



Gate-Pinned State Memory

Exact storage in a model's recurrent state, and how far a model gets using it

Abstract

A coding agent remembers by pasting text into its prompt, and pays for it on every turn. Divyansh Agrawal's [qwenputer](#) showed another way: a Gated DeltaNet head with its gates pinned is exact, addressable RAM, and a filesystem can be compiled into extra layers of a model without training¹. We call the method Gate-Pinned State Memory (GPSM), checked every link of it on our own stack, and then measured whether it helps an agent remember.

The storage holds up. [tang](#)'s kernel stores and reads one-hot rows bitwise over a 50,000-step free run, while the trained heads we measured do not qualify: in GLM-5.3-Flash, none of 8,192 channels per layer holds its gate at exactly 1. On an RTX 3090 Agrawal's model drove its store through tool calls and read files back from a small saved file in an empty conversation, with its output on prose and code close to the noise floor of the base model. [tang](#) ran the compiled layers bitwise in its reference math, emitted all 168 of their tensors bitwise from its own compiler, and served them from a CUDA window inside the engine at about 0.9 ms per token. Agrawal's compiled CPU also reproduced: 39 of 40 large multiplications against 22 for the base model.

As memory for an agent, it does not yet work. On a cross-session benchmark (LoCoMo, 125 questions) the GPSM model scored an answer F1 of 0.05, below a model with no memory at all (0.10), while retrieval over the old transcripts scored 0.44 with 1,127 prompt tokens. On 60 exact facts it recalled 23 percent against 92 percent for retrieval. The store was not the problem: when the right file existed and the model fetched it, the exact text came back. The failures were in the model's policy: what it chose to write, a store it filled and could not reclaim, and lookups that looped until its budget ran out. Replacing the policy confirmed it: with the engine writing and searching, the same store matched retrieval on every question of the full set, and engine search alone recovered most of the gap; clean writes or a larger budget did not help while the model still looked up. GPSM is exact storage that costs no prompt tokens to hold; as memory it is only as good as the policy that drives it, and the model's own policy is not good enough yet.

1 Motivation

A coding agent’s memory today is text. Notes, summaries and retrieved chunks go into the prompt, are prefilled on every turn, and compete with code for the context window. On a 16 GB laptop running a 4B model, context is the scarcest thing there is. The alternatives in the literature each pay somewhere else. KV caches are exact but large and tied to one model². Learned fixed-size states cost no context but forget^{3,4}. Test-time training writes facts into weights, at the price of optimizer state and lost general ability⁵.

Gated DeltaNet (GDN) updates a matrix state per head with the delta rule⁶:

$$S_t = g_t S_{t-1} + \beta_t k_t (v_t - S_{t-1}^T k_t)^T$$

With the decay $g = 1$, the write strength $\beta = 1$ and a one-hot key $k = e_i$, the update replaces row i of S with v and leaves every other row alone.

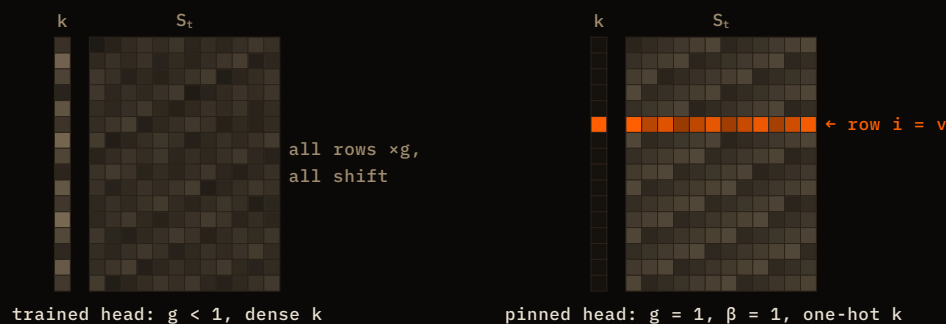


FIG 1 The delta rule, drawn. Sixteen rows stand for a 128×128 head state. A trained head decays every row and smears the write across all of them. A pinned head with a one-hot key replaces exactly one row and leaves the rest untouched.

