MEMENTO: Teaching LLMs to Manage Their Own Context

Vasilis Kontonis Yuchen Zeng Shivam Garg Lingjiao Chen Hao Tang Ziyan Wang
Ahmed Awadallah Eric Horvitz John Langford Dimitris Papailiopoulos
Microsoft Research

Reasoning models think in long, unstructured streams with no mechanism for compressing or organizing their own intermediate state. We introduce MEMENTO: a method that teaches models to segment reasoning into blocks, compress each block into a *memento*, i.e., a dense state summary, and reason forward by attending only to mementos, reducing context, KV cache, and compute. To train MEMENTO models, we release OPENMEMENTOS, a public dataset of 228K reasoning traces derived from OpenThoughts-v3, segmented and annotated with intermediate summaries. We show that a two-stage SFT recipe on OPENMEMENTOS is effective across different model families (Qwen3, Phi-4, Olmo 3) and scales (8B–32B parameters). Trained models maintain strong accuracy on math, science, and coding benchmarks while achieving ∼2.5× peak KV cache reduction. We extend vLLM to support our inference method, achieving up to 2× throughput improvement while also enabling us to perform RL and further improve accuracy. Finally, we identify a *dual information stream*: information from each reasoning block is carried both by the memento text and by the corresponding KV states, which retain implicit information from the original block. Removing this channel drops accuracy by 15 pp on AIME24.

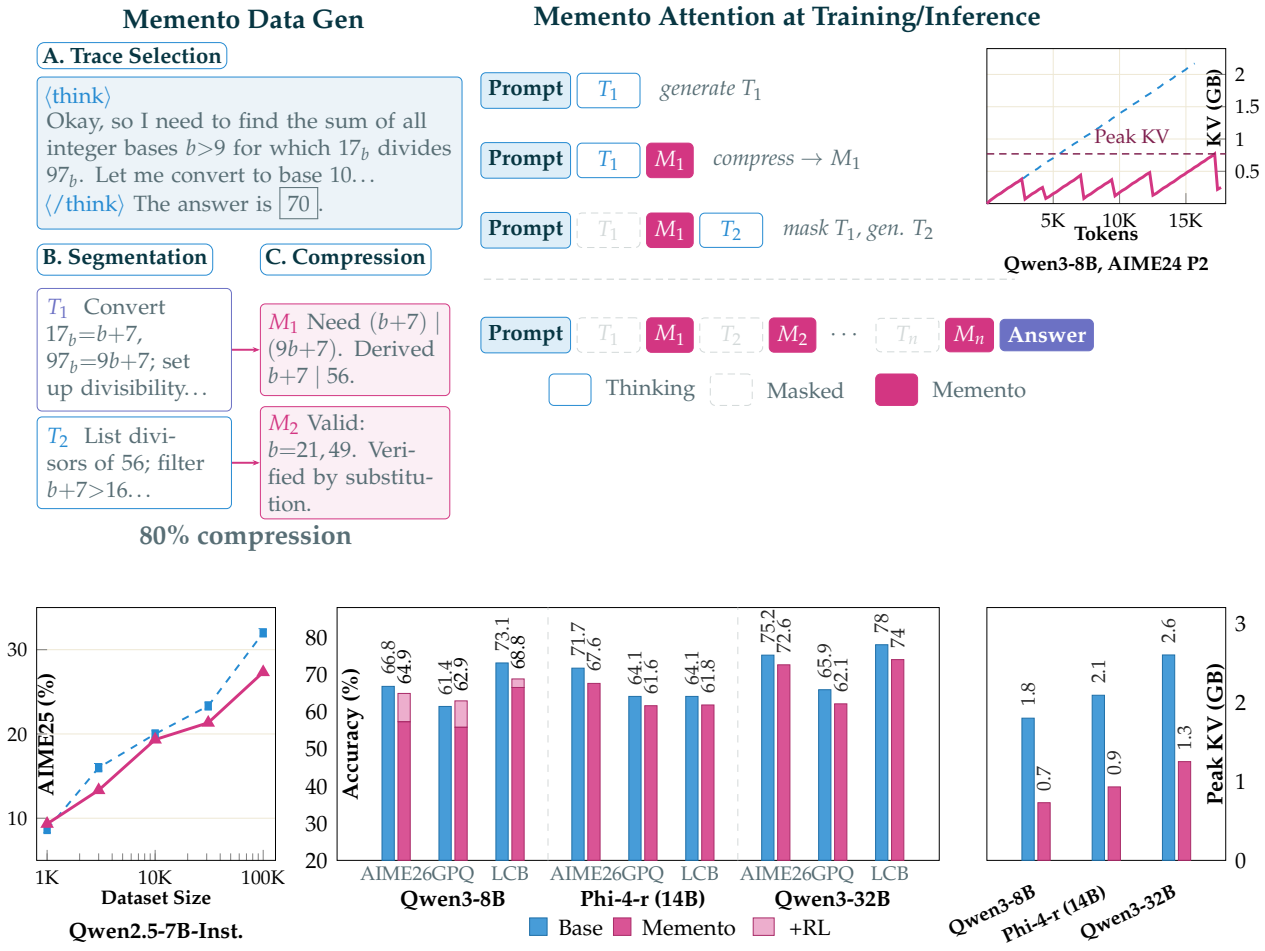

Figure 1: **MEMENTO overview. Top left:** SFT data generation pipeline. Starting from a reasoning trace, we split text into sentences, use an LLM to score each sentence boundary as a potential stopping point, optimize boundary selection algorithmically, and finally use an LLM to summarize each block into a memento. **Top right:** Sparse attention during inference. The model produces alternating thinking blocks (T_i) and mementos (M_i); once a memento is generated, KV cache entries for its preceding thinking block are physically removed. The sawtooth KV trace shows the resulting memory pattern. **Bottom left:** Data scaling on Qwen2.5-7B-Instruct (AIME25); MEMENTO scales similarly to vanilla SFT on OpenThoughts (Guha et al., 2025) across dataset sizes (Figure 4). **Bottom center:** Accuracy across three models on AIME26, GPQA-D, and LCB; lighter bars show RL gains for Qwen3-8B (Section 5). **Bottom right:** Peak KV cache (GB), averaged across all benchmark categories, showing ∼2–2.5× reduction.

1 Introduction

Large language models routinely reason at test time, spending thousands of tokens working through a problem before arriving at an answer (OpenAI, 2024; DeepSeek-AI et al., 2025; Qwen Team, 2024). This has led to dramatic gains on hard reasoning benchmarks, but has also created a new problem: *reasoning models have no built-in mechanism to organize their chain-of-thought*. A 32K-token CoT is a flat, unstructured stream, and there is no mechanism for the model to mark an intermediate result as worth keeping, or to compress a long derivation into a compact conclusion that it can reference later. Every past token sits in the attention window at equal cost, and the model has learned no way to drop it.

We propose MEMENTO, an approach that trains models to segment their chain of thought into semantically coherent blocks and, after each block, generate a compressed summary that we call a *memento*[1]. Rather than a summary in the usual expository sense, each memento is a minimal record of a reasoning block, preserving its conclusions, intermediate values, and key directional decisions in as few tokens as possible. Once a memento is produced, the preceding thinking block is masked within a single, uninterrupted generation call via a custom vLLM based engine (Kwon et al., 2023): at every subsequent step, the model attends only to past mementos and the current block. Each block is compressed to a $\sim$5–20$\times$ smaller size on average, so the effective context the model attends to is a fraction of the full trace. The inference procedure is illustrated in Figure 1 (right).

Crucially, because masking happens in-place rather than by restarting generation, the KV cache entries of each memento are computed while the full block is still in context and retained after the block is masked. Despite the fact that the original thinking tokens are gone, they remain implicitly present in the representations of the memento KV states This creates a *dual information stream*: the explicit memento text plus an implicit representational channel through the cached KV states. We verify this experimentally: recomputing memento KVs without block context reduces accuracy by 15 pp on AIME'24 (Section 6.2.1), and our probing experiments (Section 6.2.2) demonstrate that block information not present in the memento text is still recoverable from the memento KV states, with upper layers carrying the most task-relevant signal.

A prime concern is that compression could destroy reasoning capacity. Our experiments across three model families, i.e., Qwen3 (8B/32B), Phi-4-reasoning (14B), and Olmo-3-7B-Think, show that this is not the case. On AIME'26, Qwen3-32B with MEMENTO loses just 2.6 pp (72.6% vs. 75.2%) while cutting peak KV cache by $\sim$2$\times$. Averaged across the five benchmark groups in Table 1, the accuracy gap is 3.5 pp at 32B and 6.3 pp at 8B, and we observe that the gap shrinks with scale within the same model family, suggesting that larger models manage compressed context more effectively. Further, we show that RL fine-tuning can close the remaining gap, enabled by native block masking support in our vLLM fork.

To train MEMENTO models, we construct OPENMEMENTOS, a public dataset of 228K segmented and summarized reasoning traces derived from OpenThoughts (Guha et al., 2025). Building this dataset required solving a non-trivial annotation problem: reasoning traces lack natural segment boundaries, and naive summarization loses the precise intermediate state a model needs to continue. Our pipeline combines LLM-scored boundary detection, algorithmic segmentation, and iterative judge-refined summarization to produce training data where each memento is both faithful and minimal. Our key contributions can be summarized as follows:

1. **OPENMEMENTOS**: a 228K-trace public dataset of segmented, summarized reasoning chains, with the annotation pipeline and code.
2. **Demonstration at scale** across three model families (Qwen3 8B/32B, Phi-4-reasoning 14B, Olmo-3-7B-Think), showing that models internalize summarization as a learned capability while preserving reasoning accuracy at 2–3$\times$ peak KV cache reduction.
3. **Native block masking in vLLM**, a custom fork that supports in-place KV cache masking within a single generation call: a key infrastructure bottleneck for both inference and training with MEMENTO. This

[1]Named after the 2000 film by Christopher Nolan, in which the protagonist compensates for anterograde amnesia by maintaining external memory artifacts—an analogy for a model that must reason from compressed summaries of its own past thinking.

enables **RL fine-tuning with block masking**, which allows us to close the accuracy gap.

4. **The dual information stream**: we identify and verify that memento KV states encode information from masked blocks which is a mechanism absent in restart-based approaches. Removing this channel drops accuracy by 15 pp on AIME'24.

2 Related Work

Context management for long-running models is typically handled through external infrastructure—separate summarizers, memory modules, or orchestration logic (Wu et al., 2025; Xu et al., 2025). We instead focus on teaching models to manage their own context during reasoning, as an internal capability rather than an external system. Among works that do train models for context management, MemAgent (Yu et al., 2025) learns to compress long *input* documents into a fixed-size memory via RL, and MEM1 (Zhou et al., 2025) takes a similar approach for multi-turn agent interactions, maintaining a compact internal state across tool calls and environment observations. Both primarily focus on managing external information—retrieved documents, tool outputs, and environment observations—rather than complex reasoning chains typically observed in models solving hard math or coding problems.

The most closely related works to ours train models to compress their own reasoning output in domains such as math: InftyThink (Yan et al., 2025), InftyThink+ (Yan et al., 2026), and Accordion-Thinking (Yang et al., 2026). All three train models to segment reasoning into chunks and produce summaries that subsequent reasoning attends to in place of the original chunks. InftyThink relies on SFT alone, while InftyThink+ and Accordion-Thinking additionally apply RL to optimize summarization quality. All three operate at the text level: after each reasoning chunk, a summary is produced and the context is rebuilt from the accumulated summaries, discarding the original reasoning tokens and their KV cache representations. InftyThink and InftyThink+ do this via separate generation calls (explicit restart), while Accordion-Thinking implements mid-generation context rebuilding within a single inference pass. MEMENTO differs in that it retains summary KV entries via in-engine attention masking rather than text-level context rebuilding, creating a dual stream of information: the explicit memento text, and the implicit representations encoded in the memento's KV cache. Our experiments demonstrate that useful information is stored in these KV entries—dropping them and recomputing memento KVs without block context reduces AIME24 accuracy by 15 percentage points (Section 6.2.1). Beyond the KV-flow distinction, MEMENTO differs in scale (three model families at 8B–32B vs. $\leq$7B), infrastructure (native block masking in vLLM enabling efficient RL rollouts), and public data (228K traces).

Another related work (PENCIL (Yang et al., 2025)) explores learned context management for models trained from scratch on synthetic tasks. PENCIL teaches models to erase intermediate reasoning via reduction rules, enabling small models to solve hard 3-SAT problems and Einstein's Puzzle (a multi-constraint logic deduction) with bounded context. PENCIL demonstrates that the potential of context management extends beyond memory and throughput efficiency—it can enable models to solve significantly harder problems than standard CoT permits. Whether similar significant gains can be achieved in realistic settings such as math and coding problems, for instance through MEMENTO-style compression, is an interesting direction for future work.

A closely related line of work compresses reasoning chunks into learned *gist tokens*—special-purpose tokens whose KV cache entries encode a compressed representation of the preceding chunk, after which the original tokens are evicted (Zhang et al., 2025a; Monea et al., 2025). A limitation of gist-token approaches is that interpretability is lost: the compressed state is encoded entirely in hidden representations. MEMENTO's summaries are natural-language text, preserving interpretability while still achieving compression.

A complementary line of work aims to reduce the memory footprint of reasoning through shorter traces or direct KV cache compression. These include methods that train models to skip low-importance tokens (Xia et al., 2025; Li et al., 2025), train on compressed reasoning traces (Kang et al., 2025; Zhang et al., 2025b), or steer the model to produce shorter traces via RL (Hou et al., 2025; Shrivastava et al., 2025). Other works replace explicit reasoning tokens with latent representations (Shen et al., 2025; Hao et al., 2024). At the KV cache level, inference-time methods such as ThinKV (Ramachandran et al., 2025), R-KV (Cai

et al., 2025), and Reasoning Path Compression (Song et al., 2025) prune or quantize cache entries based on attention patterns, while architectures such as sliding-window attention (Beltagy et al., 2020) limit the attention span by design (Team Olmo et al., 2025). These approaches are orthogonal to MEMENTO and many can likely compose with it: for example, we show that Olmo-3-7B-Think that has sliding-window attention can be effectively combined with MEMENTO.

