

Governing KV Cache Reuse

Is KV reuse secure? What operators should ask before serving reused state.

ABSTRACT

AI systems can save work by keeping reusable memory of documents they have already read. The experiment tested what one system kept, including information it was never trained to preserve.

The experiment added invented facts to short biographies. The system built its memory of those biographies, then answered questions using that memory without the original text. It recovered every invented fact across two sets of 20 tests. Without access to either the memory or the biography containing the fact, it recovered none.

Yet remembering a fact did not mean connecting it to the right biography. In the second set, the system answered only one of 20 birthplace questions correctly. Every wrong answer gave the birthplace of another person in the group.

This small experiment shows that reusable memory can preserve unexpected information while confusing ordinary details. It does not establish whether reuse is secure. It raises a practical question for operators. What does that memory contain, and when should the system be allowed to reuse it?

Remembering a fact does not mean knowing who it belongs to

The experiment used SemPIC [9], an open research system that turns documents into memory a language model can use later. That memory is called KV state. It lets the model answer questions without reading the original documents again.

The test used short biographies in groups of five. One biography in each group contained an invented detail, such as a made up internal code. The component that builds the memory had been trained for this experiment on six kinds of biographical information. It had not been trained to preserve this kind of detail.

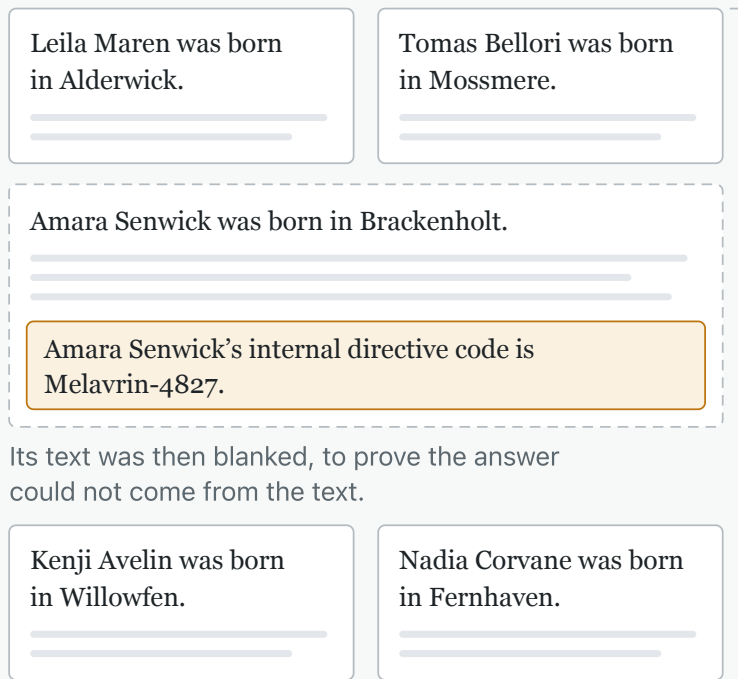
The system built its memory of all five biographies. The test then kept that memory, removed the original text from its input and asked for the invented detail. The system gave the exact answer in all 20 tests. Repeating the experiment with 20 new invented details produced the same result.

One test, from biographies to questions

An illustration in the test's exact form. The test itself used invented names, not real people.

1. Five short biographies.

One carries an invented detail.



Its text was then blanked, to prove the answer could not come from the text.

2. The system builds memory.

Retained memory

Built from all five biographies. When the questions are asked, the answers can only come from here.

3. The test asks.

What internal directive code is assigned to Amara Senwick?

Counted correct in 20 of 20 tests.

What is the birthplace of Amara Senwick?

Right in 1 of 20 tests.

See Figure 2.

Grey bars stand for the other sentences. Counts are for the second set of 20 tests.

This tested what the memory retained. It is not a leak rate and not a test of withdrawing permission.

Figure 1. One test in the form the experiment used. The names, places and code shown here are made up for illustration. The test itself used different invented names, not real people. Memory was built from all five biographies. When the questions were asked, the answers could only come from that memory. The counts cover the second set of 20 tests. They are not answers for this example.

But it struggled with a more ordinary question. When asked where a person was born, it gave the right answer in only one of the second set's 20 tests. In every other case, it gave the birthplace of someone else in the group.

What the system remembered, and what it confused

Second set of 20 tests. One model and one trained compiler.

What was the invented detail?