A one-hot query reads row i back. Compile weights that produce those gates and keys, and the recurrent state is RAM. We call fixing the gates this way *gate pinning*, a head built this way a *pinned head*, and memory kept in pinned heads **Gate-Pinned State Memory** (GPSM). The gates are pinned, not the state: rows change when they are rewritten, and stay exactly as written until then. [qwenputer](#)¹ compiled a RISC-V CPU and a 512-file store into 12 extra layers in front of Qwen3.5 4B, without training. It reported a KL divergence to the base model of 6×10^{-4} , and recall of facts across chats 90 percent of the time, against 30 percent when the old chats were pasted back in.

[tang](#) is our inference engine. It runs [Qwen3-Coder-Next](#), a mixture of experts with about 3B parameters active per token and a hybrid of 36 GDN layers and 12 attention layers, and already snapshots the fixed-size recurrent state to disk. Its GDN kernel is specified for the shapes of Flash-Next, another Qwen hybrid [tang](#) serves, with 16 key and 48 value heads per layer; Coder-Next and Qwen3.5 4B have 32 value heads. [frog](#), our coding agent, talks to it. So the question was concrete: what of this transfers to an engine we own?

2 **Exact RAM in tang's kernel**

tang pins the GDN recurrence in a CPU specification that the CUDA kernel is verified against bitwise. We tested the claim there.

- **The gates reach exactly 1.** Gate pre-activations of $a + dt = -30$ for the decay and $b = 40$ for the write strength round to exactly 1.0f.
- **A write into an empty row stores v bitwise** and touches no other row of any of a Flash-Next layer's 48 value heads.
- **An overwrite stores $\text{fma}(1, v - s, s)$** , which is exactly v only when $v - s$ is exact. On random floats, 8,643 overwrites were exact and 15,933 were off by one rounding. Stored values must be integers below 2^{24} .
- **A 50,000-step free run**, driving the kernel as a register machine, stayed bitwise exact on the CPU and on an RTX 3090.
- **One token with $g < 1$ ends exactness for good.** It scales every row by a non-power-of-two, and the integers turn fractional.

3 **Trained heads do not do it**

So a pretrained head can be borrowed for storage only if its gate is bitwise 1 on every token. We measured the real gates of GLM-5.3-Flash, whose recurrent layers use a per-channel-gated delta rule, over 427 tokens and all 34 recurrent layers.

TABLE 1 Gates of a trained hybrid model. No channel qualifies as exact storage. Ranges are model layer indices; 34 of the model's layers are recurrent. Measured on GLM-5.3-Flash UD-Q4_K_XL, RTX 3090.

LAYERS	MEAN G	BEST CHANNEL MIN G	MEAN / MAX BETA	AT 1.0 EVERY TOKEN
Early (0–10)	0.51–0.68	0.999994	0.54–0.71 / ≤ 0.997	0 of 8,192
Mid (12–26)	0.73–0.87	0.999999	0.64–0.77 / ≤ 1.000	0 of 8,192
Late (28–44)	0.67–0.79	0.999997	0.55–0.84 / 1.000	0 of 8,192

A few channels are remarkably slow. Between 1 and 8 percent per layer have a mean gate above 0.999, and the best has a mean gate of $1 - 2 \times 10^{-7}$, a half-life from the gate alone of about three million tokens.

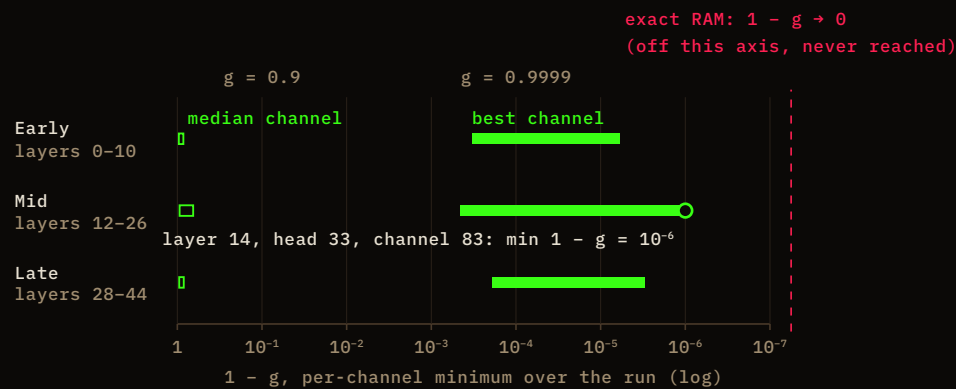


FIG 2 No channel reaches 1. Each bar is a range across the layers in its group: hollow for the median channel’s minimum gate over the run, solid for the best channel’s. GLM-5.3-Flash on an RTX 3090 over 427 tokens; per-layer distributions were not kept, only the group ranges and the best channel.