3 OPENMEMENTOS Dataset

Training models to simultaneously reason and manage context requires high-quality annotated data: reasoning traces segmented into semantically coherent blocks, paired with dense summaries. A core challenge is that typical reasoning traces are not a sequence of independent thoughts; they are a continuous stream without "natural" boundaries. Here we describe our data generation pipeline (Figure 1, top left), which takes raw CoT traces and produces structured traces annotated with mementos.

Design rationale. Each stage in our pipeline reflects a deliberate design choice. Early on, we tried having a frontier LLM directly segment CoTs into semantically coherent blocks. This failed. Even strong models struggle with a combinatorial optimization problem that considers all possible partitions, requiring simultaneous reasoning about block coherence, size balance, and semantic boundaries.

To simplify, we factored the problem: boundary *scoring* asks a local question ("is this a good place to slice the CoT?"), which LLMs handle well, while the global optimization of boundary selection given the LLM scores is handled algorithmically. We then use an LLM judge to grade the quality of summarization. Defining "good summary" programmatically is difficult, but LLMs can give reasonable scores against explicit rubrics. We then chose iterative refinement of mementos over single-shot summarization as initial mementos often miss key formulas or intermediate values; a zero-shot approach achieves only 28% pass rate (scoring $\geq 8/10$ on our rubric), while the judge-feedback loop brings this to 92%.

Stage 0: Seed selection. We source 228K reasoning traces from OpenThoughts-v3 (Guha et al., 2025), a widely-adopted dataset of CoT traces generated by QwQ-32B, and process them through our annotation pipeline to produce the final OPENMEMENTOS dataset. While we could regenerate traces using stronger teachers, we leverage OpenThoughts because: (1) substantial effort has already been invested in generating these traces at scale; (2) the OpenThinker-3 paper provides extensive baselines, making it an ideal testbed; and (3) our hypothesis that traces from a relatively strong reasoner (QwQ-32B) should transfer across model families is confirmed empirically (works for Qwen3, Phi-4, Olmo 3).

Stage 1: Sentence splitting. We partition reasoning traces into atomic "sentences": complete, modular thoughts that can stand alone. Code blocks and multi-line math are detected and protected as atomic units. Plain text is split at sentence boundaries (avoiding splits inside parentheses, inline math, or abbreviations). Finally, we merge logically connected fragments: sentences ending with colons are attached to the next; continuation words (Therefore, Thus, So) signal a need to merge with the preceding sentence; short fragments ($<$5 tokens) and consecutive math expressions are consolidated. This structure-aware splitting reduces candidate boundaries by $\sim 2\times$ vs. naive sentence splitting (397 $\rightarrow$ 187 per trace on average).

Stage 2: Boundary scoring. An LLM judge (GPT-5.x in our case) evaluates each inter-sentence boundary as a potential breakpoint, scoring from 0 (mid-thought, would disrupt flow) to 3 (major transition, natural chapter boundary). The prompt instructs: "Never score 2–3 mid-calculation. Score 0 if previous sentence ends with ':' or '='." Because traces contain hundreds of boundaries, we score them in batches: the judge sees a window of consecutive sentences and scores each boundary within that window. See Figure 2 for an example of boundary scoring.

Boundary scoring example — complex-plane geometry trace

[177] *"Hence, this approach again leads to the only solution being zero."*	0.0
[178] *"Therefore, unless I made a mistake . . . the answer is the trivial solution."*	1.5
[179] *"Perhaps the problem's answer is indeed zero, so the distance is zero."*	**2.0**
[180] *"Alternatively, maybe I made an error in interpretation."*	**3.0**
[181] *"Let me check with an example."*	0.5
. . .	
[185] *"Suppose I take* r*=2. Then* u*=5,* $v = \pm\sqrt{7}$ *. . . "*	0.0
[186] *"But* $\sqrt{56} \approx 7.48$*,* $8\sqrt{2} \approx 11.31$*, which are not equal."*	0.0

Figure 2: Boundary scoring assigns each inter-sentence boundary a score from 0 to 3. Sentence 179 wraps up a conclusion (score 2.0); sentence 180 pivots to a new strategy (score 3.0—the strongest possible boundary); sentences 185–186 are mid-derivation (scores 0.0—never split here). The segmentation optimizer selects cuts at high-scoring transitions.

Stage 3: Segmentation. Given n sentences with boundary scores $s_1, \ldots, s_{n-1}$, we partition the trace into K contiguous blocks by maximizing $\frac{1}{K}\sum_{b \in \text{boundaries}} s_b \; - \; \lambda \cdot \sigma(\ell_1, \ldots, \ell_K) \,/\, \mu(\ell_1, \ldots, \ell_K)$, subject to every block containing at least 200 tokens. Here, b is the boundary positions, ℓ_k is the token count of the k-th block, μ and σ are the mean and standard deviation of the block sizes, and $\lambda = 0.5$. We optimize over valid partitions and values of K. We observe that the first term rewards cutting at strong semantic boundaries: a partition that places cuts at score-3 transitions (major topic changes) scores higher than one cutting low-score transitions (mid-derivation). The second term penalizes uneven block sizes. The coefficient of variation σ/μ is scale-invariant, so the penalty applies equally whether the trace is 5K or 50K tokens. Without this term, the optimizer would greedily cut at the top-scoring boundaries regardless of balance, often producing one very long block and several tiny ones.

Stage 4: Iterative memento generation. Each block is compressed into a memento: a terse state representation that preserves all logically relevant information (definitions, formulas, intermediate values, chosen strategies, rejected approaches) needed for subsequent blocks to succeed. Unlike traditional summarization, the goal is *"lossless compression" of reasoning state*: mementos must capture everything a future reasoning step might need, targeting ∼15–25% of original tokens while being purely extractive (no new derivations or error corrections).

Compressor. The compressor call (using GPT-5.x) receives all blocks and produces one memento per block using terse notation (semicolon-separated clauses, "name: value" pairs, compact math). The prompt instructs: "You are a STATE-COMPRESSOR. Minimize tokens subject to fully capturing all logically relevant information."

Judge. A separate LLM call (again using GPT-5.x) evaluates each memento on a 0–10 scale across six dimensions: (1) formulas extracted verbatim (0–3), (2) numerical values preserved (0–2), (3) methods explicitly named (0–2), (4) validation included (0–1), (5) no hallucinations (0–1), and (6) result-first structure (0–1). If the score falls below the acceptance threshold $\tau = 8$ (out of 10), the judge provides actionable feedback (e.g., "Missing formula: $K^2 - 3K + 3$") used to refine the memento; mementos scoring $\geq \tau$ are accepted without refinement. We use max $T = 2$ iterations, i.e., Compressor → Judge → Compressor → Judge. Adding more iterations does not significantly improve memento quality and results in longer mementos. See Figure 11 for an example of the iterative improvement of mementos.

Iterative refinement is essential. Single-pass memento generation achieves only 28% pass rate ($\geq 8/10$ judge score). Two iterations of judge feedback bring this to 92%. Initial mementos often miss critical formulas or intermediate values that downstream blocks need to reason correctly.

Dataset statistics. Figure 3 characterizes the final OPENMEMENTOS dataset (228K samples: 54% math, 19% code, 27% science). Math and code traces produce more blocks per sample (median 9) than science (median 7), and math has the largest blocks (median 3.8K chars). Summary sizes are remarkably stable across domains (median 509–603 chars), yielding median compression ratios of 0.16 (math), 0.18 (code), and 0.23 (science)—corresponding to ∼4–6× block-level compression. Across the full dataset, the average block contains ∼1,150 tokens and the average memento ∼194 tokens, for a trace-level compression of ∼6× (from ∼10,900 block tokens to ∼1,850 memento tokens per trace).

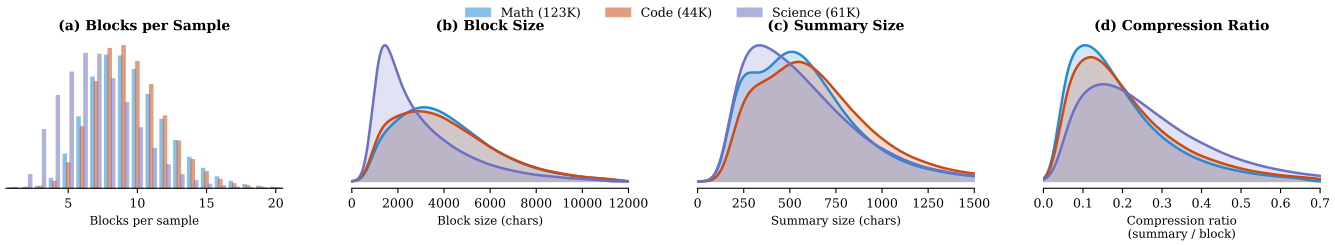

Figure 3: **OPENMEMENTOS dataset distributions by domain** (228K samples). **(a)** Math and code have ∼9 blocks/sample; science has ∼7. **(b)** Block sizes range from 2.3K (science) to 3.8K (math) chars. **(c)** Summary sizes cluster around 509–603 chars across all domains, indicating a stable compression target. **(d)** Math achieves the tightest compression ratio (median 0.16) due to its larger blocks.

4 Training the MEMENTO Models

We use a two-stage SFT procedure on OPENMEMENTOS that separates format learning from context management. The intuition follows standard curriculum learning: we first let the model acquire the block-memento format under normal conditions, then introduce the harder constraint of operating without access to masked content, see Appendix A.4.1 for an ablation.

Stage 1: Full Attention: Standard causal attention over all tokens. Loss is computed on all tokens, including thinking blocks, mementos, special tokens, and the final answer. The model learns the block-memento format without any context management pressure.

Stage 2: Memento Attention: After each completed memento, *the preceding thinking block is masked from all subsequent attention*. This teaches the model to produce self-contained mementos that carry all information needed for downstream reasoning.

The attention mask implementation maintains a **block cache** that tracks whether each token belongs to a thinking block, summary, or other content. When `<|summary_end|>` is generated, the preceding block is marked as completed and masked from future attention. For training, this mask is constructed upfront as a dense matrix; for inference, the block cache is stateful across autoregressive steps. Four special tokens (`<|block_start|>`, `<|block_end|>`, `<|summary_start|>`, `<|summary_end|>`) are added and initialized as the mean embedding of semantically related existing tokens (e.g., `<|block_start|>` from *block, start, begin, section, step*) plus small Gaussian noise.

Data Scaling. When training "from scratch" a non-reasoning model (Qwen2.5-7B-Instruct), data scaling follows a similar monotonic trend as standard reasoning SFT (Guha et al., 2025). We study how performance scales with the amount of OPENMEMENTOS training data by fine-tuning Qwen2.5-7B-Instruct on varying amounts of data (1K, 3K, 10K, 31K, 100K examples), comparing vanilla OpenThoughts (OT), OPENMEMENTOS with full attention (OM/Full), and OPENMEMENTOS with memento attention (OM/Mem). As shown in Figure 4, all three methods improve monotonically from 1K to 100K. OT achieves the highest accuracy across all data budgets, while OM/Full and OM/Mem trail by a modest margin.

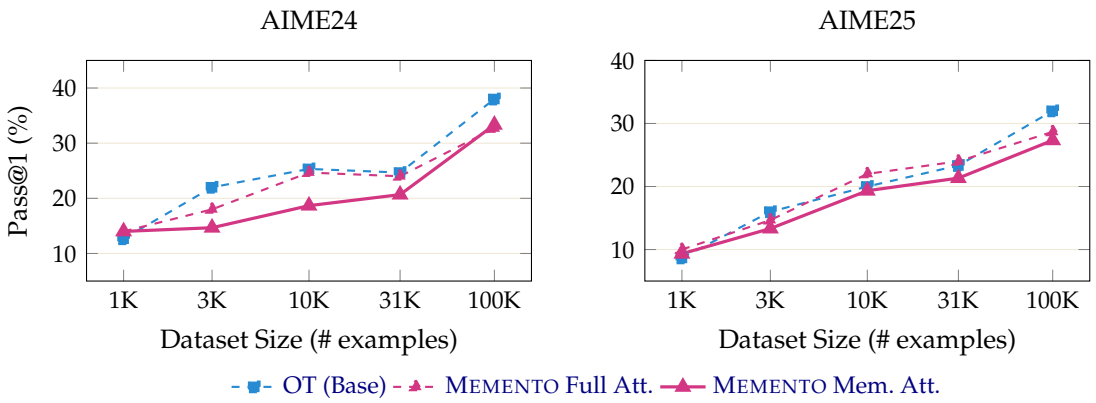

Figure 4: **Training data scaling.** Pass@1 accuracy on AIME24 and AIME25 for Qwen2.5-7B-Instruct fine-tuned on 1K–100K examples. All methods improve monotonically with data size.

Fine-Tuning Reasoning Models. When starting with already strong reasoning models, we found that training for more epochs on fewer samples is more effective than training on more samples with fewer epochs. We train on 31K samples from the 228K OPENMEMENTOS pool with 32K sequence length, as further gains are more effectively attainable through reinforcement learning (Section 5) rather than additional supervised data. We release the full 228K dataset to support future research in both directions.

Hyperparameters. We use the same hyperparameters for all models and stages. Key hyperparameters: learning rate 8×10^{-5}, cosine schedule with 5% warmup, 5 epochs per stage, AdamW (β_1=0.9, β_2=0.999), no weight decay, bfloat16 precision, batch size 512, and 32 B200 GPUs. See Appendix A.2.1 for details.