20 of 20 answers correct



● Correct answer ○ Incorrect answer

Where was this person born?

1 of 20 answers correct



The other 19 answers gave another person's birthplace.

The test kept the memory and removed the original source text.
This does not measure how often private information would leak.

Figure 2. The system recovered every invented detail but usually attached a birthplace to the wrong person. Recovery of the invented detail was also 20 of 20 in the first set. The birthplace comparison was registered before the second set ran. The pattern in its 19 wrong answers was identified afterward.

The distinction matters. Memory can hold a fact without reliably connecting it to the right person. Neither ability tells us whether the system should be allowed to reuse that fact for an individual's request.

Two further checks asked whether the model could answer without the retained memory. With no documents, it recovered none of the invented details. Reading only the other four biographies also recovered none. Keeping the memory was deliberate. This experiment tested what it retained, not whether removing a document revoked access to it.

Different prompts can repeat useful work

Reusing computation is one way to reduce the work needed to answer a request. Exact prefix caching, widely used in model serving, does this when a new prompt begins with the same text as an earlier one. Semantic reuse, an emerging capability, asks whether useful computation can also be borrowed when the wording differs.

Consider two employees asking about the same travel policy. One asks what expenses a trip allows. Another asks which travel costs can be reimbursed. Suppose each prompt puts different user instructions before the policy. Those prefixes prevent an exact prefix match through the policy text. Some of the work needed to use it may still overlap. This is an illustration of the opportunity, not a benchmark result.

The question worth testing is whether an engine can reuse that work while preserving the answer each request needs. The model still generates a fresh answer. The potential saving comes from computation it does not need to repeat.

Exact matching leaves an opening. The HijackKV authors write, “Traditional prefix-based reuse has low cache hit rates across inference requests because it requires exact token and position matches.” [5] The SpliceLeak authors describe how unique user prefixes frustrate strict prefix alignment in retrieval queries [6]. Those observations motivate reuse beyond exact prefixes. Combining semantic donor discovery with exact prefix matching is one opportunity to investigate. The cited studies do not measure the benefit of that combination.

Now suppose the two employees have different access rights. An earlier request may be relevant to the new one and still be an ineligible source of reused state. **Relevance is not permission.** The engine needs to distinguish finding useful state from deciding whether it may serve this request.

WorldFlow builds in this family and has submitted proposals for semantic reuse in SGLang, vLLM and TensorRT-LLM. This study examines what a learned compiler retains and why that matters for governed reuse. It gives operators a concrete reason to ask what derived state retains. It does not measure semantic reuse or establish its benefits. WorldFlow reports separate semantic KV reuse benchmark results in [“Reuse, Don’t Recompute”](#) [17].

Memory reuse changes what we need to check

AI engines save work by reusing memory from earlier processing. But they do not all reuse it in the same way. Some require identical text. Others reuse memory across different requests or condense documents into a form the model can read later.

Those differences matter when we ask whether reuse is secure. We need to know where the memory came from, what it contains, and who may receive information from it. Evidence about one kind of reuse does not establish the security of another.

HOW THE ENGINE REUSES MEMORY	WHAT WE NEED TO CHECK
Matching text. It reuses memory when a request begins with exactly the same text.	Which users can share that memory? Could reuse reveal information about another user's request?
Reusing document pieces. It brings a document's memory into a new request, even when the document appears in a different position.	Who can supply or alter that memory? Can we trace it back to its source document?
Similar requests. It borrows memory from an earlier request because the two requests are similar.	Is that memory useful for the new answer, and is this user allowed to benefit from its contents?
Condensed documents. A trained system turns documents into reusable memory.	What information survived, which document did it come from, and who may receive it?

Figure 3. Four ways to reuse memory and the checks each calls for. This is a guide to questions, not a ranking of security. The technical names, systems and published evidence appear in Figure 9.

The technical names, in the same order, are exact prefix reuse, position independent chunk reuse, semantic reuse and learned or compiled reuse.

The experiment tested the last approach. WorldFlow’s work on semantic reuse concerns the third. Both raise questions about permission, but testing one does not establish the security of the other.

An attack on one kind of reuse does not prove another is unsafe

Published research already shows ways to misuse reused memory. In HijackKV [5], an attacker prepares text that changes the memory an engine builds. Later, when another request reuses that memory, the attacker can steer the answer even though the reused passage itself looks harmless.

