor with the highest KV ratio. Its recurrent state compresses even better than its KV ($414\times$ against $365\times$ on payload alone), so adding it does not lower the ratio: KV and recurrent state together reach $295.9\times$. The drop to $222.1\times$ comes from the convolution state. It is 0.1% of the original bytes but, stored raw, 25% of the stored bytes.

Table 2: Byte accounting for the Qwen3.5-9B RULER compressor with the highest KV ratio, summed over 704 ruler samples (GB $= 10^9$ bytes).

Component	Original (GB)	Stored (GB)
Full-attention KV	701.02	1.921
Recurrent (GDN) state	35.43	0.086
Convolution state (lossless)	0.83	0.830
Per-prefix metadata	–	0.099
Shared tensors	–	0.067
Decompression workspace	–	0.316
Total	737.28	3.319

Ratio	Value
KV payload only: $701.02/1.921$	$364.9\times$
KV ratio: $701.02/(1.921 + 0.099 + 0.067 + 0.316)$	$291.7\times$
KV and recurrent state: $736.45/2.489$	$295.9\times$
Whole cache: $737.28/3.319$	$222.1\times$

C Full Accuracy–Compression Curves

We first show the accuracy–compression trade-offs, then report the underlying per-task scores for RULER (Appendix C.2) and MRCR (Appendix C.3). Each benchmark is organized by model, in the same order as Figure 11.

C.1 Accuracy–Compression Trade-offs

Figure 11 shows the full accuracy–compression curves behind the operating points in Figure 4. Baseline curves sweep each method's compression settings; AutoCache curves trace the Pareto frontier of the compressors discovered for each model and benchmark.

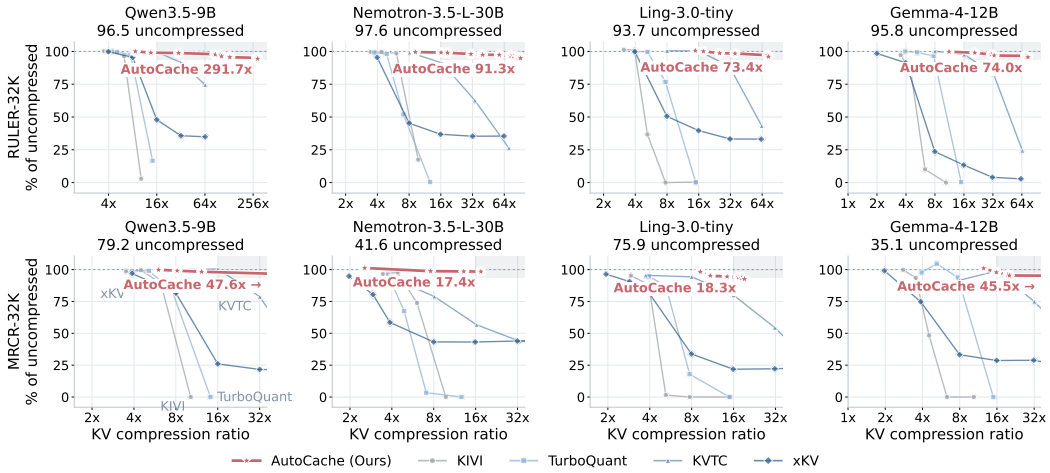

Figure 11: Accuracy–compression curves at 32K context. Accuracy is relative to the uncompressed score on the corresponding evaluation split. KV ratios include the overheads defined in Appendix B. Labels mark the operating points in Figure 4.

The largest gains from AutoCache occur at high compression. Below approximately $16\times$, KVTC matches or slightly exceeds AutoCache's accuracy on several model–benchmark pairs. The labeled AutoCache points reach 73.4–$291.7\times$ on RULER while retaining 94.4–96.0% of uncompressed accuracy. On MRCR, the corresponding ratios are lower for all four models, ranging from 17.4 to $47.6\times$ while retaining 94.0–98.5% of uncompressed accuracy.