Results and Evaluation. We evaluate MEMENTO across four model families and scales: Qwen3-8B, Phi-4-reasoning (14B), Qwen3-32B, and Olmo-3-7B. Example MEMENTO traces from Qwen3-32B are provided in Appendix A.5. All results report pass@1 accuracy. We evaluate on 14 benchmarks spanning competition mathematics (11 contests sourced from MathArena, grouped as "Comp. Math" in Table 1), standard math (MATH-500), science (GPQA Diamond), and code (LiveCodeBench v6); Table 1 summarizes accuracy alongside KV cache footprints.

Control Runs. To decouple the effect of performing SFT on already strong reasoning models (that leads to some performance loss) we do control (shown in Gray in Table 1) runs where we train the base models on the original unmodified OpenThoughts subsets. As one may expect the highest performance loss, both for MEMENTO and the control runs, happens for the most challenging Competition math benchmarks while for easier benchmarks such as MATH-500 MEMENTO is able to match baselines almost perfectly.

Scale Helps. Within the Qwen3 family the accuracy gap shrinks with scale, from -6.3 pp at 8B to -3.5 pp at 32B (averaged across the five benchmark groups in Table 1). This suggests that larger models manage compressed context more effectively and that further gains may be achievable at greater scale.

Peak KV cache and AUC savings. We observe that peak KV is reduced by 2–3$\times$ and KV AUC (area under the KV-cache-size curve over generation steps) capturing total memory-time cost, by 2–3.5$\times$ on competition math, with even larger reductions on benchmarks where the base model generates long responses. Figure 5 illustrates the range of per-problem KV cache behaviors produced by block masking.

Memento on Olmo-3-7B-Think. We applied our OPENMEMENTOS dataset and training recipe to Olmo-3-7B-Think, which uses a hybrid attention architecture: 24 of its 32 layers employ sliding-window attention

Table 1: MEMENTO achieves 2–3× peak KV reduction on models with uniform attention layers while maintaining strong reasoning performance; RL on top of our Qwen3-8B further improves accuracy. Olmo-3-7B shows more modest savings (∼0.85–0.93×) due to its hybrid sliding-window architecture (Section 4). The Δ columns show changes of MEMENTO and Mem.+RL relative to Control: accuracy deltas are additive (pp); KV deltas are multiplicative (Method / Control, so 0.39× means 61% KV reduction). *Metrics.* Accuracy (%): pass@1 accuracy. Peak KV (GB): peak KV cache size; determines the minimum memory required to serve a request. AUC KV (GB·ktok): area under the KV-occupancy-vs-token curve, capturing total memory-time cost that penalizes both large footprints and long generations (see Figure 5 for an illustration). *Rows.* For each model we report Base (unmodified), Control (Base fine-tuned on the same OpenThoughts source traces used to create OPENMEMENTOS, but without block/memento annotations, for the same number of training examples), and MEMENTO (SFT on OPENMEMENTOS with block masking); for Qwen3-8B we additionally report MEMENTO+RL (Appendix A.2.3). Olmo-3 MEMENTO competition-math evaluations use 8 generations per problem (vs. 64 for all other models) and are run with HuggingFace Transformers rather than vLLM. See Appendix A.2.2 for full benchmark and evaluation details.

			AIME'26		Comp. Math		MATH-500		GPQA-D		LCB v6	
			Val	Δ	Val	Δ	Val	Δ	Val	Δ	Val	Δ
Qwen3-8B	Base	Acc	$66.8_{1.1}$		$54.3_{0.3}$		$90.5_{0.9}$		$61.4_{2.4}$		$73.1_{1.0}$	
		Peak KV	2.41		2.71		0.84		1.23		1.76	
		AUC KV	25.3		30.9		4.3		6.6		15.6	
	Control	Acc	$64.7_{1.1}$		$49.2_{0.3}$		$89.7_{1.0}$		$57.8_{2.5}$		$70.0_{1.0}$	
		Peak KV	2.59		2.82		0.88		1.60		1.89	
		AUC KV	28.3		33.1		4.7		11.8		19.2	
	MEMENTO	Acc	$57.3_{1.1}$	−7.4%	$45.1_{0.3}$	−4.1%	$90.1_{0.9}$	+0.4%	$55.8_{2.5}$	−2.0%	$66.5_{1.0}$	−3.5%
		Peak KV	1.02	0.39×	1.08	0.38×	0.41	0.47×	0.56	0.35×	0.60	0.32×
		AUC KV	9.7	0.34×	10.7	0.32×	1.9	0.40×	4.0	0.34×	5.6	0.29×
	Mem. + RL	Acc	$64.9_{1.1}$	+0.2%	$49.4_{0.3}$	+0.2%	$91.0_{0.9}$	+1.3%	$62.9_{2.4}$	+5.1%	$68.8_{1.0}$	−1.2%
		Peak KV	1.45	0.56×	1.48	0.52×	0.68	0.77×	1.24	0.77×	1.12	0.59×
		AUC KV	14.9	0.53×	16.4	0.50×	3.2	0.68×	9.2	0.78×	10.3	0.54×
Phi-4-r (14B)	Base	Acc	$71.7_{1.0}$		$55.1_{0.3}$		$87.3_{1.1}$		$64.1_{2.4}$		$64.1_{1.0}$	
		Peak KV	2.65		3.06		1.43		0.80		2.45	
		AUC KV	28.8		35.9		17.8		3.7		29.2	
	Control	Acc	$69.8_{1.0}$		$51.4_{0.3}$		$90.6_{0.9}$		$64.1_{2.4}$		$65.0_{1.0}$	
		Peak KV	3.04		3.48		1.04		2.11		2.64	
		AUC KV	28.6		36.7		4.9		14.2		26.8	
	MEMENTO	Acc	$67.6_{1.1}$	−2.2%	$48.7_{0.3}$	−2.7%	$89.7_{1.0}$	−0.9%	$61.6_{2.4}$	−2.5%	$61.8_{1.1}$	−3.2%
		Peak KV	1.17	0.38×	1.25	0.36×	0.51	0.49×	0.80	0.38×	0.92	0.35×
		AUC KV	11.3	0.40×	13.1	0.36×	2.6	0.53×	6.2	0.44×	9.5	0.35×
Qwen3-32B	Base	Acc	$75.2_{1.0}$		$62.7_{0.3}$		$91.9_{0.9}$		$65.9_{2.4}$		$78.0_{0.9}$	
		Peak KV	3.24		3.67		1.26		1.89		2.88	
		AUC KV	26.7		34.7		5.5		9.7		22.9	
	Control	Acc	$74.1_{1.0}$		$58.5_{0.3}$		$91.8_{0.9}$		$64.6_{2.4}$		$75.3_{0.9}$	
		Peak KV	3.83		4.51		1.36		2.45		3.05	
		AUC KV	35.2		48.5		6.2		15.7		27.7	
	MEMENTO	Acc	$72.6_{1.0}$	−1.5%	$56.2_{0.3}$	−2.3%	$91.1_{0.9}$	−0.7%	$62.1_{2.4}$	−2.5%	$74.0_{1.0}$	−1.3%
		Peak KV	1.67	0.44×	1.74	0.39×	0.64	0.47×	1.07	0.44×	1.12	0.37×
		AUC KV	14.0	0.40×	15.7	0.32×	2.8	0.45×	7.6	0.48×	9.3	0.34×
Olmo 3 (7B)	Base	Acc	$67.9_{1.1}$		$52.7_{0.3}$		$91.3_{0.9}$		$50.8_{2.5}$		$64.5_{1.0}$	
		Peak KV	3.95		4.21		2.11		3.21		3.33	
		AUC KV	50.8		60.1		10.7		30.8		40.0	
	Control	Acc	$59.8_{1.1}$		$48.3_{0.3}$		$90.4_{0.9}$		$45.7_{2.5}$		$58.8_{1.1}$	
		Peak KV	3.51		3.78		2.00		2.94		3.22	
		AUC KV	37.1		46.0		9.1		23.8		37.6	
	MEMENTO	Acc	$55.4_{3.2}$	−4.4%	$48.1_{0.9}$	−0.2%	$91.1_{0.9}$	+0.7%	$49.5_{2.5}$	+3.8%	$56.0_{1.1}$	−2.8%
		Peak KV	3.21	0.91×	3.43	0.91×	1.70	0.85×	2.72	0.93×	2.21	0.69×
		AUC KV	37.8	1.02×	43.6	0.95×	8.5	0.93×	25.2	1.06×	20.6	0.55×

(window size 4096), while only 8 layers use full causal attention. Additionally, Olmo 3 uses multi-head attention (MHA, 32 KV heads) rather than grouped query attention (GQA, 8 KV heads in Qwen3). MEMENTO transferred with no architecture-specific modifications: accuracy is well preserved, with Comp. Math dropping only 0.2 pp relative to Control and MATH-500 *improving* by 0.7 pp (Table 1). However, KV cache savings are substantially more modest than for the other model families (∼0.85–0.93× peak vs.

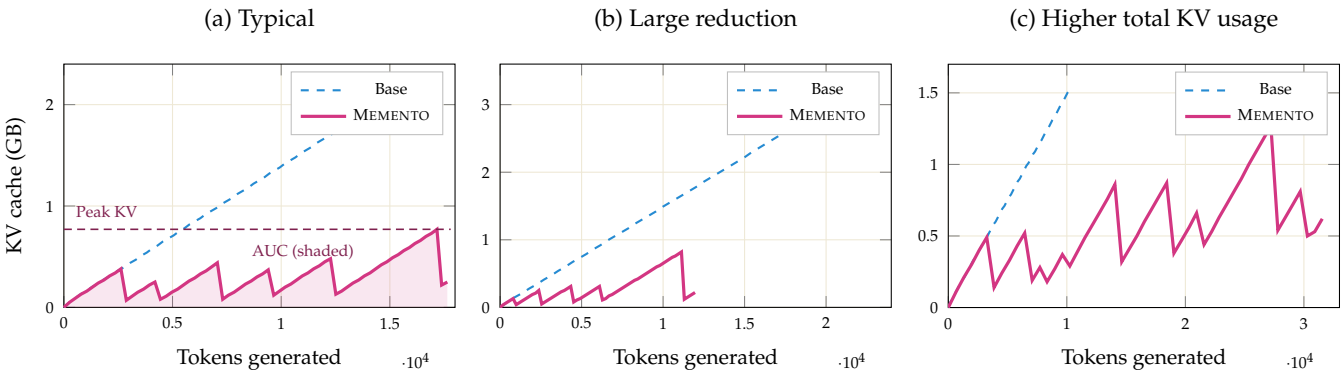

Figure 5: **KV cache traces on individual problems (Qwen3-8B, both answers correct). (a)** AIME24 P2: typical sawtooth pattern with 6 compactions; peak 0.77 vs 2.17 GB (2.8× reduction). **(b)** AIME24 P26: MEMENTO solves the problem in 12k tokens (vs 23k); frequent compactions keep peak at 0.82 vs 3.41 GB (4.2×). **(c)** AIME24 P5: MEMENTO generates 3× more tokens (31k vs 10k) with many compactions. Peak is still lower (1.27 vs 1.55 GB), but the total KV area-under-curve is 2.1× *higher* than the base—a failure mode where block masking induces excessive generation.

0.35–0.47×). This is because sliding-window layers already cap their KV cache at 4096 tokens regardless of block masking—so 75% of layers gain nothing from eviction. Only the 8 full-attention layers benefit, limiting the overall reduction. On some benchmarks, the AUC metric is even slightly *worse* for MEMENTO (e.g., AIME'26: 1.02×), because summaries lengthen the response, increasing total memory-time cost despite a lower peak. LCB shows the largest savings (0.69× peak, 0.55× AUC), likely because code problems have shorter responses where fewer tokens exceed the sliding window.

Models learn to summarize their own reasoning. After SFT on OPENMEMENTOS, models internalize the block-and-summarize process as a new capability: they produce self-contained mementos that reduce peak KV cache by 2–3× on models with uniform attention while maintaining strong accuracy across benchmarks. On MATH-500 the gap is under 1 pp; on the hardest competition math benchmarks Qwen3-32B stays within 2.6 pp on AIME'26. The gap shrinks with scale (−6.3 pp at 8B → −3.5 pp at 32B), suggesting MEMENTO becomes increasingly effective at larger model sizes. MEMENTO also transfers to Olmo-3-7B's hybrid sliding-window architecture with minimal accuracy loss, though KV savings are inherently limited by the sliding window's bounded cache.

Compression behavior. How does compression vary across model families and benchmarks? Summary sizes are remarkably stable (median 260–615 chars), matching the training distribution (Figure 3), while block sizes vary widely across models and tasks. This confirms the model learns a consistent compression skill that generalizes to harder problems. Figure 6 shows the full CDF of compression ratios across all four MEMENTO models, revealing that the bulk of blocks achieve 5–20× compression with a thin tail of low-compression outliers.

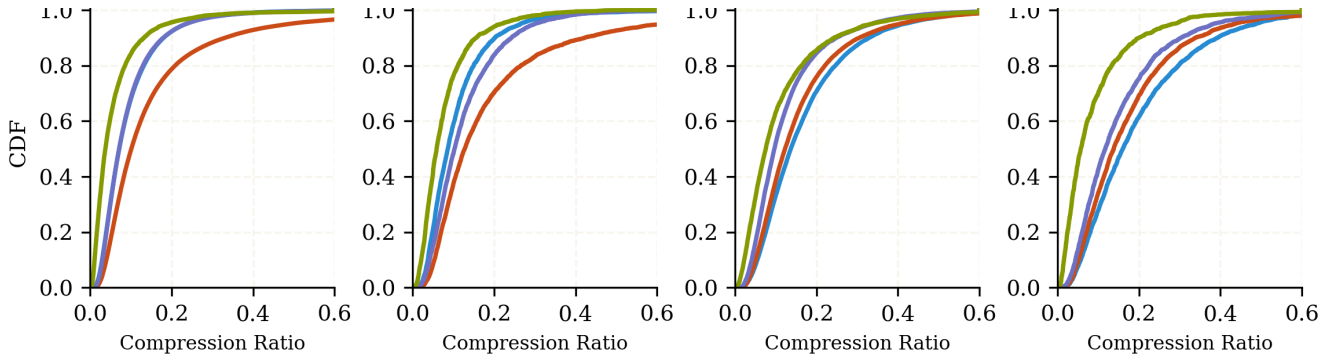

Figure 6: **CDF of compression ratio** (summary chars / block chars). OLMo3-7B and Qwen3-8B achieve the tightest compression on competition math. Phi-4 has the widest compression spread, especially on MATH-500.