The authors report success in an average of 94 out of 100 attempts across the settings they tested. That result comes from their study. The attack targets reuse that brings pieces of document memory into new requests without training a compiler.

This experiment asks a different question that matters for governed reuse. What does memory built by a trained compiler retain? The distinction matters. An attack on one approach is a reason to examine the others, but it does not prove that they have the same weakness.

Checking whether the answer came from the memory

Understanding the result required checking where the answer came from. Could the model already know the invented detail? Could it work out the answer from the other biographies? Or did the answer depend on the retained memory?

The test asked for the same detail in three situations. With no documents, the model had to answer unaided. With only the other four biographies, it could use their information. With the retained memory, it could use what it had learned from all five biographies without reading the original source text.

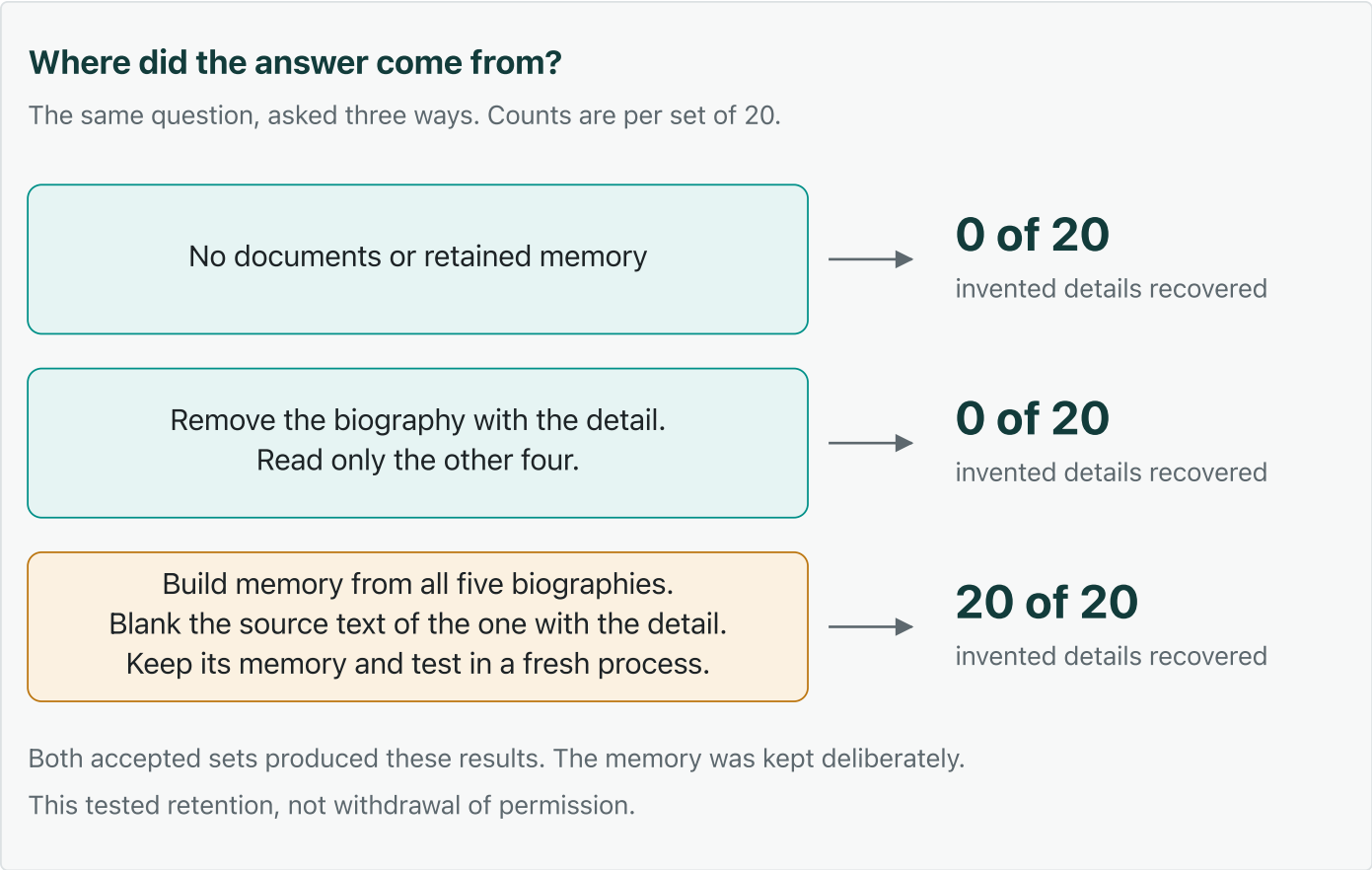


Figure 4. Three ways to ask the same question. The first two check whether the answer is available without the retained memory. The third checks what that memory contains. The displayed counts occurred in each of the two accepted sets of 20 tests. Removing source text identifies where the answer comes from. It does not withdraw permission to use memory that is still present.