AutoCache is evaluated on the first 64 validation examples per task, and baselines on the full validation split. Each curve is normalized to its own uncompressed reference. Selecting the AutoCache frontier on the same examples used to score it may overestimate accuracy on unseen requests.

C.2 RULER: PER-TASK ACCURACY

Tables 3–6 report every RULER curve point, one model per table. S1–S3 are single-needle tasks; MK1–MK2, MV, and MQ are multi-key, multi-value, and multi-query tasks. VT denotes variable tracking, CWE common words, and QA1–QA2 question answering.

Table Conventions. The RULER and MRCR tables report absolute accuracy (%). Within each method, rows are ordered by measured KV compression ratio (KV CR); bold KV CR marks the point in Figure 4. Baseline labels give bit width (b) or target compression ratio (×). KIVI uses groups of 32, xKV uses groups of 4, and KVTC uses its online basis. Shaded rows form the AUTOCACHE frontier, with one compressor per row. Each table includes both uncompressed evaluation references.

Table 3: RULER-32K: Qwen3.5-9B. Per-task accuracy (%) at each KV compression ratio.

		Needle retrieval							Other tasks				
Method	KV CR	S1	S2	S3	MK1	MK2	MV	MQ	VT	CWE	QA1	QA2	Avg.
Uncompressed	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	98.7	92.9	70.5	96.6
Uncompressed†	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	98.9	95.3	75.0	97.2
AUTOCACHE†	8.6×	100.0	98.4	100.0	100.0	100.0	99.6	100.0	99.7	98.3	96.9	76.6	97.2
	13.5×	100.0	100.0	100.0	100.0	100.0	98.8	100.0	100.0	97.7	92.2	70.3	96.3
	30.0×	100.0	100.0	98.4	98.4	96.9	100.0	100.0	100.0	98.6	92.2	71.9	96.0
	84.9×	100.0	98.4	96.9	98.4	96.9	98.4	99.6	99.7	97.2	92.2	68.8	95.1
	86.5×	100.0	98.4	96.9	98.4	96.9	99.6	98.8	100.0	97.3	89.1	68.8	94.9
	99.8×	96.9	100.0	96.9	100.0	100.0	97.7	100.0	95.3	97.5	87.5	70.3	94.7