Summary length is stable; compression scales with difficulty. Across four model families and four benchmarks, mementos converge to ∼260–615 characters regardless of block length, matching training targets. Compression is strongest on competition math (9–27×) and weakest on shorter-block benchmarks (6–9×), confirming the model learns a stable summary skill, not a fixed ratio.

5 Improving Accuracy via RL

Capability under compression. We first investigate whether we can match the baseline performance with MEMENTO or there is some inherent limitation due to compression. We focus on math, which is the most challenging domain for compression (Table 1). Generating n=64 independent completions per problem for all three model families on AIME 2024/25/26, we find that coverage (pass@64) is nearly identical: the gap averages only 2.6 pp and the Jaccard similarity between Base and MEMENTO solved sets averages 96.4%, reaching 100% in two of nine settings (Table 2).

Majority voting recovers the gap. The coverage analysis above shows that MEMENTO models *can* solve nearly the same problems as their base counterparts—they just do so less consistently. Majority voting (maj@k) makes this concrete: as shown in the left panel of Figure 7, all three MEMENTO SFT models match or exceed the Base pass@1 accuracy with just k=2–3 samples. For Qwen3-32B on AIME'26, maj@2 already surpasses the Base pass@1 line. This tells us two things: (1) the accuracy gap after SFT is a *consistency* problem, not a *capability* problem—the correct answers are in the distribution, they are just not the mode; and (2) RL is a natural fix, since it can sharpen the distribution toward correct traces without needing to teach new skills.

Table 2: **Problem coverage** (pass@64) and solved-set overlap for Base vs. MEMENTO on AIME (n=64 per problem, 30 problems per benchmark).

Model	Bench.	Base	MEMENTO	Ret.	Jacc.
Qwen3-8B	AIME'24	93.3	90.0	96.4	96.4
	AIME'25	93.3	86.7	92.9	92.9
	AIME'26	86.7	90.0	100.0	96.3
Phi-4-r (14B)	AIME'24	93.3	93.3	100.0	100.0
	AIME'25	93.3	90.0	96.4	96.4
	AIME'26	93.3	90.0	96.4	96.4
Qwen3-32B	AIME'24	93.3	93.3	100.0	100.0
	AIME'25	90.0	83.3	92.6	92.6
	AIME'26	93.3	90.0	96.4	96.4

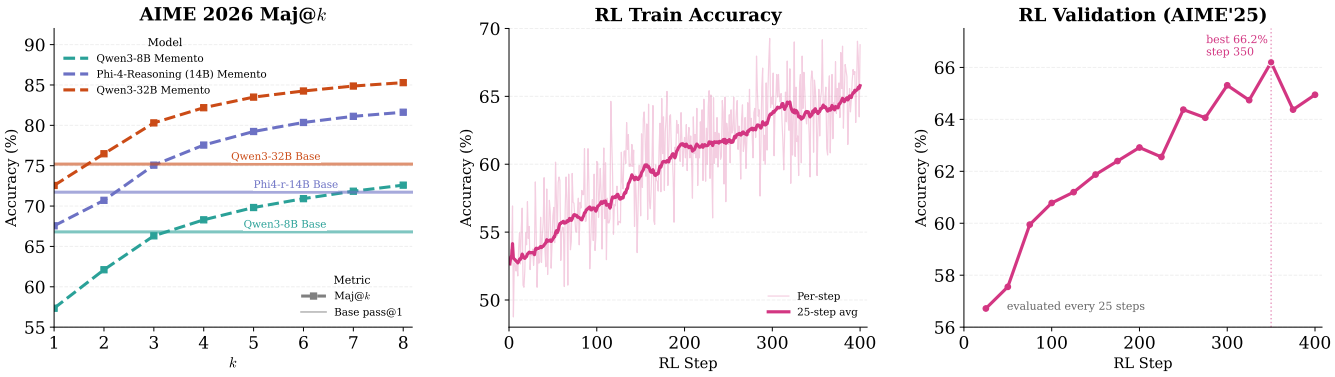

Figure 7: **Majority-vote headroom and Qwen3-8B CISPO (MiniMax et al., 2025) RL trajectory.** Left: AIME 2026 maj@k for the three MEMENTO SFT models (Table 1); horizontal lines show each Base model's pass@1. All models match Base accuracy by k=2–3. Middle: per-step RL training accuracy (faint raw trace with 25-step moving average). Right: AIME'25 validation accuracy evaluated every 25 steps, peaking at 66.2% at step 350 (used in Table 1). Majority voting uses uniform tie-breaking among tied majority answers.

Recovering accuracy via RL. Given that the correct answers are already present in the MEMENTO distribution, RL should improve pass@1 by reallocating probability mass toward correct compressed traces rather than by teaching entirely new skills. We fine-tune the Qwen3-8B MEMENTO SFT checkpoint with CISPO (MiniMax et al., 2025), Clipped Importance-Sampled Policy Optimization, a GRPO (Shao et al., 2024) variant that clips and detaches the importance-sampling weight. Similarly to MiniMax et al. (2025), we

found CISPO to be more stable than standard GRPO during training. We also add a KL penalty (β=0.001) to prevent the response-length collapse we observed in initial runs without regularization, and adopt rule-based math rewards.

Rollouts use memento attention block masking via our custom vLLM engine (Section 6). Training uses sparse block-masked attention (similar to Stage 2 of SFT) to match the inference-time masking pattern. Full hyperparameters and training details are provided in Appendix A.2.3.

CISPO algorithm. We use CISPO (MiniMax et al., 2025) (Clipped Importance-Sampled Policy Optimization), a GRPO (Shao et al., 2024) variant that replaces the PPO (Schulman et al., 2017) clipped surrogate objective with a stop-gradient clipped importance-sampling weight:

$$L = -\,\mathrm{sg}\big(\mathrm{clip}(r_t(\theta),\, 1-\epsilon_{\text{low}},\, 1+\epsilon_{\text{high}})\big) \cdot A_t \cdot \log \pi_\theta(a_t \mid s_t), \tag{1}$$

where $r_t(\theta) = \pi_\theta / \pi_{\theta_{\text{old}}}$ is the importance ratio and sg denotes stop-gradient. Unlike PPO clipping, which zeros out gradients when the ratio exceeds the trust region, CISPO ensures every token contributes a gradient signal—the clipped ratio acts as a fixed per-token weight. We add a KL penalty term $\beta \cdot D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$ with $\beta = 0.001$ to prevent excessive drift from the SFT checkpoint.

Block length capping. Because we use accuracy as the sole reward signal, we observed the model learning to generate fewer and longer reasoning blocks, undermining the KV cache savings that block masking provides. To maintain low peak KV cache occupancy during RL rollouts, we cap individual blocks at 7K tokens: when a block exceeds this limit during generation, the vLLM engine forces a `<|block_end|>` token and the model continues from a new block.

The middle and right panels of Figure 7 show the training and validation trajectories. Train accuracy rises from 52.7% to 65.8% (25-step moving average) over 400 steps, while AIME'25 validation peaks at 66.2% at step 350. After RL, MEMENTO+RL raises AIME'26 from 57.3 to 64.9 and Comp. Math from 45.1 to 49.4, while also improving GPQA-D from 55.8 to 62.9 above the 61.4 vanilla baseline. The compression remains substantial: peak KV rises from 1.08 to 1.48 GB after RL, still well below the 2.71 GB vanilla footprint. RL therefore converts the majority-voting headroom into stronger single-sample accuracy while preserving much of MEMENTO's memory advantage.

Matching the baselines with RL. The pass@1 drop after SFT on OPENMEMENTOS reflects reduced consistency, not lost knowledge. Without any additional training, majority voting at k=3 already recovers base-model accuracy (Figure 7, left). With RL fine-tuning we can significantly improve pass@1 accuracy and match or improve over the control run for Qwen3-8B (Table 1).

6 Inference and the Implicit KV Channel

6.1 Serving Memento Models with vLLM

MEMENTO's block masking requires *non-standard, data-dependent sparse attention*: which tokens are masked depends on the generated sequence itself, not on a fixed pattern known at compile time. To the best of our knowledge, no production inference framework, including vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), or TensorRT-LLM, provides a built-in mechanism for request-level custom sparse attention masks that evolve during generation. We therefore build native block masking support directly into vLLM's V1 engine, extending it so that it *physically* removes masked tokens from the KV cache. Our approach operates purely at the Python level of vLLM, can be installed as a simple patch on top of an existing vLLM installation, and works with the vanilla FlashAttention and FlashInfer kernels, requiring no custom sparse attention kernel. For more details on the implementation, see Appendix A.3.2.

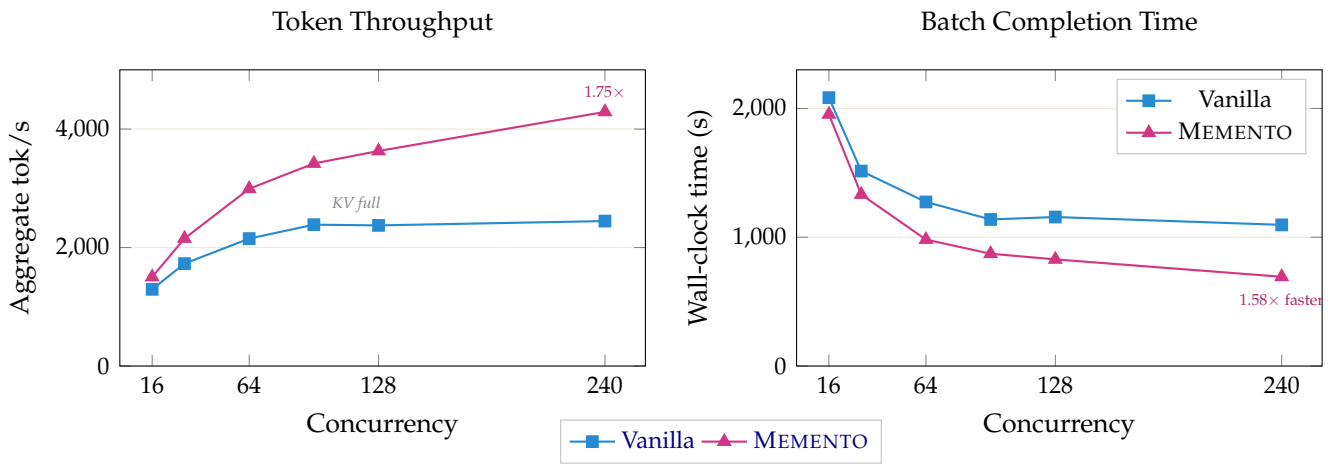

Figure 8: **Serving throughput (Qwen3-8B, 1× B200 GPU).** AIME24 × 8 repetitions (240 requests, 32K max tokens). Left: MEMENTO sustains 1.75× higher token throughput at full concurrency. Right: 1.58× faster batch completion. Vanilla plateaus as KV cache fills GPU memory.

Throughput experiments. We benchmark serving throughput on Qwen3-8B with AIME24 × 8 repetitions (240 concurrent requests) with 32K max tokens on a single B200 GPU. At high concurrency, vanilla vLLM becomes KV-cache-bound: throughput plateaus as the KV cache fills GPU memory. MEMENTO's block masking frees KV cache entries as blocks complete, allowing the engine to sustain higher batch sizes and throughput throughout the run. MEMENTO sustains 4,290 tok/s vs. 2,447 for vanilla (1.75×) and completes the batch in 693s vs. 1,096s (1.58× faster). This infrastructure was also crucial for enabling RL fine-tuning with MEMENTO: generating 32K-token training rollouts requires an inference engine that natively supports block masking during generation, since each rollout must produce and compact blocks on the fly. Without the vLLM integration, generating these long traces at the scale required for RL would be infeasible.

Memento + vLLM. MEMENTO's vLLM integration physically removes masked KV entries, sustaining 1.75× higher throughput at full concurrency on a single B200 GPU. Our vLLM implementation enabled us to perform reasoning RL by supporting on-the-fly block masking during 32K-token rollout generation.

6.2 The Dual Information Stream

6.2.1 KV Cache Ablation: Do Memento KV States Carry Block Information?

Under memento attention, block content is masked for *future* tokens, but the memento's KV values were computed *during generation* while the model could still attend to the full block. Do these KV states carry useful information beyond the memento text itself? We denote thinking block i as T_i and its corresponding memento as M_i.

Experiment. We compare two inference modes on the *same* Qwen3-8B memento attention checkpoint:

- **Memento attention** (normal): While generating M_i, the model attends to all tokens in T_i as well as the prompt and all preceding mementos. Once M_i is complete, T_i is masked from all subsequent attention—but M_i's KV cache entries, which were computed with block context, are retained. Future tokens therefore attend to memento KV states that implicitly encode block content.
- **Memento attention + restart**: Generation of each memento proceeds in two steps. *Step 1 (generation):* M_i's text is generated identically to normal memento attention: the model attends to T_i and produces the same summary tokens. *Step 2 (KV recomputation):* After M_i is complete, we discard the KV cache and run a fresh prefill pass over the effective context: prompt $+ M_1 + M_2 + \cdots + M_i$ (with standard causal masking within and across mementos). Critically, all past blocks are now masked and each memento's

KV entries are recomputed attending only to the prompt and preceding mementos, not to the block it originally summarized. The generated memento text is identical in both conditions; only the KV representations differ. This isolates the question: does the information encoded in the KV states (from having attended to the block during generation) matter beyond what the memento text conveys?