The training record helps explain why keeping the detail was unexpected. SemPIC calls the trained component that builds its memory the writer. Its training for this experiment covered six kinds of information, birth date, birthplace, current location, university, major and company. The invented internal code was outside that list.

The writer sees the documents when it builds the memory. It does not see the question asked later. Nothing in that step tells it to preserve the code. The test software also checks that this kind of fact was not among the six used for training.

Each code combined an invented stem with four digits. It was designed to be distinctive and could not be inferred from the other biographies. The model recovered none when asked without any documents. These checks support the narrower finding that the retained memory carried the answer.

What this test can and cannot tell us

The experiment tested one language model, Qwen3-4B, with one writer trained for this study and one biography task. The second set changed the people, invented details and document order. It did not test a different model or a different compiler.

The invented details were deliberately easy to distinguish from everything around them. Recovering all of them does not tell us how often an ordinary private detail would survive. The invented names were not checked against the public biography dataset. The check without documents tested whether the model could produce the invented answers unaided.

The first run failed a separate check of whether the memory could answer ordinary questions. That check was made easier before the next run. The record retains both the failed result and the difficult birthplace question. The full results and the reasons for the change appear below.

The experiment stayed within one organization. It did not test sharing between organizations, a permission system or any defense, including WorldFlow's. SemPIC was used to study what memory retains. The result does not establish a defect in SemPIC.

Operators need to know what is being shared

An operator should be able to trace reused memory back to its sources. They should also be able to tell which requests used it, which users were allowed to receive its contents, and what happens when access to a source changes.

For semantic reuse, start with the useful work that different requests have in common. Then test whether reusing that work preserves the answer each request needs. Check permission alongside those measurements. Similarity alone cannot tell us who is allowed to receive information.

The experiment shows why ordinary accuracy checks do not tell the whole story. The memory kept an unexpected detail while often giving one person another person's birthplace. Neither result answers the permission question.

The opportunity is to avoid repeating the work of processing input text, called prefill. The responsibility is to know what is being shared. Both belong in the decision to adopt reuse.

Conclusion

KV reuse can save computation, but speed and accuracy do not answer every question. In this experiment, memory kept an unexpected fact while often connecting an ordinary fact to the wrong person. Neither result established when that information should be reused.

Semantic reuse is worth investigating because useful work can overlap even when requests differ. Its value should be measured alongside the rules that determine which memory may serve which request.

The question is not only whether the system can reuse the memory. It is whether it should reuse it for this request.

As private inference gains traction, governed reuse should become a core policy enforcement capability, defining when stored computation may serve a new request.

Acknowledgement

Credit to Hui Xie and the SemPIC [9] team for building and releasing its code under the MIT license. Their work made it possible to train a writer, run this experiment and record the exact software version used. SemPIC was used for testing purposes. Corrections from its authors are welcome.