But they are analog registers, and they leak. Exact storage needs dedicated pinned heads with compiled gates, which is the qwenputer design and the basis of GPSM.

4 **Two dead ends**

The same structure suggests faster prefill. Writing context straight into state cannot work: a trained head can only read the state its own prefill would have produced, so an integer written into a row means nothing to it. We tested the two ideas that could work, and neither does.

Chunked kernels. *tang* steps the GDN recurrence one token at a time. A chunked tensor-core kernel has about 35 times the throughput on a 512-token window, but the recurrence is only 4.5 percent of a prefill window on an RTX 3090, and 5.5 percent on an H200. Prefill time lives in attention and the MoE layers.

Precomputed chunk maps. The delta rule is affine in S , so a chunk of text collapses to one matrix pair per head. If a chunk’s keys and values did not depend on earlier context, prefill would be a few matrix products. We spliced context-free chunks into real GLM-5.3-Flash runs.

TABLE 2 Context-free chunk precomputation. KL is over the 32 tokens after each spliced chunk. About 0.1 nat per token: better than skipping the chunk, not lossless.

SPLICED CHUNK	STATE ERROR, EARLY / MID / LATE	KL, MEAN / MAX (NATS)	TOP-1 AGREE
4	0.34 / 0.65 / 0.54	0.095 / 0.80	30 / 32
8	0.30 / 0.62 / 0.46	0.103 / 1.13	27 / 32
12	0.44 / 0.98 / 0.60	0.111 / 0.89	31 / 32
Skip (control)	0.5–0.7	0.30–0.58 / 3.7–5.5	26–28 / 32

Mid and late layers depend on context, not position. Fixing the error means recomputing most of the depth, which is prefill again. Instant restore of *old* context is what *tang* already ships: a resumed 7,000-token session prefills in about 0.1 s from the snapshot disk tier.

5

GPSM, end to end

5.1

An engine-driven store

Figure 3 shows the whole path. **frog** keeps the store as a file on the user’s disk and gives the model a **Memory** tool. **tang** serve loads the file into a session’s pinned heads at the start and exports it after every write. Inside the engine, store operations run as their own windows through the compiled layers, apart from the host model’s layers and KV cache.

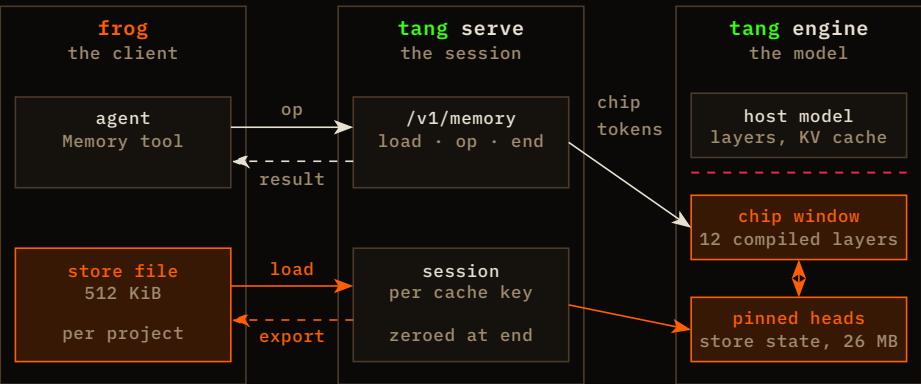


FIG 3 Architecture. Store traffic, in orange, never passes through the host model: operations run as chip tokens through the compiled layers, which read and write the pinned heads. The red line marks that separation. The store file moves only between the user’s disk and a live session.

We first built the store without compiled layers: a key-value file store in dedicated GDN heads of one layer, written and read through the real kernel with one-hot keys and pinned gates. The layout follows Agrawal’s: a hashed directory with linear probing, plus content rows of token ids. Values stay below 2^{18} , tighter than the kernel’s 2^{24} , because a read passes the output norm and a division, about eight roundings.

TABLE 3 Store capacity. The store takes 4 of a layer’s 16 key heads and their value heads, a quarter of one GDN layer. The heads column gives the whole layer.