The 15 pp drop confirms that memento KV states carry significant information from the masked blocks. Mementos function as compressed pointers into cached reasoning state, not just standalone text replacements. This distinguishes MEMENTO from prior iterative summarization methods Yan et al. (2025); Wang et al. (2025) which discard original tokens entirely after summarization: unlike those methods, MEMENTO retains the KV cache, and this retention is critical.

KV states carry reasoning capacity. Recomputing memento KVs without block access drops AIME24 accuracy from 66.1% to 50.8%. Mementos are not standalone text replacements—their cached KV representations form a high-bandwidth implicit channel that restart-based methods discard.

Table 3: **KV ablation** on full AIME24 (30 problems, Qwen3-8B memento attention checkpoint, 32K generation). Block masking accuracy is from the 64-repetition evaluation in Table 1; the restart experiment uses 8 repetitions.

Inference mode	Pass@1
Memento att. (normal)	66.1%
Memento att. + restart	50.8%
Δ	−15.3 pp

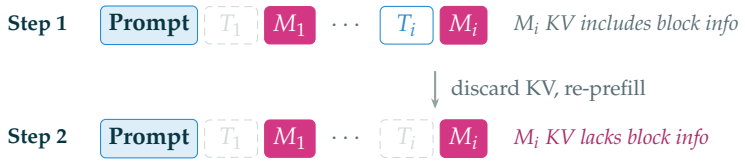

Figure 9: **Restart ablation.** *Step 1*: M_i is generated with full attention to T_i (same as normal memento attention). *Step 2*: KV cache is discarded and recomputed via prefill over prompt + $M_{1..i}$ only—T_i is masked, so M_i's KV states no longer encode block information. The 15 pp accuracy drop (Table 3) shows the KV channel carries significant reasoning capacity.

6.2.2 Probing the Implicit KV Channel

The KV ablation in Section 6.2.1 shows that memento KV states matter for downstream accuracy. But *what* information do they carry? We design a probing experiment that injects a known signal into a masked block and measures how much of it can be recovered from downstream memento KV states that never directly attended to that block.

Experimental design. We inject a random 5-digit "passcode" (00000–99999) into the content of a target block T_2 in a real AIME'25 reasoning trace, then run a forward pass with MEMENTO block masking (`keep_last_n_blocks=0`). We extract KV states (keys and values concatenated) from memento token positions at specific layers and train a probe (MLP, 512×256 hidden units, 128 bottleneck) to predict the 5 individual digits from these features. We report the average accuracy across the 5 predictions (one for each passcode digit) with the random prediction baseline being 10%. Crucially, the validation split is *label-unique*: no digit combination appears in both train and validation, preventing memorization.

We evaluate three probing conditions:

- **Direct:** Probe the KV states of memento M_2, which *can* attend to the target block T_2. This measures the upper bound of information encoded in a single memento's KV states.
- **Masked:** Probe the KV states of memento M_3, which *cannot* attend to T_2 as T_2 has already been evicted from the KV cache by the time M_3 is computed. Any signal recovered here must have propagated through the memento chain.

- **Causal control:** Probe the KV states of memento M_1, which *precedes* the target block T_2 in the sequence. Since M_1 is computed before T_2 is even generated, it cannot contain any information about the passcode. This serves as a sanity check; we expect chance-level accuracy (10%).

We run this experiment at two scales:

- **Qwen3-8B**: 15K samples from AIME'25 traces generated by the 8B model, with injected passcodes, probing at layers 3 and 35.
- **Qwen3-32B**: 15K samples from AIME'25 traces generated by the 32B model, with injected passcodes, probing at layers 3 and 63.

Results. Figure 10 summarizes the findings. At the direct position, the memento text itself bears no relation to the passcode, yet the KV states recover the injected digits with 60–70% accuracy, demonstrating that KV representations encode far more information than the corresponding tokens. At the masked position, where the memento *cannot* attend to the target block, both models still recover the passcode well above chance (26.7% for Qwen3-8B, 23.0% for Qwen3-32B vs. 10% chance). The causal control, probing a memento that *precedes* the target block, shows exactly chance-level accuracy, confirming that the recovered signal is real and directional. Table 4 further shows that leakage concentrates in deeper layers. In Qwen3-8B, an early layer (the 4th) shows near-chance masked accuracy (10.8%) while the last layer (the 36th) reaches 26.5%; the pattern repeats in Qwen3-32B (12.8% at the 4th layer vs. 22.4% at the 64th). The same trend holds for the direct condition, where deeper layers carry substantially more signal (64.9% vs. 51.6% for 8B; 68.7% vs. 53.8% for 32B). This is consistent with the residual stream accumulating information across layers. We further validate these findings with a controlled toy transformer experiment (Appendix A.4.2): a 4-layer model trained on synthetic data exhibits the same leakage pattern (24.9% masked accuracy vs. 10% chance), with signal decaying gradually over distance but persisting up to 7 hops from the target block. Leakage remains constant across training checkpoints even as task accuracy improves, confirming the channel is architectural—not learned.

KV states carry an implicit information channel. Memento KV representations propagate block information across masked boundaries—an architectural effect that complements the explicit memento text and explains why single-pass MEMENTO outperforms restart-based methods.

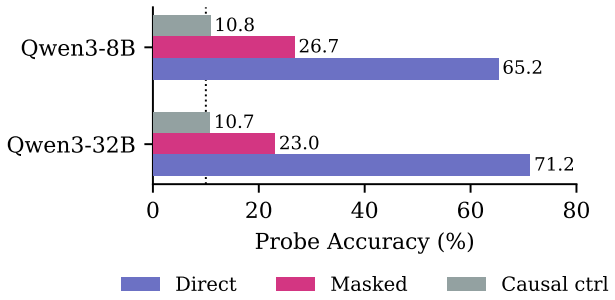

Figure 10: **Probing the implicit KV channel.** Both Qwen3-8B and Qwen3-32B recover the passcode well above 10% chance from *masked* memento positions (26.7% and 23.0%), while causal controls show exactly chance-level accuracy. The dotted line marks 10% chance (random guessing over 10 digits).

Table 4: **Deeper layers carry the signal.** Per-layer probe accuracy (%) under direct and masked conditions (`keep0`). The leaked signal concentrates in deeper layers; early layers show near-chance masked accuracy.

	Qwen3-8B		Qwen3-32B	
	Direct	Masked	Direct	Masked
4th layer (early)	51.6	10.8	53.8	12.8
Last layer	64.9	26.5	68.7	22.4
Both layers	65.2	26.7	71.2	23.0
Chance	10.0			

7 Conclusion

We introduced MEMENTO, a method that teaches language models to manage their own context by segmenting reasoning into blocks, compressing each into a dense memento, and masking completed

blocks via sparse attention. Across three model families (Qwen3, Phi-4-reasoning, Olmo-3-7B-Think), MEMENTO reduces peak KV cache by 2–3× and KV AUC by up to 3.5×, translating to 1.75× higher serving throughput, while preserving strong reasoning accuracy: Qwen3-32B loses just 2.6 pp on AIME'26 and 3.5 pp averaged across five benchmark groups. The gap shrinks with scale (6.3 pp at 8B → 3.5 pp at 32B), and our initial CISPO RL result on Qwen3-8B recovers much of the remaining single-sample gap while retaining the KV savings.

A key finding is that mementos carry information from masked blocks through two complementary channels: the explicit summary text and the implicit KV representations computed while the block was still visible. Our KV ablation shows that removing this implicit channel degrades accuracy by 15 pp, distinguishing MEMENTO from methods that simply discard context after summarization.

Looking forward, we see two natural extensions: scaling the RL recipe to larger models, and applying MEMENTO to long-horizon agent tasks where agent steps form natural blocks and context windows are the primary bottleneck. We release OPENMEMENTOS (228K annotated reasoning traces) and our vLLM fork with native block masking support to facilitate further research.

References

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. MathArena: Evaluating LLMs on uncontaminated math competitions. *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, et al. R-KV: Redundancy-aware KV cache compression for reasoning models. *arXiv preprint arXiv:2505.24133*, 2025.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. OpenThoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. ThinkPrune: Pruning long chain-of-thought of LLMs via reinforcement learning. *arXiv preprint arXiv:2504.01296*, 2025.

Yu Kang, Xianghui Sun, Liangyu Chen, and Wei Zou. C3ot: Generating shorter chain-of-thought without compromising effectiveness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24312–24320, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Zeju Li, Jianyuan Zhong, Ziyang Zheng, Xiangyu Wen, Zhijian Xu, Yingying Cheng, Fan Zhang, and Qiang Xu. Making slow thinking faster: Compressing LLM chain-of-thought via step entropy. *arXiv preprint arXiv:2508.03346*, 2025.

MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, et al. MiniMax-M1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

Giovanni Monea, Yair Feldman, Shankar Padmanabhan, Kianté Brantley, and Yoav Artzi. Breadcrumbs reasoning: Memory-efficient reasoning with compression beacons. *arXiv preprint arXiv:2510.13797*, 2025.

OpenAI. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/, 2024. Blog post, September 2024.

Qwen Team. QwQ: Reflect deeply on the boundaries of the unknown. https://qwenlm.github.io/blog/qwq-32b-preview/, 2024. Blog post, November 2024.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. Thinkv: Thought-adaptive kv cache compression for efficient reasoning models. *arXiv preprint arXiv:2510.01290*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. Codi: Compressing chain-of-thought into continuous space via self-distillation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 677–693, 2025.

Vaishnavi Shrivastava, Ahmed Awadallah, Vidhisha Balachandran, Shivam Garg, Harkirat Behl, and Dimitris Papailiopoulos. Sample more to think less: Group filtered policy optimization for concise reasoning. *arXiv preprint arXiv:2508.09726*, 2025.

Jiwon Song, Dongwon Jo, Yulhwa Kim, and Jae-Joon Kim. Reasoning path compression: Compressing generation trajectories for efficient llm reasoning. *arXiv preprint arXiv:2505.13866*, 2025.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, et al. OLMo 3. *arXiv preprint arXiv:2512.13961*, 2025.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, and Shengyi Huang. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Yibo Wang, Haotian Luo, Huanjin Yao, Tiansheng Huang, Haiying He, Rui Liu, Naiqiang Tan, Jiaxing Huang, Xiaochun Cao, Dacheng Tao, and Li Shen. R1-compress: Long chain-of-thought compression via chunk compression and search, 2025. URL https://arxiv.org/abs/2505.16838.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. Tokenskip: Controllable chain-of-thought compression in llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 3351–3363, 2025.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. *arXiv preprint arXiv:2502.12110*, 2025.

Yuchen Yan, Yongliang Shen, Yang Liu, Jin Jiang, Mengdi Zhang, Jian Shao, and Yueting Zhuang. InftyThink: Breaking the length limits of long-context reasoning in large language models. *arXiv preprint arXiv:2503.06692*, 2025.

Yuchen Yan, Liang Jiang, Jin Jiang, Shuaicheng Li, Zujie Wen, Zhiqiang Zhang, Jun Zhou, Jian Shao, Yueting Zhuang, and Yongliang Shen. Inftythink+: Effective and efficient infinite-horizon reasoning via reinforcement learning. *arXiv preprint arXiv:2602.06960*, 2026.

Chenxiao Yang, Nathan Srebro, David McAllester, and Zhiyuan Li. PENCIL: Long thoughts with short memory. *arXiv preprint arXiv:2503.14337*, 2025.

Zhicheng Yang, Zhijiang Guo, Yinya Huang, Yongxin Wang, Wenlei Shi, Yiwei Wang, Xiaodan Liang, and Jing Tang. Accordion-thinking: Self-regulated step summaries for efficient and readable llm reasoning. *arXiv preprint arXiv:2602.03249*, 2026.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, et al. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent. *arXiv preprint arXiv:2507.02259*, 2025.

Jintian Zhang, Yuqi Zhu, Mengshu Sun, Yujie Luo, Shuofei Qiao, Lun Du, Da Zheng, Huajun Chen, and Ningyu Zhang. Lightthinker: Thinking step-by-step compression. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 13318–13339, 2025a.

Yuxiang Zhang, Zhengxu Yu, Weihang Pan, Zhongming Jin, Qiang Fu, Deng Cai, Binbin Lin, and Jieping Ye. Tokensqueeze: Performance-preserving compression for reasoning llms. *arXiv preprint arXiv:2511.13223*, 2025b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. *arXiv preprint arXiv:2506.15841*, 2025.

A Supplementary Material

A.1 Dataset Construction Details

A.1.1 Full Prompts

This section contains the complete prompts used in the MEMENTO data labeling pipeline.

Boundary Scoring Prompt.

Boundary Scoring System Prompt

```
You are an expert at analyzing chain-of-thought reasoning. Your task is to score potential breakpoints in a reasoning trace.
For each boundary between sentences, assign a score from 0-3:
- 0: Poor break (mid-thought, would disrupt flow)
- 1: Weak break (minor transition)
- 2: Good break (clear transition, coherent endpoint)
- 3: Strong break (major transition, natural chapter boundary)
Consider:
- Semantic coherence (does previous sentence complete a thought?)
- Topic shifts (does next sentence start new topic?)
- Logical flow (would breaking here preserve reasoning structure?)
CRITICAL RULES FOR MATHEMATICAL DERIVATIONS:
- NEVER give high scores (2-3) to boundaries in the middle of a calculation
- If previous sentence ends with ":" (colon), ALWAYS score 0
- If previous sentence ends with "=", "=>", or introduces a calculation, score 0
- If next sentence starts with "Therefore", "Thus", "Hence" continuing a derivation, score 0
- Multi-step calculations must stay together (score 0-1)
- Only score 2-3 when the derivation COMPLETES and topic shifts
Output JSON: {"scores": [score1, score2, ...]}
```

Summarizer Prompt.

