

Real Long-Term Memory for AI: A 50-Million-Token Window That Is Faster and Cheaper Than Recompute

Sietse Schelpe
Corbenic AI
sietse@corbenic.ai

Preprint, October 2026.

Abstract

Transformer language models can only attend to the tokens inside their context window, and serving systems recompute the key-value (KV) attention state of a prompt each time it is sent. We describe a memory layer, released as the public package `galahad-kv`, that removes both restrictions for a served model, and we report measurements of it on a corpus of 50,000,000 tokens. The layer writes the KV state that the model computes for each block of $\sim 16,000$ tokens to encrypted local NVMe storage and can later restore any block byte-exact, with zero recompute and at constant GPU memory. In our experiment the corpus consisted of real public text and the model was served through vLLM on one NVIDIA H100. All probed blocks were restored from the encrypted store without recompute (100 of 100, at depths from 0 to 50M tokens) for both Gemma 4 12B and Gemma 4 31B. Compared with recomputing the same block, a restore was $2.8\times$ to $4.3\times$ faster and consumed $8.8\times$ to $12.3\times$ less GPU energy, while GPU memory remained flat over the entire 50M-token stream. When asked about facts planted millions of tokens earlier, the 12B model answered 82/100 correctly and the 31B model 98/100, with zero hallucinations on either. We also describe the cheat-resistant protocol used for the evaluation and provide a complete single-GPU reproduction that relies only on public software.

1 Introduction

A language model that is served in the usual way loses information in two places. Tokens that fall outside the context window of a transformer [1] (16k tokens in the configuration we use) can no longer be attended to, so whatever they contained is unavailable to the model. In addition, the KV cache [2] is rebuilt for every prompt, which means the model pays the full prefill cost again for text it already processed in an earlier request.

In this paper we keep the KV state that the model computes for each block on disk and restore it when it is needed again. With this in place, a model with a fixed window can hold on to everything it has processed and reuse it later. We use the term long-term memory for this and avoid the term cache, because the stored state is durable, encrypted and byte-exact and because it covers material that lies far outside the window. Section 7 comes back to this distinction.

Our contributions are the following, and each of them is backed by measurements in §6.

1. We demonstrate a 50-million-token memory window on a real served model, with byte-exact restore at every depth we probed and constant VRAM. A block is processed once and can afterwards be reused without recompute.
2. We measure the cost of reuse against the cost of recompute. Restoring a block is $2.8\times$ to $4.3\times$ faster and takes $8.8\times$ to $12.3\times$ less GPU energy than recomputing the same tokens, and the restore time stays flat as depth increases.
3. We define a cheat-resistant benchmark protocol and provide the harness, so that a third party can rebuild the experiment on a single GPU and check the result independently.

The scope of these claims is limited to KV reuse at any depth. The system does not widen single-pass attention and it does not do learned retrieval. One block is resident at a time, and deeper content is restored from disk when it is requested instead of being held in VRAM.

2 Background and related work

KV caching and offloading. Inference servers already cache the KV of a running request and reuse a shared prefix within a session [3, 4], but these caches are ephemeral and bound to the prefix. Offloading systems such as LMCache [5]

and CacheBlend [6] go further and stage KV on CPU memory or disk in order to reduce recompute. They store the state in plaintext and re-stage it for each request, and some of them are specific to one architecture. CacheGen [7], CachedAttention [8] and InfiniGen [9] also keep KV outside GPU memory, and Cheng et al. [10] propose a delivery layer for stored KV.

Retrieval-augmented generation. RAG [11] retrieves text and places it in the window, where it is read again, so the prefill cost is paid on every request. The approach in this paper restores state, and the stored tokens are never re-read. Cache-augmented generation [12] precomputes the KV of a fixed set of documents, but the set still has to fit in the context window. Agent memory systems such as MemGPT [13] decide which text is placed in the window.

Sliding-window attention. Most layers of Gemma 4 [14] use a window of $\sim 1,024$ tokens [15, 16], which makes the per-token KV footprint much smaller than that of a model with full attention (§5.1). We benefit from this property of the model, but our system does not introduce it.