	103.1×	100.0	98.4	96.9	96.9	95.3	96.5	97.7	88.1	98.3	93.8	67.2	93.5
	129.6×	100.0	96.9	89.1	96.9	98.4	97.7	96.1	99.7	95.8	90.6	65.6	93.3
	131.1×	100.0	98.4	89.1	95.3	96.9	97.7	93.4	99.7	94.1	92.2	65.6	92.9
	282.3×	100.0	100.0	85.9	98.4	96.9	99.6	100.0	99.7	92.3	78.1	65.6	92.4
	289.3×	100.0	100.0	85.9	98.4	96.9	98.8	100.0	100.0	92.3	71.9	65.6	91.8
	291.7×	100.0	100.0	85.9	98.4	95.3	99.2	100.0	99.7	91.6	76.6	62.5	91.7
KIVI 4b	3.5×	100.0	100.0	100.0	100.0	100.0	99.6	100.0	99.1	98.8	92.9	74.1	96.8
KIVI 3b	4.5×	100.0	100.0	100.0	100.0	100.0	99.1	100.0	98.8	98.6	92.9	72.3	96.5
KIVI 2b	**6.2**×	98.2	99.1	96.4	100.0	99.1	99.6	99.8	95.2	87.1	81.2	68.8	93.1
KIVI 1b	10.2×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	19.6	10.7	2.8
TurboQuant 4b	3.9×	100.0	100.0	100.0	99.1	99.1	99.6	100.0	100.0	98.8	93.8	68.8	96.3
TurboQuant 3b	5.1×	100.0	100.0	100.0	98.2	100.0	99.8	100.0	99.3	98.5	91.1	70.5	96.1
TurboQuant 2b	**7.5**×	100.0	98.2	100.0	98.2	98.2	98.7	99.3	97.9	95.1	86.6	64.3	94.2
TurboQuant 1b	14.2×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	92.0	84.8	16.1
KVTC 8×	8.0×	100.0	100.0	100.0	100.0	100.0	98.2	99.8	99.5	98.8	91.1	72.3	96.3
KVTC 16×	**16.0**×	100.0	100.0	100.0	100.0	100.0	99.8	100.0	98.8	98.6	90.2	70.5	96.2
KVTC 32×	31.9×	100.0	95.5	97.3	99.1	100.0	96.9	99.6	31.8	98.3	86.6	67.9	88.4
KVTC 64×	64.6×	55.4	78.6	57.1	81.2	91.1	76.1	91.1	14.1	95.5	78.6	72.3	71.9
xKV 4×	4.0×	100.0	100.0	100.0	100.0	100.0	99.3	100.0	99.6	98.7	91.1	71.4	96.4
xKV 8×	**8.0**×	100.0	94.6	98.2	96.4	97.3	90.4	98.2	86.8	97.9	86.6	67.9	92.2
xKV 16×	16.0×	29.5	25.9	25.9	33.0	85.7	37.3	28.6	32.1	97.5	50.9	62.5	46.3
xKV 32×	32.1×	22.3	25.0	25.9	28.6	20.5	30.1	23.7	37.3	90.2	23.2	52.7	34.5
xKV 64×	64.4×	22.3	25.0	25.9	28.6	17.9	30.1	25.2	39.1	88.8	17.9	50.0	33.7