MODEL SHAPE	HEADS, K / V	DIRECTORY	CONTENT
Flash-Next	16 / 48, tiled	1,024 slots, 512 files	65,024 tokens · 768 KiB
Coder-Next, Qwen3.5 4B	16 / 32, grouped	512 slots, 256 files	65,024 tokens · 512 KiB

A fill of each layout to capacity, 512 and 256 files, and a 10,000-operation random workload checked against a hash map all pass. The store persists through the bytes **tang**’s snapshot already carries. It also moves into a different conversation’s state while leaving that conversation’s own heads bitwise untouched.

5.2

The model drives it

Reproduction. We ran Agrawal’s code and [checkpoint](#) on an RTX 3090. In one conversation the model issued all nine store calls itself: three puts, a list, three gets, a delete and a list. Every answer was printed by the model, and 160 printed tokens matched Agrawal’s reference with no mismatches. We saved the store Agrawal’s way, as a 14,276-byte file of its occupied rows, and loaded it into an empty conversation.

The model listed the two surviving files and read both back exactly. In our run, KL to the base model was 6.9×10^{-4} against a noise floor of 6.7×10^{-4} , close to Agrawal’s reported figure, and bitwise identical with files loaded or not.

tang runs the compiled layers. tang’s reference math ran the checkpoint’s 12 compiled layers on Agrawal’s trace. All 126 tokens of a shorter put, get and list trace matched, and every store head’s 128×128 state was bitwise equal after each operation.

tang compiles them. tang-chip, a Rust port of Agrawal’s compiler, emits the layers from the circuit graph: 3,753 dimensions, 164 heads and 47,532 units. The dimensions are live at different times, so the compiler packs them into far fewer slots of the residual stream. At Qwen3.5 4B shapes, all 168 tensors equal the checkpoint bitwise, zero padding included: 2.44 billion elements, 1.24 million of them nonzero. Retargeted to Coder-Next’s shapes, its hidden size of 2,048 widened to 3,968, the store replays the trace bitwise and no weight touches the host model’s dimensions.

tang serves them. A chip window in the engine uploads the emitted layers as sparse blocks and runs store calls without touching the base layers, the LM head or the KV cache. On an RTX 3090 it ran with 0 mismatching layer steps against the CPU path. Inside a running engine, on a slot holding a prompt, store calls returned the stored ids. The slot’s tokens and greedy continuation stayed identical to a twin slot that never ran them.

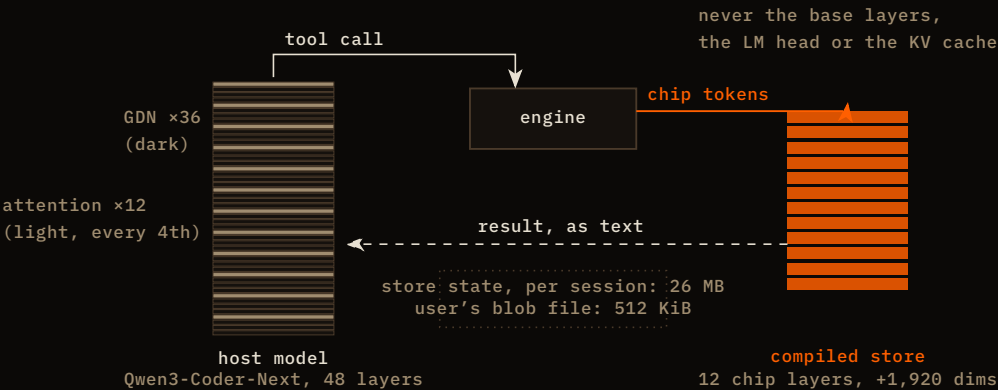


FIG 4 Where the store lives. The compiled layers sit beside the host model, not inside it. The host model emits the tool call; the engine runs it as chip tokens through the chip layers only, and the host sees the result as text.

TABLE 4 The chip window on an RTX 3090, with CUDA graphs. Agrawal’s PyTorch stack took 1.0–1.65 s per operation.