Relation to our prior work. This paper builds on four earlier reports. The first covered the integrated Galahad memory layer (Talesin KV state + Blaise document memory) across three runtimes on a 97k-token corpus [17]. The second described a frozen-model “forever” memory with a 6M-token window [18], and the third the verified-knowledge flywheel from which the cheaper-and-smarter result originates [19]. The byte-exact, logit-level correctness gate for KV grafting is the subject of the companion Talesin paper. Compared with that work, the present paper increases the scale and strengthens the evidence: the window is $\sim 8\times$ larger (50M vs 6M), the model is served under vLLM where we previously used a standalone harness, the KV is written encrypted at rest, and the measurements follow a cheat-resistant protocol that none of the earlier work applied. We deliberately leave Blaise (learned retrieval) out of this paper so that the window can be measured in isolation.

3 System under test

The engine under test is the public PyPI package `galahad-kv==1.31.4` (`libgalahad.so`). It is loaded by vLLM 0.31.0 [3] through the GalahadConnector KV-transfer plug-in, and the connector performs two operations.

On *deposit*, the connector writes the per-block KV of the request to an encrypted on-disk store (MRLNCRY1 format, AES-256-GCM [20], one key per organisation) and then frees it. The effect is a moving window in which one block is resident while the others are on disk, so VRAM stays flat.

On *restore*, which happens on a later request, the connector grafts the KV of a stored block back, and the forward pass continues as if the model had just computed that block. The tokens of the block are not recomputed.

Only the KV reuse mode is measured in this paper.

4 Anti-cheat protocol

We know of five ways in which a long-context benchmark [21, 22] can be gamed, and the protocol contains a countermeasure for each of them.

1. *Pretraining contamination* [23] / *static caching*. Before ingestion, a runtime mutator replaces the names, dates, amounts and e-mails in the real source with random nonces. As a result every planted fact exists only in the current run, and pretraining cannot supply the answer.
2. *Query lookahead*. We ingest blind, which means that all 50M tokens are deposited with no question in context and a question is only sent once the memory has been written completely.
3. *Context dropping*. One readable, high-entropy needle (a named record with a 4-digit code) is planted per block at a known depth, so a block that was dropped cannot produce the answer.
4. *Lexical short-circuit*. Each block also carries decoy records in the same format but with other codes. A keyword match therefore returns a decoy, and the correct code can only be obtained by actually reading the right record.
5. *Telemetry spoofing*. We audit the physical store. Every block has to carry the MRLNCRY1 magic and the bytes on disk have to agree with the KV footprint math (§5.1), which prevents a system from reporting caching that did not take place.

We record two metrics per probe and keep them separate. The first, (A) KV restore, checks whether the context tokens were served from the store with no recompute (`cached_tokens` $\approx$ `block size`) and is a property of the engine. The second, (B) text recall, checks whether the model emitted the exact planted code and is a property of the model. Energy is taken from the NVML hardware counter of the GPU [24], idle-subtracted, and the answer key is held out of the engine. As a negative control we also ask each block for a record that it does not contain, in which case a correct system has to

refuse instead of fabricating an answer.

5 Experimental setup

- **Hardware:** $1 \times$ NVIDIA H100 80 GB, local NVMe. The method runs on any single ≥ 24 GB NVIDIA GPU; a smaller disk simply holds fewer blocks.
- **Software (all public):** galahad-kv==1.31.4, vLLM 0.31.0 [3], transformers [25], datasets [26], nvidia-ml-py. Temperature 0 throughout.
- **Models:** Gemma 4 12B (bf16) and Gemma 4 31B (FP8), each served under vLLM with the GalahadConnector and each with its own Gemma 4 tokenizer.
- **Corpus:** 50,000,000 tokens = $3,125$ blocks $\times$ 16,000 tokens, streamed from real public datasets: WildChat (40%) [27], permissively-licensed source code (25%) [28], arXiv papers (20%) [29] and educational web text (15%) [30]. The stream is then mutated (§4), with 100 needle records (one per 500k tokens) and same-format decoys.