†First 64 validation examples per task (704 requests); other rows use all 1,232 requests. Bold KV CR identifies the point in Figure 4.

Table 4: RULER-32K: Nemotron-3.5-L-30B. Per-task accuracy (%) at each KV compression ratio.

		Needle retrieval							Other tasks				
Method	KV CR	S1	S2	S3	MK1	MK2	MV	MQ	VT	CWE	QA1	QA2	Avg.
Uncompressed	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.9	91.1	84.8	97.6
Uncompressed†	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.3	92.2	81.2	97.3
AUTOCACHE†	9.2×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	98.8	92.2	75.0	96.9
	16.1×	100.0	100.0	100.0	98.4	100.0	99.2	100.0	99.7	98.6	84.4	81.2	96.5
	18.5×	100.0	100.0	100.0	100.0	100.0	100.0	99.6	99.7	98.4	81.2	81.2	96.4
	30.9×	100.0	100.0	100.0	100.0	100.0	99.6	100.0	99.7	98.6	70.3	81.2	95.4
	38.8×	100.0	100.0	93.8	100.0	96.9	97.7	97.7	100.0	97.8	84.4	76.6	95.0
	39.9×	100.0	100.0	98.4	100.0	100.0	98.4	99.2	100.0	98.8	71.9	76.6	94.8
	57.4×	100.0	100.0	95.3	100.0	100.0	99.6	98.8	99.1	98.6	76.6	75.0	94.8
	72.9×	100.0	100.0	93.8	100.0	100.0	100.0	99.6	99.4	98.6	78.1	67.2	94.2
	73.9×	100.0	100.0	92.2	100.0	98.4	100.0	99.2	98.8	98.8	75.0	71.9	94.0
	78.4×	100.0	98.4	93.8	98.4	90.6	99.2	97.3	100.0	98.0	79.7	75.0	93.7
	84.3×	100.0	100.0	92.2	96.9	98.4	99.6	99.6	94.7	97.8	78.1	70.3	93.4
	84.4×	100.0	100.0	90.6	96.9	95.3	99.2	98.4	92.5	98.1	73.4	76.6	92.8
	86.6×	100.0	100.0	89.1	98.4	98.4	95.3	95.3	95.6	97.8	76.6	73.4	92.7
	88.9×	100.0	100.0	93.8	98.4	92.2	97.3	98.8	92.8	98.0	78.1	70.3	92.7
	91.3×	100.0	100.0	87.5	98.4	92.2	97.3	97.7	93.8	98.3	78.1	71.9	92.3
KIVI 4b	3.4×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.7	96.1	89.3	85.7	97.2
KIVI 3b	4.4×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	96.2	98.4	92.0	83.0	97.2
KIVI 2b	**6.1**×	100.0	100.0	100.0	100.0	100.0	100.0	99.8	93.0	98.5	87.5	79.5	96.2
KIVI 1b	9.8×	57.1	34.8	0.0	14.3	3.6	10.5	12.1	8.4	41.2	1.8	3.6	17.0
TurboQuant 4b	3.8×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	95.4	98.6	88.4	83.0	96.9
TurboQuant 3b	**4.9**×	100.0	100.0	99.1	100.0	99.1	99.6	100.0	92.5	96.9	85.7	79.5	95.7
TurboQuant 2b	7.1×	87.5	78.6	17.0	61.6	8.9	58.9	65.2	37.9	70.6	34.8	37.5	50.8
TurboQuant 1b	12.7×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.4	0.0	0.0	0.4
KVTC 8×	**8.1**×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	94.5	98.6	91.1	81.2	96.9
KVTC 16×	16.4×	100.0	87.5	72.3	90.2	100.0	90.2	90.8	95.5	98.6	77.7	76.8	89.1
KVTC 32×	33.6×	94.6	50.0	28.6	49.1	100.0	47.1	51.1	34.8	98.1	60.7	58.9	61.2
KVTC 64×	70.9×	24.1	21.4	22.3	18.8	48.2	11.6	15.4	33.7	34.2	22.3	30.4	25.7
xKV 4×	**4.0**×	100.0	99.1	97.3	99.1	100.0	98.7	99.6	74.3	98.7	78.6	79.5	93.2
xKV 8×	8.0×	26.8	23.2	23.2	25.0	99.1	31.7	30.6	34.3	97.9	33.9	60.7	44.2
xKV 16×	15.9×	24.1	22.3	23.2	23.2	36.6	27.0	23.9	41.8	97.2	26.8	49.1	35.9
xKV 32×	32.1×	24.1	22.3	23.2	23.2	25.9	27.0	23.9	43.2	92.4	23.2	50.9	34.5
xKV 64×	63.7×	24.1	22.3	23.2	23.2	25.9	27.0	23.9	43.2	91.9	23.2	52.7	34.6

†First 64 validation examples per task (704 requests); other rows use all 1,232 requests. Bold KV CR identifies the point in Figure 4.

Table 5: RULER-32K: Ling-3.0-tiny. Per-task accuracy (%) at each KV compression ratio.