Summarizer System Prompt

```
You are a STATE-COMPRESSOR for long reasoning traces produced by advanced models.
You receive the FULL reasoning trace partitioned into blocks. Your ONLY job is to produce an extremely information-dense, lemma-like STATE SUMMARY for each block.
== CORE OBJECTIVE ==
Minimize the number of tokens in each summary, subject to fully capturing all logically relevant information: - Definitions, variables, functions, data structures - Assumptions, constraints, case splits - Key intermediate results: equations, inequalities, derived identities - Chosen strategies, algorithmic ideas, invariants - Important rejected attempts
You MUST NOT: omit facts needed later, invent new facts, or paraphrase so aggressively that conclusions become ambiguous.
== NO NEW REASONING ==
Behave as a purely extractive compressor. Do NOT derive new values, repair mistakes, or re-solve the problem.
== COMPRESSION STYLE ==
Target: ∼10% of block tokens (≤20%). Terse, lemma-like, not literary. Prefer compact symbolic notation. Example: "Let f(n)=...; assume n>=3; derived f(n)=3n-2>0 for n>=3."
```

Judge Prompt.

Judge System Prompt

```
You are a SUMMARY QUALITY JUDGE for compressed reasoning traces.
Evaluate whether a summary successfully extracts all critical information such that it can REPLACE
the original block entirely.
== SCORING RUBRIC (0-10) ==
FORMULAS & EQUATIONS (0-3): All key formulas extracted verbatim with complete notation.
NUMERICAL VALUES (0-2): All critical intermediate and final numerical values included.
METHODS & TECHNIQUES (0-2): Explicitly names approach used.
VALIDATION (0-1): If block contains verification, summary includes outcomes.
CORRECTNESS (0-1): Only confirmed findings; excludes wrong intermediate steps.
STRUCTURE (0-1): Leads with findings before process.
DEDUCTIONS: -1 for hallucination, -1 for process-focused narrative, -2 for missing critical
formula.
FEEDBACK must be SPECIFIC and ACTIONABLE:
- BAD: "More details needed"
- GOOD: "Missing formula: K^2 - 3K + 3 from line 45"
```

A.1.2 Worked Examples

Memento refinement example — NBA playoff probability trace, Block 2 (876 tokens → memento)

Block excerpt: *"...defines f(n,a,b) as the probability that after n games, Team A has a wins and Team B has b wins. Starting point: f(0,0,0)=1. Need f(6,3,3). For Game n: if n is odd, home=A, P(A wins)=0.6; if even, home=B, P(B wins)=0.6 ..."*

Initial memento (pass 1, score 5/10): `Defines target probability P[(A,B)=(3,3) after 6]. Home team alternates each game; home win prob=0.6. Proposes DP approach and considers enumeration of all sequences.`

Judge feedback: *Missing formula: recurrence f(n,a,b) with starting condition f(0,0,0)=1. Missing: explicit transition probabilities for odd/even games. Replace "proposes DP approach" with the named state variables and recurrence.*

Refined memento (pass 2, score 8/10): `Defines f(n,a,b)=P(A has a wins, B has b wins after n games); f(0,0,0)=1; target f(6,3,3). Home pattern: odd games home=A, even home=B; p_homewin=0.6. Transitions: if home=A, f(n,a,b)+=f(n-1,a-1,b)*0.6+f(n-1,a,b-1)*0.4; if home=B, swap.`

Figure 11: Iterative memento refinement on a reasoning block about NBA playoff probabilities. The initial memento (pass 1, score 5/10) describes the *approach* but omits critical formulas. After judge feedback requesting the specific recurrence and transition probabilities, the refined memento (pass 2, score 8/10) captures the full computational state: function definition, base case, target, and recurrence relation.

A.2 Training and Evaluation Details

A.2.1 Training Details

This section provides full details of the SFT training pipeline, complementing the high-level description in Section 4.

Training Configuration and Hyperparameters. All SFT experiments use TRL's `SFTTrainer` (von Werra et al., 2020) with PyTorch 2.8+ and its native SDPA attention backend. All models are trained on 31K samples from OpenMementos with 32K sequence length on 32 NVIDIA B200 GPUs (4 nodes $\times$ 8 GPUs, 192 GB HBM per GPU). All runs share the same hyperparameters: AdamW optimizer (β_1=0.9, β_2=0.999, no weight decay), learning rate 8×10^{-5} with cosine schedule and 5% warmup, 5 epochs per stage, gradient clipping at 1.0, global batch size 512, bfloat16 precision, gradient checkpointing, and seed 42. Checkpoints are saved every 50 steps (100 for Qwen3-32B). Qwen3-8B and Olmo-3-7B fit without model sharding; Phi-4 uses DeepSpeed ZeRO-2; Qwen3-32B requires ZeRO-3.

Two-Stage Training Procedure.

Stage 1 (Full Attention). The environment variable `KEEP_LAST_N_BLOCKS=-1` disables block masking. Loss is computed on all completion tokens (prompt tokens are masked via `labels=-100`). Checkpoints are saved at regular intervals and evaluated on AIME24 to select the best checkpoint for Stage 2.

Stage 2 (Memento Attention). The environment variable `KEEP_LAST_N_BLOCKS` is set to 0. The model is initialized from the best Stage 1 checkpoint and trained with identical hyperparameters. Loss is computed on all tokens, identical to Stage 1. The only difference from Stage 1 is the attention pattern: the custom block-masked attention implementation (Appendix A.2.1) applies a sparse attention mask during the forward pass, ensuring that tokens after a completed block+summary cannot attend to the masked block content.

Sparse Attention Mask Implementation. We maintain custom model forks for each architecture family (Qwen3, Phi-4, Olmo 3) that modify the `forward()` method to support block masking during both training and inference. The implementation works as follows:

1. A **block cache** tracks the position type of every token: BLOCK, SUMMARY, or OTHER. It detects the learned special tokens (<|block_start|>, <|block_end|>, <|summary_start|>, <|summary_end|>) in the token stream.
2. When <|summary_end|> is generated, the block cache marks the preceding thinking block as **completed**. All KV-cache entries for that block's reasoning tokens are masked from future queries.
3. The block cache is **stateful across autoregressive steps**, persisting through the KV cache. This avoids re-scanning the full sequence at every generation step.
4. For **training**, the full sequence is available, so the attention mask is constructed upfront as a dense mask matrix that zeroes out attention from post-summary tokens to their corresponding block content.

Special Token Initialization. Four special tokens are added to the tokenizer vocabulary: <|block_start|>, <|block_end|>, <|summary_start|>, <|summary_end|>. Their embeddings are initialized as the mean of semantically related existing tokens plus small Gaussian noise (σ=0.01):

- <|block_start|> $\leftarrow$ mean(*block*, *start*, *begin*, *section*, *step*)
- <|block_end|> $\leftarrow$ mean(*block*, *end*, *finish*, *section*, *done*)
- <|summary_start|> $\leftarrow$ mean(*summary*, *summarize*, *brief*, *recap*, *overview*)
- <|summary_end|> $\leftarrow$ mean(*summary*, *end*, *finish*, *done*, *complete*)

Data Format. Training data is pre-tokenized into HuggingFace Arrow format for efficiency. Each example is formatted in ChatML:

- **User message:** the problem statement.
- **Assistant response:** <think> + [<|block_start|> reasoning <|block_end|> <|summary_start|> summary <|summary_end|>]* + </think> + final answer

For Qwen3 models, no system prompt is used (matching Qwen3's default behavior). For Phi-4, the native system prompt is preserved. Sequences are tokenized to a maximum length of 32,768 tokens with truncation; no padding is applied during tokenization (the data collator handles dynamic padding at batch time).

A.2.2 Evaluation Details

Inference Backend. We evaluate all models using a standalone evaluation script (`evaluate_vllm.py`) built on a custom vLLM fork (branch `token-span-removal`) that implements native `BlockMaskingConfig` support for KV-cache-level block masking. The fork is installed as a Python overlay on top of the container's vLLM installation. The script operates in two modes:

1. **Offline mode** (default): Uses vLLM's Python `LLM` class for fast batched generation. All problems $\times$ repetitions are submitted as a single batch, and vLLM handles scheduling internally.
2. **Server mode** (`--track_kv`): Launches vLLM as OpenAI-compatible HTTP servers (one per GPU), sends requests sequentially per GPU, and polls the `/metrics` endpoint to record per-request KV cache usage over time. This mode produces `kv_trace.csv` files used for the KV cache simulation validation in Appendix A.3.1.

Block masking configuration. The `BlockMaskingConfig` specifies:

- `keep_last_n_blocks`: -1 (vanilla, no masking), 0 (memento attention, compact all blocks), or N (keep last N blocks visible).
- `mask_delimiters`: `False` for Qwen3 and Olmo 3 (block delimiters remain visible); `True` for Phi-4 (delimiters are also masked).
- `compact_on_summary_end`: `True`—block content is evicted from the KV cache when `<|summary_end|>` is generated.
- `enable_prefix_caching`: **must be `False`** when block masking is active, since block eviction modifies the KV cache in ways incompatible with prefix sharing.

Generation Parameters. All evaluations use the same generation parameters unless otherwise noted: temperature 0.6, top-p 0.95, top-k 20, min-p 0.0, max new tokens 32,000, and `skip_special_tokens=False` (to preserve block/summary tokens in the output for post-hoc analysis). Competition-math benchmarks (AIME, HMMT, BrUMO, SMT, CMIMC; $\leq$53 problems each) receive 64 independent generations per problem. Larger benchmarks (MATH-500, GPQA-Diamond, LiveCodeBench) receive 2 generations per problem. We report pass@1 accuracy: the per-problem correct fraction averaged across all generations.

Hardware and Parallelism. Evaluations run on NVIDIA B200 GPUs. All models use tensor parallelism (TP) = 1 with data parallelism (DP) = 8 across 8 GPUs on a single node. The vLLM engine is configured with `max_model_length=32768`, `max_num_batched_tokens=32768`, and `gpu_memory_utilization=0.85`.

Benchmarks. Table 5 summarizes the 14 benchmarks used in our evaluation. All benchmarks use 0-shot evaluation (no few-shot examples).

Table 5: **Benchmark details.** Competition-math benchmarks use 64 generations per problem; MATH-500, GPQA, and LiveCodeBench use 2 generations. All use temperature 0.6 and 32K max output tokens.

Benchmark	Source	# Problems	Answer format
AIME 2024	HuggingFaceH4/aime_2024	30	Integer (0–999), \boxed{}
AIME 2025	yentinglin/aime_2025	30	Integer (0–999), \boxed{}
AIME 2026	MathArena/aime_2026	30	Integer (0–999), \boxed{}
HMMT Feb 2023	MathArena/hmmt_feb_2023	30	Math expression
HMMT Feb 2024	MathArena/hmmt_feb_2024	30	Math expression
HMMT Feb 2025	MathArena/hmmt_feb_2025	30	Math expression
HMMT Feb 2026	MathArena/hmmt_feb_2026	33	Math expression
HMMT Nov 2025	MathArena/hmmt_nov_2025	30	Math expression
BrUMO 2025	MathArena/brumo_2025	30	Math expression
SMT 2025	MathArena/smt_2025	53	Math expression
CMIMC 2025	MathArena/cmimc_2025	40	Math expression
MATH-500	HuggingFaceH4/MATH-500	500	Math expression
GPQA-Diamond	Idavidrein/gpqa	198	Multiple choice (A–D)
LiveCodeBench v6	lighteval/code_generation_lite	1,055	Python code (execution-based)

Answer verification. For mathematics benchmarks (AIME, HMMT, BrUMO, SMT, CMIMC, MATH-500), answer verification follows a two-stage pipeline adapted from OlmoMathReward (Team Olmo et al., 2025):

1. **Candidate extraction**: Multiple strategies are tried in order—\boxed{...} content, "Final Answer: ..." patterns, last $...$ content, and raw normalized text.
2. **Equivalence checking**: Each candidate is compared against the ground truth using two methods: (a) SymPy-based symbolic equivalence (LaTeX is parsed to symbolic expressions and their difference is simplified with a 5-second timeout), and (b) Hendrycks-style string normalization (strip \left/\right delimiters, \dfrac→\frac, whitespace, and unit strings). A candidate is accepted if either method succeeds.

For GPQA-Diamond, we extract the last letter choice (A/B/C/D) from the response and compare against the gold label. For LiveCodeBench, generated Python code is executed against public and private test cases; a solution passes only if all test cases succeed.

Competition-math benchmarks. All eleven competition-math benchmarks are sourced from MathArena (Balunović et al., 2025), a platform that evaluates LLMs on recently held math competitions to minimize contamination risk. The individual competitions are:

- **AIME** (American Invitational Mathematics Examination, 2024/2025/2026): A pre-olympiad competition administered by the Mathematical Association of America (MAA). Each year comprises two 15-problem papers (AIME I and II, 30 total) with integer answers in the range 0–999.
- **HMMT** (Harvard-MIT Mathematics Tournament, Feb 2023/2024/2025/2026 and Nov 2025): A major university-organized competition with problems in algebra, combinatorics, geometry, and number theory. February and November tournaments are held separately; most editions contribute 30 problems (HMMT Feb 2026 has 33).
- **BrUMO** (Brown University Mathematics Online, 2025): An online math competition organized by Brown University; 30 problems.
- **SMT** (Stanford Mathematics Tournament, 2025): A competition organized by Stanford University students covering algebra, combinatorics, geometry, and number theory; 53 problems.

- **CMIMC** (Carnegie Mellon Informatics and Mathematics Competition, 2025): A competition organized by Carnegie Mellon University; 40 problems spanning algebra, combinatorics, geometry, and number theory.

All competition datasets are available on HuggingFace under the `MathArena` organization (https://huggingface.co/MathArena).