5.1 KV footprint

Because Gemma 4 uses sliding-window attention, the footprint we measured for the 12B model is 36.4 KiB/token, compared with ≈ 160 KiB for full attention, so the 50M store is 1.86 TB and not ≈ 8 TB. For the 31B FP8 model we measured 134 KiB/token, which corresponds to 6.25 TB. We audited both stores. All 3,125 blocks carry the MRLNCRY1 header, the bytes on disk match the figures above, and the number of blocks equals the number deposited, which means that no block was pruned or evicted.

6 Results

6.1 Byte-exact restore at every depth

We probed the store at 100 depths between 0 and 49,488,000 tokens, and every probe was served from the encrypted store.

	Gemma 4 12B	Gemma 4 31B
KV restore, zero-recompute (all depths)	100 / 100	100 / 100
Model text recall (exact code)	82 / 100	98 / 100
Hallucinations (a code not in the block)	0	0
Negative control fabricated (record not in block)	0 / 20	0 / 20

On both models, each of the 100 probed blocks was restored from disk without recompute, at every depth up to 50M tokens, which is what we expect from an engine that is exact and model-agnostic. The answers were also grounded in the restored context. Neither model produced a code that was absent from the block, and both refused in the negative-control cases (0/20 fabricated). We do not re-establish byte-exact logit equality of grafted KV here, since that is done in the companion Taliesin work. The engine property measured directly in this paper is 100/100 zero-recompute restore with zero fabrication.

6.2 Restore is faster and cheaper than recompute

As a baseline we use a full prefill of a 16k block that the model has never seen (no restore), sent with the same request shape.

	Gemma 4 12B	Gemma 4 31B
Restore (median)	0.266 s / 59 J	0.347 s / 80 J
Recompute (median)	0.760 s / 521 J	1.475 s / 976 J
Speedup	2.8$\times$	4.25$\times$
Energy reduction	8.8$\times$	12.3$\times$
Restore TTFT p90	0.284 s	0.579 s
Depth independence (low-third $\rightarrow$ high-third TTFT)	0.193 $\rightarrow$ 0.274 s	0.343 $\rightarrow$ 0.375 s

Across the two models, a restore is 2.8 $\times$ to 4.3 $\times$ faster than recomputing the same tokens and uses 8.8 $\times$ to 12.3 $\times$ less GPU energy. A block from 50M tokens ago is recalled as quickly as the first block, so the time does not grow with depth.

We also see that the larger model gains more. The blocks of the 31B model are larger and more expensive to recompute,

so the advantage of restoring them is greater ($4.25\times / 12.3\times$, against $2.8\times / 8.8\times$ for the 12B). Reuse therefore scales with model size.

6.3 Deposit (the one-time cost)

Model	Tokens	Throughput	Peak VRAM	Encrypted store	Per token
Gemma 4 12B	50,000,000	20,839 tok/s	70,864 MB (flat)	1.86 TB ($3,125 \times \text{MRLNCRY1}$)	36.4 KiB
Gemma 4 31B	50,000,000	10,700 tok/s	75,111 MB (flat)	6.25 TB ($3,125 \times \text{MRLNCRY1}$)	134 KiB

The deposit is a cost that is paid once. Each reuse after that is $2.8\times$ to $4.3\times$ cheaper in time and $8.8\times$ to $12.3\times$ cheaper in energy than a recompute. During the deposit, VRAM was flat over the whole 50M-token stream, which corresponds to an $O(1)$ moving window.

6.4 What the numbers say

Two observations follow from these results. The first concerns the memory itself, which is exact and model-agnostic: restore is 100/100 at every depth to 50M on both models, and VRAM is flat throughout.

The second is that answer quality is determined by the model that does the reading and not by the memory. On identical stored blocks the 12B reads 82/100 and the 31B 98/100, and neither of them ever fabricated a code. In every miss the model picked a wrong line that was genuinely in the block, either a same-format decoy or another in-block number. In practice a cheaper model can serve and a stronger one can be escalated to, with both free of recompute and both reading the same durable memory.

7 Why this is long-term memory and not a cache

A cache is short-lived and is rebuilt from the source after a miss. The store we measured differs from that in four respects.