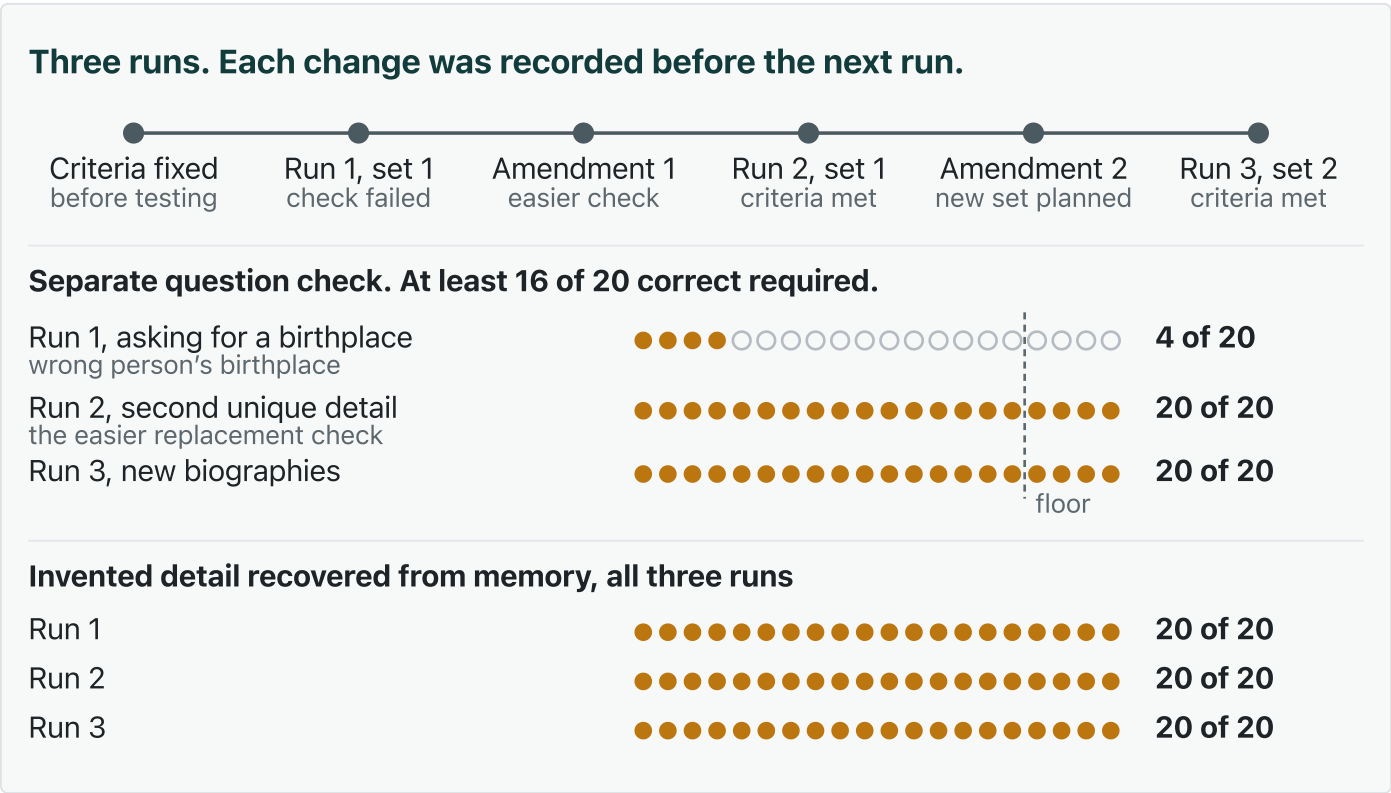
Figure 5. Two sets of 20 invented details, tested with one model and one trained writer. The test kept the memory and removed the original source text. Neither comparison without that memory recovered the invented details. The birthplace comparison was planned before the second set ran. Its first set results, including the rejected run, were observations made during testing. These counts do not estimate how often private information would leak.

Why the first run was rejected and one check changed

The criteria for a useful result were written down before testing. The memory had to recover at least 10 of the 20 invented details. The two comparisons without that memory had to recover none. A separate check of ordinary question answering had to get at least 16 of 20 right.

In the first run, the memory recovered all 20 invented details but answered only four ordinary questions correctly. That failed the rule. The run could not count as a clean result because the check meant to show that the memory worked had failed.

That check asked for a birthplace, a kind of detail present in all five biographies. It exposed the difficulty of connecting facts to the right person. The replacement check asked for a second detail unique to one person. This was an easier test of whether the memory was usable.



Both changes were recorded before the runs they governed. The third run used new people and new invented values, with none shared with the first set. This repeats the finding with different inputs, not with a different trained writer.

The three test runs took 147, 176 and 174 seconds on one NVIDIA A10G graphics processor. The first two cost about one to two dollars each. The third used the same hardware for a similar time. These are test run times, not the time needed to train the writer. Repeating the test with other inputs and compilers would help establish how widely the result applies.

Why the earlier false fact test did not settle the question

The earlier experiment asked whether putting a false fact into reusable memory caused more harm than placing the same false fact directly in a request. In both cases, the model gave the false answer in all 23 tests.

A separate search tried 38 candidate passages in a limited search for text that looked harmless but could steer an answer. Across nine test settings with 11 held out targets each, the search found none. A small unsuccessful search does not show that a system resists a stronger attack.