		Needle retrieval							Other tasks				
Method	KV CR	S1	S2	S3	MK1	MK2	MV	MQ	VT	CWE	QA1	QA2	Avg.
Uncompressed	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.7	80.8	83.0	68.8	93.7
Uncompressed†	1.0×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	95.9	79.2	85.9	64.1	93.2
AUTOCACHE†	14.9×	100.0	100.0	100.0	100.0	100.0	99.6	99.6	95.3	95.9	76.6	64.1	93.7
	17.7×	100.0	98.4	100.0	100.0	96.9	100.0	99.6	96.9	96.7	76.6	60.9	93.3
	25.3×	100.0	100.0	98.4	100.0	100.0	98.4	98.4	97.5	95.6	71.9	51.6	92.0
	35.3×	100.0	98.4	96.9	98.4	100.0	96.9	97.7	95.6	95.5	68.8	60.9	91.7
	71.6×	100.0	100.0	98.4	100.0	96.9	99.2	99.6	92.5	96.6	59.4	53.1	90.5
	72.1×	100.0	98.4	96.9	100.0	98.4	98.8	99.6	94.4	95.5	64.1	48.4	90.4
	73.4×	100.0	100.0	96.9	100.0	93.8	99.2	100.0	88.4	97.8	57.8	50.0	89.4
KIVI 5b	3.1×	100.0	100.0	100.0	100.0	100.0	99.8	100.0	98.0	90.1	83.9	71.4	94.8
KIVI 4b	**3.9**×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	97.3	97.0	81.2	67.9	94.9
KIVI 3b	5.2×	60.7	42.9	26.8	58.0	1.8	41.1	54.0	28.2	33.6	16.1	15.2	34.4
KIVI 2b	7.7×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0
KIVI 1b	15.0×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.5	0.4
TurboQuant 4b	3.9×	100.0	100.0	100.0	100.0	100.0	99.8	100.0	98.9	83.8	80.4	68.8	93.8
TurboQuant 3b	**5.2**×	100.0	99.1	100.0	99.1	100.0	99.6	100.0	98.8	81.3	83.9	65.2	93.4
TurboQuant 2b	7.7×	85.7	77.7	63.4	88.4	47.3	86.6	90.0	68.4	92.4	46.4	43.8	71.8
TurboQuant 1b	15.0×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.8	0.0	0.0	0.2
KVTC 8×	8.0×	100.0	100.0	100.0	100.0	100.0	99.6	100.0	98.2	90.1	83.0	64.3	94.1
KVTC 16×	**16.1**×	100.0	100.0	100.0	100.0	100.0	99.1	99.8	95.0	97.9	76.8	66.1	94.1
KVTC 32×	31.4×	86.6	74.1	86.6	88.4	99.1	73.0	88.4	85.5	98.1	67.9	62.5	82.7
KVTC 64×	63.4×	22.3	25.0	25.9	28.6	83.9	29.9	26.6	15.2	97.3	42.9	48.2	40.5
xKV 4×	**4.0**×	100.0	100.0	100.0	100.0	100.0	97.1	97.5	98.2	97.9	74.1	63.4	93.5
xKV 8×	8.0×	23.2	26.8	33.0	32.1	93.8	32.4	35.7	36.4	97.3	55.4	54.5	47.3
xKV 16×	15.9×	22.3	25.0	25.9	28.6	57.1	30.1	26.6	39.1	92.1	17.9	43.8	37.1
xKV 32×	31.7×	22.3	25.0	25.9	28.6	17.9	30.1	26.6	39.5	65.3	17.9	43.8	31.2
xKV 64×	62.7×	22.3	25.0	25.9	28.6	18.8	29.9	26.6	27.3	77.9	16.1	42.9	31.0

†First 64 validation examples per task (704 requests); other rows use all 1,232 requests. Bold KV CR identifies the point in Figure 4.

Table 6: RULER-32K: Gemma-4-12B. Per-task accuracy (%) at each KV compression ratio.