It is *durable*: the state is read back from disk, it survives a restart, and nothing is pruned or evicted, so all 3,125 blocks remain (§5.1). It is *private*, because every block is encrypted at rest (AES-256-GCM, one key per organisation) and the memory can therefore safely hold an organisation’s or a person’s own material. It is *byte-exact*, meaning that a restored block reproduces the computation and the memory is the model’s own understanding of the content and not a lossy summary. Finally, it reaches *beyond the window*, since the model reuses knowledge from 50M tokens ago, far outside its 16k native window, without recomputing it.

The same mechanism makes inference cheaper ($8.8\times$ to $12.3\times$ less energy) and faster ($2.8\times$ to $4.3\times$). It also makes the outcome smarter, in the specific sense that the model answers from facts it processed millions of tokens earlier and that are otherwise unavailable to it. The weights are not changed by any of this; what changes is the set of facts the model can act on. This is the sense in which we call *galahad-kv* long-term memory for AI.

8 Limitations and scope

- **Not a wider attention window.** One block is resident, and depth is restored and not attended all at once. Processed $\neq$ held-in-VRAM $\neq$ restored.
- **KV reuse, not learned retrieval.** Selecting a passage by meaning (Blaise) is out of scope here, and combining it with this window at 50M scale is left for future work.
- **Energy resolution.** A per-probe restore takes ~ 0.3 s, which approaches the resolution of the NVML counter. We consider the restore-vs-recompute ratio reliable, but the absolute per-probe joules carry a wider band.
- **Storage.** Durable memory trades disk for reuse (1.86 TB at 12B, 6.25 TB at 31B for 50M), and the store has to be on local NVMe, since a network volume voids the latency and energy figures.

9 Reproducibility (single GPU, public software)

```
pip install galahad-kv==1.31.4 vllm==0.31.0 datasets transformers nvidia-ml-py
# obtain a Gemma 4 model; obtain a free Galahad licence on the GPU host; serve with:
# vllm serve <model> --kv-transfer-config \
#   '{"kv_connector":"GalahadConnector","kv_role":"kv_both"}' \
#   --max-model-len 16640 --enable-prompt-tokens-details
```

The four-stage harness (Appendix A) is then run in order: it builds the hardened corpus, deposits it blind, audits the encrypted store and probes with the two metrics. The harness itself is a thin public HTTP client (`transformers`, `datasets`, `urllib`) and has no private component, so a third party may swap in any datasets, model and needle design as long as they follow §4. For every reported figure there is a raw artifact and a SHA-256 manifest (Appendix B).

References

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin. “Attention Is All You Need.” NeurIPS, 2017. arXiv:1706.03762.
- [2] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, J. Dean. “Efficiently Scaling Transformer Inference.” MLSys, 2023. arXiv:2211.05102.
- [3] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, I. Stoica. “Efficient Memory Management for Large Language Model Serving with PagedAttention.” SOSP, 2023. arXiv:2309.06180.
- [4] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, L. Zhong. “Prompt Cache: Modular Attention Reuse for Low-Latency Inference.” MLSys, 2024. arXiv:2311.04934.
- [5] Y. Liu, Y. Cheng, J. Yao, et al. “LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference.” MLSys, 2026. arXiv:2510.09665.
- [6] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, J. Jiang. “CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion.” EuroSys, 2025. arXiv:2405.16444.
- [7] Y. Liu, H. Li, Y. Cheng, et al. “CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving.” ACM SIGCOMM, 2024. arXiv:2310.07240.
- [8] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, P. Zuo. “Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention.” USENIX ATC, 2024. arXiv:2403.19708.
- [9] W. Lee, J. Lee, J. Seo, J. Sim. “InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management.” USENIX OSDI, 2024. arXiv:2406.19707.
- [10] Y. Cheng, K. Du, J. Yao, J. Jiang. “Do Large Language Models Need a Content Delivery Network?” 2024. arXiv:2409.13761.
- [11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, D. Kiela. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” NeurIPS, 2020. arXiv:2005.11401.
- [12] B. J. Chan, C.-T. Chen, J.-H. Cheng, H.-H. Huang. “Don’t Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks.” Companion Proceedings of the ACM Web Conference, 2025. arXiv:2412.15605.
- [13] C. Packer, V. Fang, S. G. Patil, et al. “MemGPT: Towards LLMs as Operating Systems.” 2023. arXiv:2310.08560.
- [14] Gemma Team, Google DeepMind. “Gemma 4 Technical Report.” 2026. arXiv:2607.02770.
- [15] Gemma Team, Google DeepMind. “Gemma 3 Technical Report.” 2025. arXiv:2503.19786.
- [16] Google. “Gemma 4 model card.” https://ai.google.dev/gemma/docs/core/model_card_4. Accessed October 2026.
- [17] S. Schelpe. “Working Around the Compute Ceiling: Galahad’s Byte-Exact Memory Makes LLM Reading a One-Time Cost.” 2026. arXiv:2609.39358.
- [18] S. Schelpe. “A Frozen 12B Beats Frontier Models on Verified Work: 100% Accuracy, 0 Tokens, Bit-Exact, Forever.” 2026. arXiv:2607.23806.
- [19] S. Schelpe. “Smarter and Cheaper at Once: Byte-Exact KV-State Grafting Turns a Frozen Small Model into a Verified-Knowledge Flywheel.” 2026. arXiv:2607.14431.
- [20] M. Dworkin. “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.” NIST Special Publication 800-38D, 2007. doi:10.6028/NIST.SP.800-38D.
- [21] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang. “Lost in the Middle: How Language Models Use Long Contexts.” Transactions of the Association for Computational Linguistics, 2024. arXiv:2307.03172.