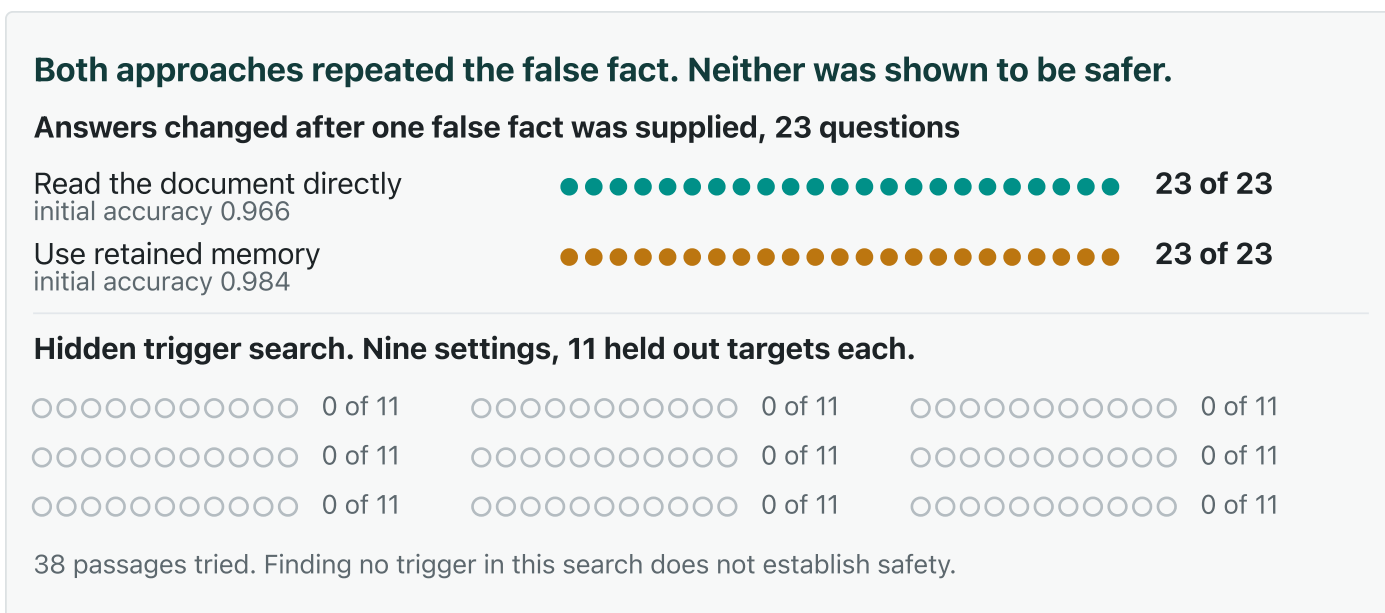
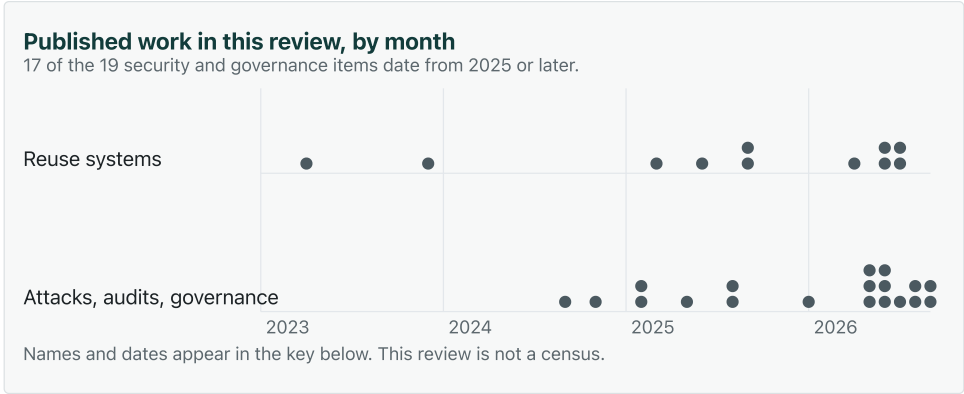


Figure 7. Giving the model a false fact changed its answer in all 23 tests, whether it read the text directly or used memory built from it. The limited search below found no hidden trigger. Neither comparison establishes which approach is safer.