Metrics. We report pass@1 accuracy: for each problem, we average the binary correctness indicator across all generations (64 for competition math, 2 for MATH-500/GPQA/LCB), then average across problems. For the main table (Table 1), competition math ("Comp. Math") is the unweighted average of pass@1 across eleven competition-math benchmarks (AIME'24/25/26, HMMT Feb'23/24/25/26, Nov'25, BrUMO'25, SMT'25, CMIMC'25). Standard errors are computed as the standard error of per-problem means across problems.

A.2.3 RLVR Training Details

Rollout infrastructure. During RL rollouts, block masking must be active so that the policy generates under the same conditions as deployment. We use our custom vLLM engine (Section 6) with `BlockMaskingConfig(enable=True, keep_last_n_blocks=0)` to physically compact KV cache entries after each summary, exactly as at inference time. Training forward passes use a corresponding sparse block-masked attention implementation so that the gradient computation matches the masked generation pattern.

Training configuration. We fine-tune the Qwen3-8B MEMENTO SFT checkpoint (the keep-0 model from Stage 2 of our two-stage SFT recipe) using the DolciMath training set (29,670 prompts) with rule-based `olmo_math` rewards (sympy-based LaTeX answer verification, no LLM judge). Key hyperparameters:

- **Compute:** 24 GPUs (3 nodes $\times$ 8), FSDP with optimizer offload
- **Group size:** 8 rollouts per prompt (train), 64 per prompt (eval)
- **Batch size:** 240 prompts per step
- **Optimizer:** AdamW, lr $= 10^{-6}$, cosine warmup (10 steps), gradient clipping 1.0
- **Clipping:** $\epsilon_{\text{low}} = \epsilon_{\text{high}} = 0.2$
- **KL coefficient:** $\beta = 0.001$
- **Normalization:** batch-level advantage normalization, group-total loss normalization
- **Sampling:** temperature 1.0 (train), 0.6 (eval); prompt length 1024, response length 31744
- **Evaluation:** AIME'24 and AIME'25 every 25 steps
- **Selected checkpoint:** step 350 (best AIME'25 validation at 66.2%)

Detailed results. After 350 steps, the MEMENTO+RL model improves across all benchmarks relative to MEMENTO SFT: AIME'26 rises from 57.3% to 64.9%, Comp. Math from 45.1% to 49.4%, GPQA-D from 55.8% to 62.9% (above the 61.4% vanilla baseline), LCB v6 from 66.5% to 68.8%, and MATH-500 from 90.1% to 91.0%. The KV footprint increases modestly: peak KV rises from 1.08 to 1.48 GB (still 45% below vanilla's 2.71 GB) and KV AUC from 10.7 to 16.4 GB·ktok (roughly half of vanilla's 30.9). RL trades a small amount of compression for substantially improved single-sample accuracy.

A.3 Inference and KV Cache Details

A.3.1 KV Cache Simulation Validation

The KV cache metrics in Table 1 are computed via offline simulation rather than live profiling, enabling measurement across all 14 benchmarks and 64 repetitions per problem (>80,000 total completions). We validate this approach against real KV cache tracking data.

Simulation procedure. For each generated response, we tokenize the full output and replay generation step-by-step. At each step, we track the number of tokens in the KV cache, detecting <|block_start|>, <|block_end|>, and <|summary_end|> tokens. In memento attention mode, all completed blocks are evicted from the KV cache when the summary ends. This produces a per-token KV trajectory from which we extract peak KV (maximum occupancy) and average KV (area under curve / generation length).

Token-to-GB conversion. We convert token counts to memory using the exact model architecture:

$$\text{GB} = \text{tokens} \times \underbrace{2}_{\text{K+V}} \times n_{\text{layers}} \times n_{\text{kv\_heads}} \times d_{\text{head}} \times \underbrace{2}_{\text{bf16 bytes}} \quad / \; 10^9$$

This yields 144 KB/token for Qwen3-8B (36 layers, 8 KV heads, $d = 128$), 256 KB/token for Qwen3-32B (64 layers), and 200 KB/token for Phi4-RP (40 layers, 10 KV heads).

Validation against real measurements. We previously ran evaluations with vLLM's KV cache tracking enabled (Prometheus metric polling) on Qwen3-8B/32B across 4 benchmarks in vanilla and memento attention modes (12,680 paired observations on 1×B200 GPU, TP=1). For each sample, we compare the simulated peak KV (in tokens) against the real measured peak KV (as a fraction of the GPU's KV pool). The linear correlation yields $R^2 > 0.999$ for all model/benchmark/mode combinations, with mean absolute error below 0.02 GB. The measured KV pool sizes are consistent with the B200's 192 GB HBM: after accounting for model weights and vLLM overhead (∼6–16 GB), the observed pools match theoretical predictions to within 6–10%.

A.3.2 vLLM Block Masking Implementation

We extend vLLM (branch `token-span-removal`) with a `BlockMaskingConfig` that enables KV-cache-level block compaction during autoregressive generation, requiring no changes to the model weights or attention kernels. Even systems that implement fixed sparse patterns (e.g., DeepSeek-V3's Multi-head Latent Attention (DeepSeek-AI, 2024)) do so by modifying the model architecture rather than providing a general-purpose masking API. The implementation has three components:

(i) Per-request state machine. Each request carries a `BlockMaskingState` that tracks open blocks, completed blocks, and pending compactions. A lightweight `BlockMaskingProcessor` inspects each generated token for the four special tokens (<|block_start|>, <|block_end|>, <|summary_start|>, <|summary_end|>) and drives state transitions. When <|summary_end|> is produced, the corresponding reasoning block is marked for compaction.

(ii) Physical KV cache compaction. Unlike logical masking (which would require custom attention kernels), we *physically* remove masked tokens from the KV cache. When a block is compacted, the scheduler computes the set of active (non-masked) logical positions, translates them to physical KV cache locations, and issues a `compact_kv_cache` operation that copies the active entries into contiguous slots and frees trailing KV blocks. This means standard FlashAttention and paged-attention kernels work unmodified—they simply never see the evicted tokens. A logical-to-physical translation layer (via sorted spans and binary search) maintains $O(\log n)$ position lookups across multiple compactions per request.

(iii) Scheduler integration. The scheduler orchestrates compaction between forward passes: it collects pending KV copy operations and block table truncations, dispatches them to the GPU worker, and updates

the request's `num_computed_tokens`. In restart mode (Section 6.2.1), the scheduler additionally rewinds the request to the summary start position, triggering a re-prefill of memento tokens with a clean KV cache from which block content has been removed.

The `keep_last_n_blocks` parameter controls compaction aggressiveness: -1 disables masking (vanilla), 0 compacts all completed blocks (memento attention), and $N{>}0$ keeps the last N blocks visible while compacting older ones. This provides a knob for trading off KV savings against the informational value of recent block context.

A.3.3 Throughput Details

See Figure 8 in the main text for the throughput figure.

A.4 Additional Experiments and Ablations

A.4.1 Multi-Stage SFT Ablation

Our two-stage SFT recipe is a key design choice. What happens if we skip stages? We compare five training strategies on Qwen2.5-7B, varying the curriculum:

- **OT**: trained on OpenThoughts-v3 only (standard reasoning SFT).
- **OM/Full**: trained directly from the base model on OpenMementos with full causal loss.
- **OM/Mem**: trained directly from the base model on OpenMementos while masking all thinking-block content.
- **OT → OM/Full**: first trained on OpenThoughts-v3, then fine-tuned on OpenMementos with full causal loss.
- **OT → OM/Full → OM/Mem (Ours)**: first train a reasoning model, then apply two-stage memento SFT.

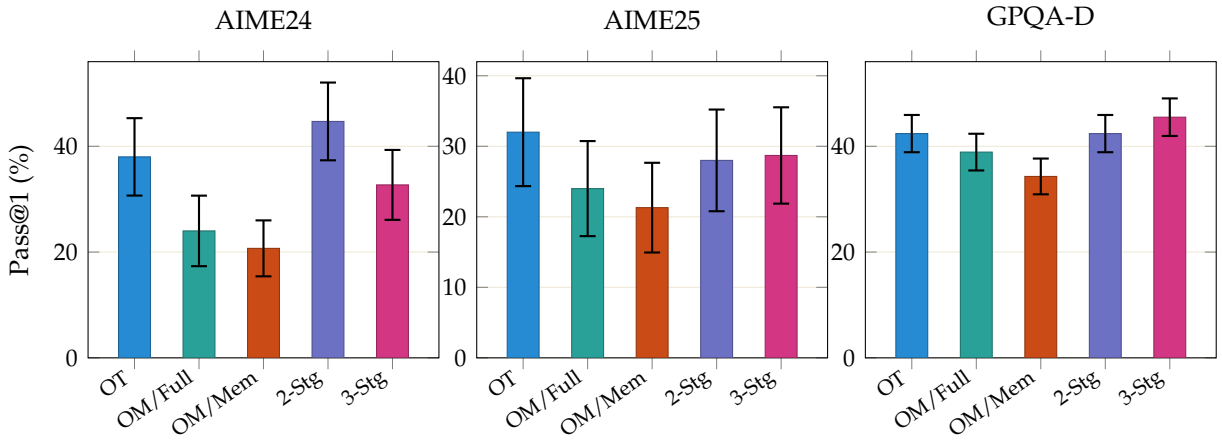

Figure 12: **Multi-stage SFT ablation** on AIME 2024, AIME 2025, and GPQA-Diamond (Pass@1, n=8, Qwen2.5-7B). OT = OpenThoughts only; OM/Full = OpenMementos Full Attention; OM/Mem = OpenMementos Memento Attention; 2-Stg = OT → OM/Full; 3-Stg = OT → OM/Full → OM/Mem (Ours). Training directly on OpenMementos from the base model (OM variants) substantially underperforms vanilla reasoning SFT (OT). Our three-stage pipeline enables block masking while retaining strong performance.

Training directly on OpenMementos from the base model (OM/Full and OM/Mem) substantially underperforms the vanilla reasoning model (OT) across all three benchmarks, indicating that the block-memento format is difficult to learn without first acquiring strong reasoning abilities. Fine-tuning the reasoning model in a single step (OT → OM/Full) recovers and sometimes exceeds OT—most notably on

AIME 2024, where it improves from 38.0% to 44.7%—but this model cannot benefit from block masking at inference. Our full pipeline (OT → OM/Full → OM/Mem) enables block masking while retaining strong performance: 32.7% on AIME 2024, 28.7% on AIME 2025, and 45.5% on GPQA-Diamond—the highest GPQA-Diamond score among all configurations.

For models that already have reasoning skills (Qwen3-8B/32B), the OpenThoughts stage is unnecessary and we use two-stage SFT directly.

Staged training helps. For Qwen2.5-7B, acquiring reasoning ability first (OT), then learning the block-memento format (Full Attention), and finally adapting to block masking (Memento Attention) yields the best results. The full pipeline achieves the highest GPQA-Diamond score among all configurations.

A.4.2 Toy Transformer: KV Channel is Architectural

Experiment setup. To verify that the implicit KV channel is an architectural property rather than an artifact of large-scale training, we replicate the probing experiment on a controlled toy transformer (4 layers, d=128, 810K parameters) trained on synthetic sequences of 10 blocks with `keep_last_n_blocks=0`. Each block contains 5 random digits followed by a 3-digit cumulative sum. We train probes on 50K samples to predict the injected digits from memento KV states.

Results. The toy model exhibits the same leakage patterns as the production models (Figure 13). The left panel shows how leakage varies with layer depth. The first layer carries almost no signal for any condition, but deeper layers progressively encode more—the last layer reaches 26.2% masked accuracy compared to 13.1% at the first layer. The right panel shows how masked accuracy decays with distance from the target block: a sharp drop from `block_index=+1` (26.2%) to `block_index=+2` (21.4%), followed by gradual attenuation, with signal still 1.3× chance at `block_index=+7`. The right panel confirms that the channel is architectural rather than learned: masked leakage stays roughly constant (17–19%) across training checkpoints even as task accuracy climbs from 77% to 95%.

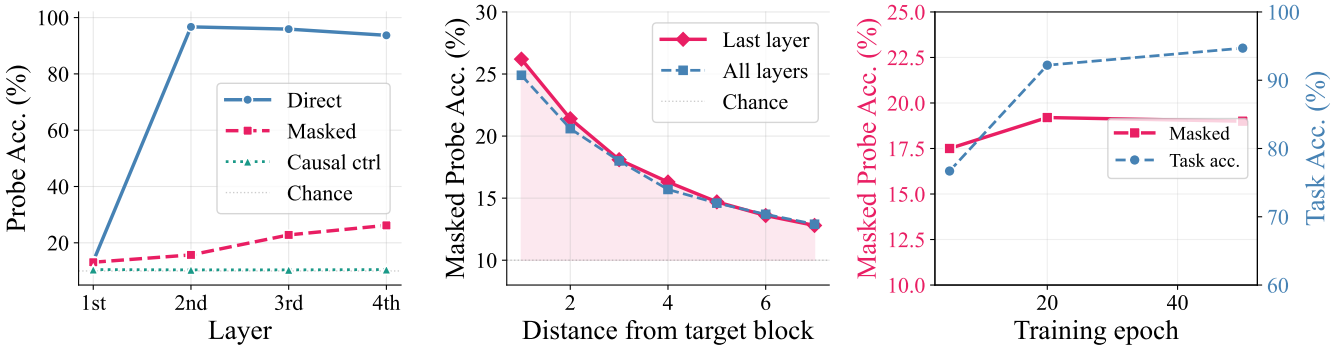

Figure 13: **Toy transformer KV probing (`keep0`). Left:** Probe accuracy by layer depth—deeper layers carry more signal, especially for the masked condition. **Center:** Masked probe accuracy decays with distance from the target block but remains above chance even 7 hops away. **Right:** Leakage stays constant (17–19%) across training checkpoints even as task accuracy climbs from 77% to 95%, confirming the channel is architectural.