		Needle retrieval							Other tasks				
Method	KV CR	S1	S2	S3	MK1	MK2	MV	MQ	VT	CWE	QA1	QA2	Avg.
Uncompressed	1.0×	100.0	100.0	100.0	100.0	100.0	99.8	98.4	97.7	98.8	92.0	67.0	95.8
Uncompressed†	1.0×	100.0	100.0	100.0	100.0	100.0	99.6	98.4	98.8	98.6	92.2	68.8	96.0
AUTOCACHE†	11.2×	100.0	100.0	100.0	100.0	100.0	99.2	100.0	99.4	96.1	92.2	67.2	95.8
	20.8×	100.0	100.0	100.0	100.0	100.0	100.0	100.0	95.9	98.0	89.1	62.5	95.0
	30.8×	100.0	100.0	100.0	98.4	100.0	98.8	98.4	98.4	98.4	78.1	60.9	93.8
	33.0×	100.0	100.0	100.0	100.0	100.0	99.2	99.2	97.8	97.8	75.0	54.7	93.1
	69.9×	98.4	100.0	98.4	100.0	100.0	96.9	99.6	98.1	94.2	84.4	50.0	92.7
	71.4×	98.4	100.0	98.4	100.0	100.0	93.0	97.7	98.1	95.8	79.7	51.6	92.1
	74.0×	96.9	100.0	98.4	100.0	98.4	96.9	100.0	93.4	94.8	81.2	50.0	91.8
KIVI 4b	**3.5**×	100.0	99.1	100.0	99.1	100.0	96.0	94.6	86.6	95.4	87.5	67.0	93.2
KIVI 3b	4.5×	94.6	98.2	48.2	92.0	100.0	80.4	61.4	81.8	75.7	77.7	64.3	79.5
KIVI 2b	6.3×	40.2	26.8	0.0	8.9	1.8	8.5	5.8	0.7	0.1	9.8	3.6	9.7
KIVI 1b	10.4×	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TurboQuant 4b	3.9×	100.0	100.0	100.0	100.0	100.0	99.8	99.6	97.7	98.6	91.1	67.0	95.8
TurboQuant 3b	5.2×	100.0	100.0	100.0	100.0	100.0	100.0	99.8	94.8	97.3	88.4	66.1	95.1
TurboQuant 2b	**7.7**×	100.0	100.0	94.6	98.2	99.1	97.3	96.4	94.6	89.4	82.1	66.1	92.5
TurboQuant 1b	15.0×	2.7	0.0	0.0	0.0	0.9	0.4	0.0	0.2	0.0	0.0	0.0	0.4
KVTC 8×	8.0×	100.0	100.0	100.0	100.0	100.0	99.1	99.8	96.1	98.8	91.1	70.5	95.9
KVTC 16×	**16.1**×	99.1	100.0	100.0	100.0	100.0	98.0	99.1	84.8	97.7	87.5	62.5	93.5
KVTC 32×	32.4×	92.9	98.2	54.5	90.2	100.0	88.6	88.2	28.2	94.6	79.5	62.5	79.8
KVTC 64×	64.8×	8.0	3.6	6.2	7.1	65.2	4.2	4.2	5.9	97.6	17.9	36.6	23.3
xKV 2×	**2.0**×	100.0	100.0	96.4	99.1	100.0	99.6	98.4	97.0	97.2	87.5	62.5	94.3
xKV 4×	4.0×	100.0	97.3	96.4	97.3	99.1	96.9	91.5	80.9	98.3	56.2	48.2	87.5
xKV 8×	8.0×	8.0	1.8	1.8	4.5	82.1	2.0	1.6	3.2	98.3	10.7	34.8	22.6
xKV 16×	15.9×	3.6	1.8	0.9	6.2	12.5	4.2	2.7	0.2	82.8	2.7	23.2	12.8
xKV 32×	31.7×	1.8	0.9	0.9	2.7	0.0	3.3	1.3	0.4	7.8	0.9	22.3	3.8
xKV 64×	62.7×	0.0	0.0	0.0	0.9	0.0	1.8	1.8	0.0	2.3	2.7	20.5	2.7

†First 64 validation examples per task (704 requests); other rows use all 1,232 requests. Bold KV CR identifies the point in Figure 4.

C.3 MRCR: PER-TASK ACCURACY

Tables 7–10 report every MRCR curve point. Columns 2, 4, and 8 give accuracy for the corresponding number of needles; Avg. is the benchmark average. Table conventions follow Appendix C.2.

Table 7: MRCR-32K: Qwen3.5-9B. Accuracy (%) by needle count.