OPERATION	TOKENS	PER OP	PER TOKEN
Put	56	51.4 ms	0.92 ms
Get	43	39.6 ms	0.92 ms
List	27	25.1 ms	0.93 ms

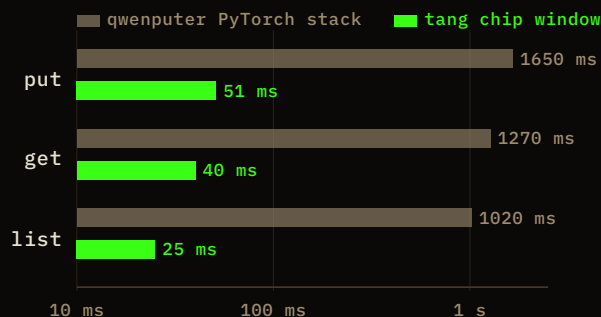


FIG 5 Latency per store operation, same RTX 3090, log scale from 10 ms. Agrawal's time is mostly PyTorch fallback kernels, not the store.

The compiled layers take 13.75 MB of sparse weights and 26 MB of state per session in **tang**, and about 332 million multiply-adds per token, roughly 8 percent of the 4B model's. That count understates the cost of the published artifact (§6.4).

5.3 Serving

tang serve loads a store per session, runs operations and exports it. An ephemeral mode zeroes the store when a session ends and never writes it to disk. **frog** loads the user's file at session start, saves after every write, and gives the model a `Memory` tool. **frog** `memory dump` prints exactly what is stored.

ASIDE • PRIVACY

The part of memory that persists becomes a file on the user's disk, 512 to 768 KiB. In a hosted session the provider sees it in plaintext while the session runs, and with the ephemeral mode keeps nothing afterwards. This is not encryption: the layout is known, so anyone holding the file can read it. The model can still repeat what it reads, so egress controls matter as before. What changes is where memory piles up: in a file you can read and delete, not with us.

6 Does it help an agent remember?

Sections 2 to 5 show that the storage is exact. This section asks the question that matters for an agent: given the store, does a model remember better across sessions? We ran Agrawal's filesystem model against four memory arms on the same base weights, Qwen3.5 4B, on one RTX 3090 with greedy decoding. The data, prompts, model revisions and scripts are [published with this post](#).

6.1 Setup

- **A0, no memory.** The question alone.
- **A1, full history.** The most recent 16,384 tokens of the earlier sessions, plus the question and template.
- **A2, retrieval.** The earlier sessions in chunks of six turns, embedded with `bge-small`, top five chunks in the prompt.
- **A3, text notes.** After each session the model writes notes of up to 400 tokens; all notes go into the prompt. This is how coding agents remember today.
- **A4, GPSM.** After each session the `filesystem model` is shown the session and its store tools, with Agrawal's system prompt, and saves what it chooses, up to 1,000

of its own tokens per session. At question time it sees only the question and may list and get files, with 300 of its own tokens to spend.

Two tests. **LoCoMo** is a standard long-conversation benchmark⁷, released under CC BY-NC 4.0; we publish our scores per question but not its conversations, questions or answers. We used 3 of its 10 conversations (80 sessions), excluded its adversarial category, and scored 125 questions with its official F1. The **exact-facts probe** plants 60 seeded values (UUIDs, paths, ports, configuration values, function signatures and hashes) in five short sessions among filler, then asks for each one in a new session, scored by exact match. Intervals are bootstrap 95 percent.

6.2 Results

TABLE 5 Cross-session QA on LoCoMo, 125 questions from 3 conversations, Qwen3.5 4B, RTX 3090. Retrieval matches full history at a fifteenth of the prompt. GPSM scores below no memory. [†] 3,565 including tool output. Intervals reflect question sampling, not variation across conversations.

ARM	LOCOMO F1	EM	PROMPT TOKENS	S / QUESTION
A0 no memory	0.10 (0.06–0.13)	0.01	52	0.3
A1 full history	0.46 (0.40–0.53)	0.18	16,711	4.1
A2 retrieval	0.44 (0.37–0.51)	0.18	1,127	0.7
A3 text notes	0.40 (0.34–0.46)	0.14	6,477	1.7
A4 GPSM	0.05 (0.02–0.08)	0.00	854 [†]	76.7

TABLE 6 Exact-facts probe, 60 seeded values asked in a later session. GPSM's reply contained the right value in 67 percent of cases but matched exactly in 23.

ARM	EXACT MATCH	CHAR. ERROR RATE	PROMPT TOKENS
A0 no memory	0.00	1.29	33
A1 full history	0.97 (0.92–1.00)	0.01	3,063
A2 retrieval	0.92 (0.83–0.98)	0.06	591
A3 text notes	0.88 (0.80–0.95)	0.16	1,743
A4 GPSM	0.23 (0.13–0.35)	2.29	835

GPSM came last on both tests. Retrieval over the old transcripts, the simplest memory arm, came close to full history at about a fifteenth of the prompt: within full history's interval on LoCoMo and at its lower edge on exact facts.