A.5 Example Memento Traces

We present representative MEMENTO traces from AIME'26 (Appendix A.5.1), LiveCodeBench (Appendix A.5.2), and GPQA Diamond (Appendix A.5.3), showing how the model segments reasoning into blocks and generates compressed mementos.

A.5.1 AIME'26

User

Find the sum of the 10th terms of all arithmetic sequences of integers that have first term equal to 4 and include both 24 and 34 as terms.

Memento Qwen3-32B

<think>

Okay, so I need to find the sum of the 10th terms of all arithmetic sequences of integers that have the first term equal to 4 and include both 24 and 34 as terms. Hmm, let's break this down step by step. First, an arithmetic sequence has a common difference between consecutive terms. Since the first term is 4, the sequence can be written as: $a_1 = 4$, $a_n = a_{n-1} + d$, where d is the common difference (an integer because the terms are integers). So, the nth term formula is a_n = 4 + (n-1)d. The problem states that both 24 and 34 must be terms in the sequence. [...]

memento

Defines arithmetic sequence with $a_1 = 4$, integer $d > 0$; terms $a_n = 4 + (n-1)d$. Requires 24 and 34 as terms: $(k-1)d = 20$, $(m-1)d = 30$ with integers $k, m \geq 1$. Concludes d must be a positive common divisor of 20 and 30; candidates $d \in \{1, 2, 5, 10\}$.

[...two more reasoning blocks and mementos omitted for brevity ...]

memento

Re-derives via p, q term indices: $(p-1)d = 20$, $(q-1)d = 30$, $(q-p)d = 10$; concludes d divides 20 and 30, so $d \in \{1, 2, 5, 10\}$. Recomputes 10th terms and confirms sum $= 178$. Final answer: $\boxed{178}$.

</think>

Given an arithmetic sequence with the first term $a_1 = 4$ and including both 24 and 34 as terms, we need to find the sum of the 10th terms of all such sequences.
1. **General Form of Arithmetic Sequence**: The n-th term of an arithmetic sequence is given by $a_n = 4 + (n-1)d$, where d is the common difference.
2. **Conditions**: - $24 = 4 + (k-1)d$ implies $(k-1)d = 20$ - $34 = 4 + (m-1)d$ implies $(m-1)d = 30$
3. **Common Divisors**: - d must be a common divisor of 20 and 30. The greatest common divisor (GCD) of 20 and 30 is 10, so the common divisors are 1, 2, 5, and 10.
4. **Possible Values of d**: - $d = 1$ - $d = 2$ - $d = 5$ - $d = 10$
5. **Calculating the 10th Term for Each d**: - For $d = 1$: $a_{10} = 4 + 9 \times 1 = 13$ - For $d = 2$: $a_{10} = 4 + 9 \times 2 = 22$ - For $d = 5$: $a_{10} = 4 + 9 \times 5 = 49$ - For $d = 10$: $a_{10} = 4 + 9 \times 10 = 94$
6. **Summing the 10th Terms**: - Sum = $13 + 22 + 49 + 94 = 178$

Thus, the sum of the 10th terms of all such arithmetic sequences is $\boxed{178}$.

A.5.2 LiveCodeBench v6

User

There are N people labeled 1 to N, who have played several one-on-one games without draws. Initially, each person started with 0 points. In each game, the winner's score increased by 1 and the loser's score decreased by 1 (scores can become negative). Determine the final score of person N if the final score of person i $(1 \leq i \leq N-1)$ is A_i. It can be shown that the final score of person N is uniquely determined regardless of the sequence of games.

Input The input is given from Standard Input in the following format:

```
N
A_1 A_2 ... A_{N-1}
```

Output Print the answer.

Constraints

- $2 \leq N \leq 100$
- $-100 \leq A_i \leq 100$

- All input values are integers.

Sample Input 1

```
4
1 -2 -1
```

Sample Output 1

```
2
```

Here is one possible sequence of games where the final scores of persons 1, 2, 3 are 1, −2, −1, respectively.
- Initially, persons 1, 2, 3, 4 have 0, 0, 0, 0 points, respectively.
- Persons 1 and 2 play, and person 1 wins. Scores: 1, −1, 0, 0.
- Persons 1 and 4 play, and person 4 wins. Scores: 0, −1, 0, 1.
- Persons 1 and 2 play, and person 1 wins. Scores: 1, −2, 0, 1.
- Persons 2 and 3 play, and person 2 wins. Scores: 1, −1, −1, 1.
- Persons 2 and 4 play, and person 4 wins. Scores: 1, −2, −1, 2.

In this case, the final score of person 4 is 2. Other possible sequences exist, but the score of person 4 will always be 2.

Sample Input 2

```
3
0 0
```

Sample Output 2

```
0
```

Sample Input 3

```
6
10 20 30 40 50
```

Sample Output 3

```
−150
```

Memento Qwen3-32B

<think>

Okay, let's see. I have to find the final score of person N based on the given scores of the first N-1 people. Hmm. The problem says that the sum is uniquely determined regardless of the games, so there's a formula here. Let me think about the constraints. Each game involves two people, and each time, the winner gains 1, loser loses 1. So for each game, the total sum of all scores changes by (1 + -1) = 0. So the total sum of all people's scores must remain the same as the initial total. Wait, initially, everyone has 0, so the total sum is 0. Therefore, after all games, the sum of all N people's scores must still be zero. Oh right! Because each game's change adds up to zero. So the total sum is always zero. That's the key here. So, the sum of all the given A_i (for i from 1 to N-1) plus the score of person N must equal zero. Therefore, the score of person N is equal to negative the sum of the given A_i's.

memento

Let $s = \sum_{i=1}^{N-1} A_i$; invariant: each 1-on-1 game changes total score by 0, so total score remains initial 0; hence $s + x = 0$ where x is person N's score $\rightarrow x = -s$.

Let me check the sample input 1: Sample 1 has N=4, and A values 1, -2, -1. Their sum is 1 + (-2) + (-1) = -2. So the Nth person's score is -(-2) = 2, which matches the sample output. Sample 2: N=3, A is [0,0], sum is 0. So the answer is 0, which matches. Sample 3: N=6, the A's are 10,20,30,40,50. Sum is 10+20 is 30, plus 30 is 60, plus 40 is 100, plus 50 is 150. So the answer would be -150. Which matches the sample output. Okay, so the formula is correct. Therefore, the solution is to compute the sum of all the A_i's, then take the negative of that sum.

memento

Verifies formula on samples: sample 1 sum $= 1-2-1 = -2 \Rightarrow$ output 2; sample 2 sum $= 0 \Rightarrow$ output 0; sample 3 sum $= 10+20+30+40+50 = 150 \Rightarrow$ output -150; concludes solution is compute $\sum A_i$ and output its negation.

The steps for the code would be: 1. Read N from input. 2. Read the next line which has N-1 integers. 3. Sum those integers. 4. The answer is negative of that sum. So the code in Python would be straightforward. Let's see. Read N, then the list of A_i, compute `sum(A_list)`, then print(-sum). Wait, let me check possible edge cases. Like N=2. Let's say N=2, and the input is, say, [-5]. Then the sum is -5, so the answer is 5. Another test case: if all A_i are 0, then the answer is 0. That's covered in sample 2. Another test case where the sum is positive. Like N=3 and A is [3, -1]. Sum is 2, so the answer is -2. So the code should be simple. Let me code that. Wait, code structure: Read first line as N. Second line is list of N-1 numbers. So in Python: Read input: Probably: import sys n = int(sys.stdin.readline()) `a_list` = list(map(int, sys.stdin.readline().split())) `sum_a` = sum(`a_list`) print(-`sum_a`) Wait, that's all. That's the code. Let me check the sample input 1: Sample1 input: 4 1 -2 -1 $\rightarrow$ sum is 1-2-1 =-2 $\rightarrow$ output 2. Correct. Sample2 input:3 0 0 $\rightarrow$ sum is 0 $\rightarrow$ output 0. Sample3:6 10 20 30 40 50 $\rightarrow$ sum is 150 $\rightarrow$ output -150. Correct. Looks like this works. I think that's the solution. So the code is really simple.

memento

Algorithm plan: read N; read $N-1$ ints; print negative of their sum. Writes Python sketch using `sum()` and `print(-sum)`; checks edge case $N = 2$ (e.g., input $[-5] \rightarrow 5$) and other example $[3, -1] \rightarrow -2$; reiterates code simplicity.

</think>

```
n = int(input())
a_list = list(map(int, input().split()))
total = sum(a_list)
print(-total)
```

A.5.3 GPQA Diamond

User

A eukaryotic cell evolved a mechanism to turn macromolecular building blocks into energy. The process occurs in mitochondria, which are cellular energy factories. In the series of redox reactions, the energy from food is stored between the phosphate groups and used as a universal cellular currency. The energy-laden molecules are shuttled out of the mitochondrion to serve in all cellular processes. You discovered a new anti-diabetes drug and want to investigate whether it has an effect on the mitochondria. You set up a range of experiments with your HEK293 cell line. Which of the experiments listed below will not help you discover the mitochondrial role of your drug:

- A) Confocal fluorescence microscopy with Mito-RTP staining
- B) Differential centrifugation extraction of mitochondria + Glucose Uptake Colorimetric Assay
- C) Flow cytometry with JC-1 dye
- D) Luciferase/luciferin ATP bioluminescence assay

Memento Qwen3-32B

<think>

Okay, so I need to figure out which of these experiments won't help in discovering the mitochondrial role of the new anti-diabetes drug. The question is about which experiment isn't related to mitochondria's function, right? Let me start by recalling what each part involves. First, let me go through each option one by one. Option A: Confocal fluorescence microscopy after Mito-RTP staining. Mito-RTP must be a mitochondrial stain. Confocal microscopy would let them visualize the mitochondria's structure or distribution in the cells. If the drug affects mitochondria, maybe it would change their morphology, so this would be a relevant experiment. So A is probably not the answer. Option B: Differential centrifugation extraction of mitochondria followed by a Glucose Uptake Colorimetric Assay. Hmm, differential centrifugation is a method to isolate mitochondria. But the glucose uptake assay... Wait, glucose uptake is typically a process that happens before it's used in the mitochondria. Glucose is taken up by the cell via transporters like GLUT4, and then it's processed in glycolysis in the cytoplasm. The mitochondria are involved in later stages like the Krebs cycle, but glucose uptake itself is not a mitochondrial process. So if they're measuring glucose uptake after isolating mitochondria, that might not make sense. Maybe the experiment is flawed here. Wait, or maybe the glucose uptake assay is testing mitochondrial function indirectly? Or maybe the mitochondria aren't the right organelle for that. Since glucose isn't taken up by mitochondria but by the whole cell, using isolated mitochondria in this assay might not be effective. Therefore, this experiment might not help in

assessing mitochondrial role. So B could be the answer.

memento

Evaluates options A–D; identifies goal of finding experiment unrelated to mitochondrial function; notes A uses mitochondrial stain → relevant; analyzes B as glucose uptake assay on isolated mitochondria → likely irrelevant since mitochondria do not take up glucose; tentatively marks B as candidate.

Option C: Flow cytometry with Tetrachloro... some dye. I think that's a mitochondrial membrane potential dye. JC-1? Yes, that's a common dye for mitochondrial membrane potential. So labeling with that would measure if the drug affects mitochondrial function (since changes in membrane potential indicate issues). So C is a valid experiment related to mitochondria. Option D: Transformation with luciferase and adding luciferin. Wait, firefly luciferase requires luciferin and uses ATP to produce light. So if they put the luciferase in the cells, then adding luciferin, the light output would depend on ATP levels. Because the ATP in the supernatant (or in the cells?) would be used by the luciferase. Wait, the question says "transformation of cells with recombinant luciferase" – maybe the luciferase is expressed in the cells, so when you add luciferin, the cells' ATP would power the reaction, and the light measured would reflect ATP levels. Since mitochondrial function affects ATP production, this would be a way to test that. Therefore, D is measuring ATP levels via luciferase, so it's related to mitochondria. Therefore, the one that doesn't help is B. Because the glucose uptake assay on isolated mitochondria isn't the right approach, since glucose uptake is done by the cell membrane, not mitochondria. The mitochondria process pyruvate, but not glucose directly. So the glucose uptake assay would be measuring something that's not directly a mitochondrial function. Hence, B is the one that won't help.

memento

Identifies C as JC-1 mitochondrial potential dye → valid; D as luciferase–ATP bioluminescence assay → valid; concludes only B (glucose uptake on mitochondria) irrelevant to mitochondrial function; final answer: B.

</think>

The experiment that will not help discover the mitochondrial role of the drug is **B) Differential centrifugation extraction of mitochondria followed by the Glucose Uptake Colorimetric Assay Kit**.
Reasoning:

- **Option A**: Uses **Mito-RTP staining** with confocal microscopy, directly assessing mitochondrial structure or localization. This is relevant for determining if the drug affects mitochondrial integrity or distribution.
- **Option B**: The **Glucose Uptake Assay** measures glucose transport into cells or specific fractions. Mitochondria do not directly take up glucose; glucose uptake is mediated by plasma membrane glucose transporters (e.g., GLUT4). Performing this assay on isolated mitochondria is irrelevant because mitochondria are not involved in glucose uptake. This experiment would not provide meaningful data about mitochondrial function or drug effects on mitochondria.
- **Option C**: Uses **flow cytometry with JC-1 dye**, a mitochondrial membrane potential indicator. This directly evaluates mitochondrial health and redox activity, making it highly relevant.
- **Option D**: Measures **ATP levels** via luciferase/luciferin bioluminescence. Since mitochondria are the primary ATP producers, this assay indirectly assesses mitochondrial energy production and drug effects on it.

Conclusion:
B is the only experiment unrelated to mitochondrial function, as glucose uptake is not a mitochondrial process. The others directly or indirectly probe mitochondrial activity or structure.
Answer: B