Method	KV CR	2	4	8	Avg.
Uncompressed	1.0×	96.4	87.3	53.8	79.1
Uncompressed†	1.0×	96.4	88.8	55.7	80.3
AUTOCACHE†	6.0×	96.4	88.5	55.8	80.2
	8.2×	94.9	84.2	59.8	79.6
	12.3×	93.6	87.3	55.9	79.0
	47.6×	90.7	90.0	51.8	77.5
	47.6×	94.8	82.3	47.4	74.8
KIVI 4b	3.5×	96.4	86.4	51.3	78.0
KIVI 3b	**4.5**×	96.4	85.1	54.9	78.8
KIVI 2b	6.2×	91.4	69.2	51.6	70.7
KIVI 1b	10.2×	0.0	0.0	0.0	0.0
TurboQuant 4b	3.9×	96.4	84.1	54.4	78.3
TurboQuant 3b	**5.1**×	96.4	85.3	53.7	78.4
TurboQuant 2b	7.5×	89.1	60.1	48.8	66.0
TurboQuant 1b	14.2×	0.0	0.0	0.0	0.0
KVTC 8×	8.0×	96.4	83.3	57.4	79.0
KVTC 16×	**16.0**×	95.4	86.2	57.0	79.5
KVTC 32×	32.1×	84.6	59.4	43.0	62.3
KVTC 64×	64.4×	30.3	23.5	15.2	23.0
xKV 4×	**3.9**×	93.0	80.2	57.1	76.8
xKV 8×	8.0×	86.4	59.2	48.1	64.6
xKV 16×	16.0×	25.1	22.4	14.3	20.6
xKV 32×	32.0×	20.7	18.2	12.5	17.1
xKV 64×	64.2×	20.0	17.5	12.7	16.7

Table 8: MRCR-32K: Nemotron-3.5-L-30B. Accuracy (%) by needle count.

Method	KV CR	2	4	8	Avg.
Uncompressed	1.0×	64.5	37.4	23.0	41.6
Uncompressed†	1.0×	62.1	35.6	22.5	40.1
AUTOCACHE†	2.6×	64.8	31.3	25.7	40.6
	7.6×	57.7	38.5	22.6	39.6
	13.1×	59.6	34.0	24.9	39.5
	17.4×	57.2	35.1	26.1	39.5
KIVI 4b	3.4×	61.0	35.6	24.1	40.2
KIVI 3b	**4.4**×	61.3	34.8	26.0	40.7
KIVI 2b	6.1×	46.7	27.5	18.0	30.7
KIVI 1b	9.8×	0.0	0.0	0.0	0.0
TurboQuant 4b	**3.8**×	64.0	32.3	23.8	40.0
TurboQuant 3b	4.9×	42.9	21.2	20.1	28.1
TurboQuant 2b	7.1×	1.2	1.5	1.5	1.4
TurboQuant 1b	12.7×	0.0	0.0	0.0	0.0
KVTC 4×	**3.8**×	61.7	34.1	22.0	39.3
KVTC 8×	8.1×	46.7	28.2	23.6	32.8
KVTC 16×	16.2×	36.5	18.9	15.4	23.6
KVTC 32×	32.9×	22.5	16.4	15.9	18.3
KVTC 64×	68.1×	20.7	14.9	14.3	16.7
xKV 2×	**2.0**×	61.3	33.9	23.3	39.5
xKV 3×	2.9×	52.2	25.6	22.7	33.5
xKV 4×	3.9×	31.6	17.9	23.6	24.4
xKV 8×	8.0×	21.4	16.9	15.7	18.0
xKV 16×	15.9×	20.9	15.9	17.1	18.0
xKV 32×	31.9×	21.9	15.9	17.1	18.3
xKV 64×	63.5×	21.8	18.2	17.1	19.0

†First 64 validation examples per task (192 requests); other rows use all 243 requests. Bold KV CR identifies the point in Figure 4.

Table 9: MRCR-32K: Ling-3.0-tiny. Accuracy (%) by needle count.