6.3 Where it fails

The logs separate the store from the model using it.

- **The store returns what was written.** On the facts probe, when the right file existed and the model looked for it, it fetched the right file 95 percent of the time and the exact value came back.
- **The model always writes and always looks.** It saved files in all 85 sessions, about 18 per LoCoMo session, and listed or fetched on every question, whether or not a relevant file existed.

- **What it writes loses information.** Notes were saved without session dates, so “when” questions returned “last year” instead of a year. Facts were filed under names unrelated to their content: one speaker’s home country was stored inside a file about a necklace.
- **It runs out of room.** Every write consumes one of 512 chunks for good, even an overwrite. Two of the three LoCoMo conversations exhausted the store, which answered “full” 83 and 67 times. In the facts probe the model also deleted its own files, 26 times across five sessions, including all 12 from the first; 18 of the 60 facts, 30 percent, were missing from the store when asked.
- **Lookups loop.** With 300 to 500 file names, the model guessed names, fetched near misses and kept going. 101 of 125 LoCoMo questions ended mid-lookup with no answer. On the 24 it did answer, F1 was 0.25.
- **It reports success when it fails.** When the store answered “full,” the model told the user the text was saved, both times we tested it.

6.4 Cost and limits of the artifact

TABLE 7 The published filesystem model against its base. The compiled model runs through Agrawal’s driver because it does not fit a plain load on 24 GB, so the speed rows compare two code paths as well as two models. KL is close to the noise floor: 1.2 times it on code and below it on prose. The prose here is a different 512-token text from the reproduction run in §5.2.

MEASURE (RTX 3090)	BASE QWEN3.5 4B	WITH COMPILED STORE
Decode, tokens/s (n = 3)	45.4 ± 0.3	31.0 ± 0.2
Prefill, tokens/s (n = 3)	4,303 ± 9	992 ± 37
Peak GPU memory	9.5 GB	18.9 GB
KL to base, Rust / Python / prose	–	5.1 / 7.3 / 6.7 × 10 ^{−4}
KL noise floor, Rust / Python / prose	4.1 / 6.1 / 7.2 × 10 ^{−4}	–

The store’s limits are fixed by its compilation: 512 writes over its lifetime, 53,248 tokens, and 104 tokens per write, beyond which a write is cut without an error. Its saved file grows from 9,746 bytes empty to 604,338 bytes with 512 small files and 2,163,655 bytes with one maximum-size file. **tang**’s own store, which runs on the same primitive, reuses deleted slots; Agrawal’s does not.

6.5 Exact computation

The same compiler emitted a RISC-V CPU into the model. On Agrawal’s arithmetic benchmark, drawn from [DeepMind’s mathematics dataset](#), 40 problems each, the [CPU model](#) solved 39 large multiplications and 40 large divisions; base Qwen3.5 4B solved 22 and 29. Here the model only has to hand a calculation to the circuit, and the exactness carries through.

6.6 Can a better policy close the gap?

The failure analysis points at the policy, so we replaced it piece by piece while keeping the store, driver, questions and scorer fixed.

- **B1, model writes, engine searches.** The model’s own files, as in A4, read back from the store; the engine retrieves the top five files for each question with the same embedding model as A2, and the model answers from them without tools.
- **B2, engine writes, model looks up.** The engine stores every session itself in six-turn chunks, as in A2, with the session date in each entry and descriptive names; the model looks up with its tools, at 300 tokens and at 1,000.
- **B3, engine writes, engine searches.** The B2 store, searched by the engine as in B1.

TABLE 8 Replacing the policy, same store. B2 ran on subsets because model-driven lookups take one to three minutes per question; its intervals are wide.