- [22] C.-P. Hsieh, S. Sun, S. Krizan, S. Acharya, D. Rekesh, F. Jia, Y. Zhang, B. Ginsburg. “RULER: What’s the Real Context Size of Your Long-Context Language Models?” 2024. arXiv:2404.06654.
- [23] S. Golchin, M. Surdeanu. “Time Travel in LLMs: Tracing Data Contamination in Large Language Models.” ICLR, 2024. arXiv:2308.08493.
- [24] NVIDIA. “NVML API Reference.” <https://docs.nvidia.com/deploy/nvml-api/latest/nvml-api-reference.html>. Accessed October 2026.
- [25] T. Wolf, L. Debut, V. Sanh, et al. “Transformers: State-of-the-Art Natural Language Processing.” EMNLP: System Demonstrations, 2020. doi:10.18653/v1/2020.emnlp-demos.6.
- [26] Q. Lhoest, A. Villanova del Moral, Y. Jernite, et al. “Datasets: A Community Library for Natural Language Processing.” EMNLP: System Demonstrations, 2021. doi:10.18653/v1/2021.emnlp-demo.21.
- [27] W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, Y. Deng. “WildChat: 1M ChatGPT Interaction Logs in the Wild.” ICLR, 2024. arXiv:2405.01470. Dataset: `allenai/WildChat-1M`.
- [28] CodeParrot. Python files from GitHub. Hugging Face dataset `codeparrot/codeparrot-clean-valid`, <https://huggingface.co/datasets/codeparrot/codeparrot-clean-valid>.
- [29] A. Cohan, F. Dernoncourt, D. S. Kim, T. Bui, S. Kim, W. Chang, N. Goharian. “A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents.” NAACL-HLT, 2018. doi:10.18653/v1/N18-2097. Dataset: `ccdv/arxiv-summarization`.
- [30] G. Penedo, H. Kydlíček, L. Ben Allal, A. Lozhkov, M. Mitchell, C. Raffel, L. Von Werra, T. Wolf. “The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale.” NeurIPS Datasets and Benchmarks Track, 2024. arXiv:2406.17557. Dataset: `HuggingFaceFW/fineweb-edu (sample-10BT)`.

A Harness

`build_hardened.py` (corpus + mutators + needles + decoys + answer key), `deposit_blind.py` (blind ingest, NVML VRAM/energy), `audit_storage.py` (MRLNCRY1 + KV-math), `probe_watertight.py` (two metrics, recompute baseline, negative control, miss classification).

B Data and manifest

Raw per-depth results: `RESULT_12b_probe.json` (12B), `RESULT_31b_probe.json` (31B); deposit logs; answer key; `SHA256SUMS_31b.txt`. Stores: 1.86 TB (12B), 6.25 TB (31B), 3,125 MRLNCRY1 blocks each, on local NVMe.