Method	KV CR	2	4	8	Avg.
Uncompressed	1.0×	95.2	82.3	50.1	75.9
Uncompressed†	1.0×	93.9	82.7	50.9	75.8
AUTOCACHE†	9.2×	93.7	76.7	53.2	74.5
	10.9×	94.3	74.9	47.5	72.2
	14.4×	93.7	69.7	51.6	71.7
	17.4×	91.4	72.0	50.4	71.3
	18.3×	91.6	74.8	47.4	71.3
	18.3×	88.7	69.9	53.9	70.8
	19.2×	88.8	75.3	46.6	70.2
KIVI 5b	**2.9**×	94.3	77.9	44.6	72.3
KIVI 4b	3.9×	90.5	69.9	40.4	66.9
KIVI 3b	5.2×	1.1	1.7	0.8	1.2
KIVI 2b	7.7×	0.0	0.0	0.0	0.0
KIVI 1b	15.0×	0.0	0.0	0.0	0.0
TurboQuant 4b	**3.9**×	95.1	71.9	48.4	71.8
TurboQuant 3b	5.2×	90.4	65.1	43.4	66.3
TurboQuant 2b	7.7×	18.9	11.2	10.7	13.6
TurboQuant 1b	15.0×	0.0	0.0	0.0	0.0
KVTC 4×	3.7×	93.9	79.0	44.5	72.5
KVTC 8×	**8.0**×	92.7	76.0	46.0	71.6
KVTC 16×	16.0×	86.5	58.6	36.9	60.7
KVTC 32×	31.7×	59.1	41.4	23.4	41.3
KVTC 64×	63.4×	21.7	15.9	12.4	16.6
xKV 2×	**1.9**×	93.2	80.1	46.1	73.1
xKV 3×	2.9×	90.8	71.6	43.8	68.7
xKV 4×	3.8×	88.4	62.9	42.1	64.5
xKV 8×	8.0×	33.3	20.9	22.7	25.7
xKV 16×	15.9×	19.0	16.3	14.4	16.6
xKV 32×	31.6×	19.6	16.5	14.3	16.8
xKV 64×	62.6×	20.9	17.7	14.6	17.7

Table 10: MRCR-32K: Gemma-4-12B. Accuracy (%) by needle count.

Method	KV CR	2	4	8	Avg.
Uncompressed	1.0×	62.2	27.8	15.3	35.1
Uncompressed†	1.0×	63.9	28.1	14.4	35.5
AUTOCACHE†	12.5×	61.5	28.3	17.6	35.8
	15.8×	64.1	27.5	12.6	34.7
	19.6×	55.9	30.5	15.7	34.1
	20.1×	57.6	29.7	14.1	33.8
	45.5×	58.7	24.9	17.7	33.8
KIVI 5b	**2.8**×	61.0	26.8	17.3	35.0
KIVI 4b	3.5×	57.2	27.4	13.8	32.8
KIVI 3b	4.5×	24.1	13.2	13.6	17.0
KIVI 2b	6.3×	0.0	0.0	0.0	0.0
KIVI 1b	10.4×	0.0	0.0	0.0	0.0
TurboQuant 4b	3.9×	63.6	26.1	13.1	34.3
TurboQuant 3b	**5.2**×	63.3	27.5	19.3	36.7
TurboQuant 2b	7.7×	56.7	27.1	14.8	32.9
TurboQuant 1b	15.0×	0.0	0.0	0.0	0.0
KVTC 8×	8.0×	61.1	21.8	13.5	32.1
KVTC 16×	**16.1**×	59.2	23.4	21.6	34.7
KVTC 32×	32.0×	40.1	21.7	17.1	26.3
KVTC 64×	65.0×	17.8	14.3	15.4	15.8
xKV 2×	**2.0**×	61.1	24.4	18.7	34.8
xKV 4×	3.9×	45.5	20.7	12.5	26.2
xKV 8×	8.0×	14.2	9.6	11.2	11.7
xKV 16×	15.9×	12.1	9.6	8.5	10.1
xKV 32×	31.8×	12.3	10.5	7.5	10.1
xKV 64×	63.6×	10.8	7.6	6.5	8.3

†First 64 validation examples per task (192 requests); other rows use all 243 requests. Bold KV CR identifies the point in Figure 4.