ARM	N	LOCOMO F1	N	FACTS EM
A2 retrieval (reference)	125	0.44 (0.37–0.51)	60	0.92 (0.83–0.98)
A4 GPSM as published	125	0.05 (0.02–0.08)	60	0.23 (0.13–0.35)
B1 model writes, engine search	125	0.27 (0.21–0.33)	60	0.68 (0.57–0.80)
B2 engine writes, model lookup	42	0.03 (0.00–0.07)	30	0.07 (0.00–0.17)
B2 with a 1,000-token budget	9	0.00	15	0.00
B3 engine writes, engine search	125	0.44 (0.37–0.51)	60	0.92 (0.83–0.98)

- **The store is exact as a backend.** All 351 engine-written entries read back byte- and token-identical. B3 gave the same prediction as plain retrieval on all 125 LoCoMo questions and all 60 facts.
- **Engine search recovers most of the gap.** Over the model’s own files, B1 gained 0.22 F1 on A4 (paired interval 0.16 to 0.28) and 45 points of exact match. It still trailed retrieval by 0.17 F1, because the model’s notes held the answer among the top five for only 36 percent of questions, against 78 percent for the engine’s chunks.
- **Better writes alone change nothing.** With clean, dated entries but the model still looking up, B2 matched A4 within noise (42 LoCoMo questions, 30 facts). A larger budget did not help either: on the 9 LoCoMo questions and 15 facts we could afford at 1,000 tokens, the model gave the same answers as at 300, only slower.
- **The binding failure is the model’s lookup.** With every relevant file present, the model re-fetched files until its budget ran out on two thirds of facts questions, and on LoCoMo 44 percent of its gets named files that did not exist because it split two-word names. The store’s list output, which truncates names to six tokens and runs long ones together, makes this worse; two of our naming schemes failed on it before a two-keyword scheme worked.

Ordered this way, fixing the lookup accounts for 0.22 of the 0.39 F1 gap and fixing the writes for the remaining 0.17; fixing the writes first accounts for nothing until the lookup is fixed. Once both are done by the engine, the store is a retrieval system with an exact backend, and it performs exactly like one.

7 Implications

7.1 Exact storage works; the policy does not

Everything below the tool call worked: rows were written and read bitwise, the store persisted across conversations, and the right file came back when it was asked for. Everything above it failed: what to write, under which name, when the store is full, and when to stop looking. Those are decisions a model makes in text, and an untrained 4B model made them badly. Replacing the policy confirms it (§6.6). Engine search over the store closes the gap to retrieval when the engine also writes, and most of it when the model writes. Engine writes do nothing while the model still looks up, and a larger budget does not help. What remains untested is a model that looks up well: training it on the tools, or a search operation compiled into the store itself.

7.2 Memory leaves the prompt, lookups do not

Stored files cost no prompt tokens while they sit in the state. Using them does: in the GPSM arm the context grew from 854 tokens to 3,565 on average with list output and file contents, against 1,127 for retrieval. A store only saves context if lookups are short and targeted.

7.3 Compiled, not trained

The compiled layers moved the model's output on Rust, Python and prose only slightly, to about 1.2 times the noise floor on code and below it on prose, and a compiled CPU fixed arithmetic the base model gets wrong. Exact capabilities can be added without training and with little change to the base model. The cost in the published artifact is real, 0.68 times the decode speed, 0.23 times the prefill speed and twice the memory, though some of that is the driver rather than the layers.

7.4 The engine owns the invariant

Exactness holds only while every gate in a pinned head stays bitwise at 1. So GPSM is a contract between compiled weights and an engine that routes store traffic separately and audits the store before export. **tang** can keep store tokens out of the base layers and the KV cache because it decides what runs where.

7.5 What it does not change

GPSM does not make new context cheaper. Prefill still runs every token through the attention and MoE layers, and both of our attempts to speed it up failed (§4).

8 Comparison

TABLE 9 Measured, same base model and questions. Prompt tokens are LoCoMo means; for GPSM, without and with tool output.

APPROACH	LOCOMO F1	FACTS EM	PROMPT TOKENS	NEEDS
Full history	0.46	0.97	16,711	Context to fit it
Retrieval	0.44	0.92	1,127	An embedding model
Text notes	0.40	0.88	6,477	Nothing extra
GPSM, as published	0.05	0.23	854–3,565	Compiled layers, 2× memory
GPSM store + engine search	0.44	0.92	1,127	Compiled layers and a search index

On our measurements, retrieval dominates: nearly the accuracy of full history at a fifteenth of the prompt, with no change to the model. Text notes come close behind. GPSM as published is last on accuracy, slowest by an order of magnitude per question, and the most expensive to run.

We did not run learned-state approaches such as Titans⁴, test-time training⁸, context distillation⁹, or parametric memory; their published results come from different models and benchmarks and are not comparable to the numbers above. One study of one-step LoRA memory measured free-form recall at zero⁵. Agent memory systems in the MemGPT, Mem0 and A-MEM line are elaborations of the text-notes arm^{10–12}.