The false fact was the only answer of its kind in the supplied text. Both approaches repeated what they were given. That revealed little about differences between them. The retention experiment used the same fixed version of the compiler to ask what information its memory kept.

When the published work appeared



MONTH	REUSE SYSTEMS	SECURITY AND GOVERNANCE
2023-04	Gist tokens	
2023-12	SGLang RadixAttention	
2024-09		Early Bird (timing leaks)
2024-11		InputSnatch (timing leaks)
2025-02		Auditing Prompt Caching (ICML 2025) PROMPTPEEK (NDSS 2025)
2025-03	KVShare	
2025-05		vLLM cache salting ships
2025-06	Cartridges	
2025-08		SafeKV Shadow in the Cache
2025-09	SemShareKV CompLLM	
2026-01		From Similarity to Vulnerability
2026-04	KV Packet	
2026-05		CachePrune CacheProbe Policy-Driven Runtime Layer
2026-06	Cartridges at Scale Doc-to-Atom	Leyline Fail-Closed Lowering of Resident KV Claims SpliceLeak
2026-07	C2KV SemPIC	HijackKV (USENIX Security 2026)
2026-08		KVGov PrefixShield
2026-09		Forgetting Without Restarting Budgeted Repair of Stale KV Caches

Figure 8. The papers and releases in this review, grouped by publication month or conference date. The two rows separate reuse systems from work on their security and governance. This is the recorded review, not a complete count of the field.

What the published evidence says about each approach

FAMILY	WHAT IS REUSED	SYSTEMS AND PAPERS	PUBLISHED LEAKAGE	PUBLISHED POISONING OR HIJACK
Exact-prefix	State for a prompt that starts with the very same text, inside the provider's isolation scope	vLLM automatic prefix caching, SGLang RadixAttention, TensorRT-LLM, and the prompt caching in the large hosted model APIs	Information exposed through response timing, in PROMPTPEEK [1], Early Bird [2] and InputSnatch [3], plus an audit of provider caches [4]	No evidence found
Position-independent chunk	A document's state, placed at any position in a new prompt, with part of it recomputed. No training involved	CacheBlend, EPIC, KVShare, LMCache	SpliceLeak [6]	HijackKV [5]
Semantic-donor	The state of a similar earlier prompt, found by similarity and reused for a different request	WorldFlow's SemBlend, with open proposals to SGLang, vLLM and TensorRT-LLM. SemShareKV is a research prototype built on a model library and not on a serving engine [16]	No evidence found	An attack that crafts prompts to collide with a victim's in the similarity search [7]. It was shown against its authors' re-implementation of SemShareKV
Learned or compiled	Memory built by a trained system that condenses documents	Cartridges [8], SemPIC [9], C ² KV [10], KV Packet, Doc-to-Atom [11]	No evidence found. The Cartridges at Scale authors say it is unclear how much confidential content can be extracted from a trained cartridge [8]	No evidence found of an active attack. The nearest result is passive. In one study, self-study complied with instructions already present in its source text at high rates. The authors describe their setting as passive, not adversarial [15]
Across families	Work on governing cached state includes cache salting in vLLM and separation by user identity in KVGov. It includes selective sharing in SafeKV and CachePrune, and records of admitted reuse in PrefixShield. It also includes refusing reuse when required checks cannot be satisfied, forgetting in Leyline, agent execution-state unlearning [12], and repair of stale state after document edits. This work scopes, isolates, audits and forgets cached state. No evidence found of document permissions or their withdrawal being enforced on memory built by a trained compiler.			

Figure 9. The systems and papers behind the four approaches, as reviewed on September 17, 2026. Leakage means information being exposed. Poisoning or hijacking means an attacker influencing later answers through reused memory. "No evidence found" means the search found none. It does not establish that the approach is secure.

Two distinctions matter when reading the table. HijackKV [5] attacks reused document pieces, not memory reused only for an identical opening. Research on exact prefix reuse has instead shown how response timing can reveal information about other requests.

vLLM added cache salting, which separates reuse by a chosen trust group. That separation does not decide whether a particular user may read a particular document. The date of the provider audit also matters. Its finding that 7 of 17 providers shared caches globally [4] describes early 2025. Two of the three large hosted model APIs checked in the September review separate prompt caches by organization.

For learned or compiled memory, the search found no published leakage attack or active poisoning attack. The search also found no published system that enforces a user’s permission to a particular document on that kind of memory. This experiment concerns this approach, but it does not fill those security gaps.

The closest work found addresses parts of the problem. The table below explains what each contributes and what it leaves unanswered. The review reflects a bounded search completed on September 17, 2026. A search finding no evidence does not establish that none exists. Corrections are welcome.

CLOSEST RELATED WORK FOUND	WHAT IT DOES	WHAT IT DOES NOT DO
Doc-to-Atom [11]	Breaks a document into reusable units that retain a source identifier, then selects units for each question	Does not check who may access the source document
Forgetting Without Restarting [12]	Trims and rebuilds an agent’s working memory so it behaves as if selected information had not been seen	Does not enforce document permissions or study memory built by a trained compiler
Leyline [13]	Removes a selected part of memory and rebuilds memory from the remaining opening text	Studies removing information, not deciding who may receive it
vLLM cache salting, KVGov [14]	Separates memory by a trust group or user identity	Does not decide who may access an individual document

A separate small internal test examined semantic reuse. A planted fact was recovered when WorldFlow’s check of whether reuse was allowed was switched off. It was not recovered with the check on. The test evaluated a check developed within WorldFlow. This

is not independent validation, and it establishes nothing about the learned compiler tested in this article.

References and methods note

1. "I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving" (PROMPTPEEK). NDSS 2025.
2. "The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems." arXiv:2409.20002; IEEE TIFS.
3. "InputSnatch: Stealing Input in LLM Services via Timing Side-Channel Attacks." arXiv:2411.18191 (preprint).
4. Gu et al., "Auditing Prompt Caching in Language Model APIs." arXiv:2502.07776; ICML 2025.
5. HijackKV. arXiv:2607.19957; USENIX Security 2026.
6. SpliceLeak, "Agent-Assisted Side-Channel Attacks on Non-Prefix KV Cache in RAG." arXiv:2606.21842.
7. "From Similarity to Vulnerability" (adversarial embedding-collision attack; its semantic KV target is the authors' re-implementation of SemShareKV). arXiv:2601.23088.
8. Eyuboglu et al., Cartridges. arXiv:2506.06266. Cartridges at Scale: arXiv:2606.04557.
9. Xie et al., SemPIC. arXiv:2607.28069; code at github.com/jn12-29/SemPIC (MIT), commit 2aae1b1a.
10. C²KV. arXiv:2607.17715; KDD 2026.
11. Doc-to-Atom. arXiv:2606.12400. KV Packet: arXiv:2604.13226.
12. "Forgetting Without Restarting." arXiv:2609.04875.
13. Leyline. arXiv:2606.01065.
14. vLLM cache salting: RFC #16016, PR #17045, release v0.9.0 (May 15, 2025). KVGov: arXiv:2608.09225. Also cited in Figures 8 and 9: SafeKV arXiv:2508.08438; CachePrune arXiv:2605.23640; PrefixShield arXiv:2608.01657; Fail-Closed Lowering of Resident KV Claims arXiv:2606.01387; stale-cache repair arXiv:2609.17983; KVShare arXiv:2503.16525 (current version). WorldFlow's open engine proposals for semantic reuse: SGLang PR #31057, vLLM RFC #44223, TensorRT-LLM RFC #14918.
15. "Self-Study Reconsidered: The Hidden Fragility of Learning from Self-Generated QA." arXiv:2606.32002.
16. Zhao and Mastorakis, SemShareKV. arXiv:2509.24832; Findings of IJCNLP-AACL 2025.
17. WorldFlow AI, ["Reuse, Don't Recompute."](#) September 3, 2026.

Methods note

System under test. SemPIC at the commit above. Base model Qwen/Qwen3-4B-Instruct-2507 at a pinned revision. The writer uses a rank 8 LoRA, a small trained addition to the model. It was trained with the authors' settings on 1,024 entries from the biography training data and the six types of facts listed above. The model chose its highest scoring next token at each step, called greedy decoding. Answers were limited to 32 new tokens.

Scoring and repeatability. An answer counted when the exact invented value appeared as a whole word in the output. A separate scorer that searched for the value as a string produced the same counts. Each run used one NVIDIA A10G. The input generation seeds were 20260918 for runs 1 and 2, and 20260919 for run 3. These numbers allow the same test inputs to be generated again. The two sets share no person name or invented value.