With the engine writing and searching, the GPSM store matches retrieval exactly, because at that point it is retrieval with the text kept in the model’s state. It costs the same prompt tokens and adds the compiled layers. GPSM’s distinct property is exact storage that costs nothing to hold, travels as a small file and never enters the prompt until it is read. Nothing in these results shows that property improving recall; the question is whether it is worth its cost for storage and portability alone.

9 What the data supports

- **Exact storage and transfer.** Supported. Rows round-trip bitwise, the store persists in a small file, and moves between conversations without disturbing the receiving one (§5).
- **Exact computation.** Supported on one benchmark: 39 of 40 against 22 of 40 for large multiplication (§6.5).
- **Leaving the base model nearly unchanged.** Supported: KL about 1.2 times the noise floor on code and below it on prose (§6.4).
- **As the backend of engine-driven retrieval.** Supported, and no better than retrieval: identical predictions on all 185 questions (§6.6).
- **Memory driven by the model.** Not supported. GPSM scored below no memory on LoCoMo and recalled 23 percent of exact facts, against 92 percent for retrieval (§6.2); clean engine-written entries and a larger budget did not help (§6.6).

- **Small models on small machines.** Not supported today. The published artifact doubles GPU memory and lowers decode speed, and its lookups added more context than retrieval did.

10 Open questions

- **Can the model learn to look up?** Engine policies close the gap (§6.6); a trained model, or a search operation compiled into the store, is untested.
- **Is in-state storage worth its cost?** With engine search it matches retrieval exactly while adding compiled layers. Whether persistence and portability justify that is a product question these results do not settle.
- **Does it hold at scale and on Coder-Next?** We ran one 4B model, one seed and 3 of 10 LoCoMo conversations. No build on Coder-Next's weights exists yet.
- **How much of the overhead is the driver?** The speed comparison mixes two code paths; **tang's** chip window ran at about 0.9 ms per token, but not yet beside a host model at full speed.
- **Is a lifetime write budget acceptable?** Agrawal's store never frees space; **tang's** reuses slots, but that has not been run in this benchmark.

11 Conclusion

A delta-rule head with its gates pinned is exact RAM. We confirmed that in **tang's** kernel, showed that no trained head reaches it on its own, and followed the idea from Agrawal's checkpoint to our own compiler and a CUDA window inside our engine, each step bitwise. Then we measured what it buys an agent. Driven by the model, the store remembered less than no memory at all on LoCoMo, and 23 percent of exact facts against 92 percent for retrieval. Driven by the engine, it matched plain retrieval on every question, which shows that the store is exact and that, as a backend, it adds no recall beyond retrieval. The storage is exact; the model's lookup is the failure. Gate-Pinned State Memory is a sound primitive and, today, a retrieval backend with unusual properties. Whether a model can learn to use it directly is the open question.

References

1. Agrawal, D. Qwenputer: a fully functioning, deterministic RISC-V CPU inside Qwen3.5. <https://divagr.com/blog/qwenputer> (2026).
2. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. <https://arxiv.org/abs/2405.16444> (2024).
3. Stuffed Mamba: Oversized States Lead to the Inability to Forget. <https://arxiv.org/abs/2410.07145> (2024).
4. Titans: Learning to Memorize at Test Time. <https://arxiv.org/abs/2501.00663> (2025).

5. Beyond Perplexity: A Behavioral Evaluation Framework for Deployment-Memory Claims in LLM Test-Time Training. <https://arxiv.org/abs/2607.00368> (2026).
6. Gated Delta Networks: Improving Mamba2 with Delta Rule. <https://arxiv.org/abs/2412.06464> (2024).
7. Maharana, A. *et al.* Evaluating Very Long-Term Conversational Memory of LLM Agents. <https://arxiv.org/abs/2402.17753> (2024).
8. In-Place Test-Time Training. <https://arxiv.org/abs/2604.06169> (2026).
9. Latent Context Compilation: Distilling Long Context into Compact Portable Memory. <https://arxiv.org/abs/2602.21221> (2026).
10. MemGPT: Towards LLMs as Operating Systems. <https://arxiv.org/abs/2310.08560> (2023).
11. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. <https://arxiv.org/abs/2504.19413> (2025).
12. A-MEM: Agentic Memory for LLM Agents. <https://arxiv.org/abs/2502.12110> (2025).