54Humans

Self-Hosted Memory for Coding Agents

Akhil Katakam
Koragraph
katakamakhil0@gmail.com

September 2026

Abstract

Coding agents routinely receive project conventions, hazards and procedures in the course of ordinary conversation, and routinely lose them when the session ends. Current agents address this by letting the agent maintain its own memory, but the decision of what to record competes with the task the agent is executing. We present 54Humans, a 1.7B-parameter model that runs locally alongside a coding agent and is the sole writer of the agent's persistent memory. 54Humans observes each completed turn, decides whether it contains information that would otherwise have to be restated, extracts that information in the user's own words, and writes it to a local store that delivers it to subsequent sessions. We evaluate the approach in a controlled cross-session benchmark on two codebases, in which a rule is stated once in passing and later sessions are scored deterministically on whether the code the agent writes respects it. With Claude Code as the agent, rule compliance in later sessions is 29.6% without memory, 63.0% with the agent's built-in memory, and 94.5% with 54Humans ($p < 10^{-4}$ against built-in memory, Fisher's exact test). When 54Humans' entries are delivered through the agent's own built-in memory instead, compliance is 87.0%, which places most of the gain in what is recorded rather than in how it is delivered. Rules that reached a later session were followed in 95.0% of cases, compared with 28.4% for rules that did not, which indicates that compliance in this setting is limited almost entirely by whether information is recorded and delivered rather than by the agent's willingness to follow it.

1 Introduction

Large language model agents now carry out substantial software engineering work, from resolving repository-level issues [6] to multi-hour interactive sessions with a developer. In the interactive setting, a large share of what the developer communicates is not a task but a constraint: a convention the team follows, a failure that must not recur, a step that accompanies a certain kind of change. Such constraints are rarely recoverable from the code itself. Within a session the agent can follow them. Across sessions it cannot, unless they are written somewhere the next session will read.

Existing systems place this responsibility on the agent. MemGPT [8] exposes memory management to the model through function calls, and reflective agents [9, 12] summarise their own experience into a persistent store. Commercial coding agents follow the same pattern: Claude Code maintains a memory directory it can write to and reads instruction files such as `CLAUDE.md` and `AGENTS.md` that it can also edit. In each case the model performing the task is also the model deciding what to remember. We observe that this coupling has a cost. An agent engaged in a task records only a fraction of the constraints it is given, and appears to record them selectively, favouring statements that are explicitly framed as rules over those mentioned in passing.

A concrete case from our benchmark illustrates the problem. While asking Claude Code for a small retry helper, a developer adds: "Also, fyi: we're moving off axios. Don't use it in anything new, built-in fetch only from now on." The agent writes the helper correctly. In a later session the same developer asks for a Slack notifier, and the agent's first line of code is `const axios = require('axios');`. The agent had saved two other conventions from that project to its memory, and it followed the one that applied. It had not saved this one.

This paper continues an earlier argument [7]. There we proposed that project knowledge is lost at both ends by a single mechanism, the authorship floor: recording something is a commitment to keep it true by hand, so developers record only what seems to justify the expense, and what falls below that threshold leaves with the session. We described a memory layer that makes forgetting mechanical by binding each item to the code it is about, argued that this is what makes cheap capture safe, and closed by naming the experiment the paper did not run: tasks selected for friction, a second encounter with the same constraint, and a measure of whether the agent's work changes. This paper runs that experiment, with the remaining half of the argument, capture without authorship, delegated to a model.

We examine the alternative of separating the two roles. 54Humans[1] is a small language model that runs outside the agent's control loop and holds exclusive write access to its memory. After each turn it determines whether the turn contains anything that, if forgotten, would force the user to repeat themselves. On most turns it records nothing. When it does record something, it copies the relevant statement from the user's message, classifies it, and writes it to a local store that injects it into future sessions. The agent retains read access and loses the ability to write. Our contributions are as follows.

1. We describe an architecture in which a separate local model is the only writer of a coding agent's persistent memory. The architecture is event driven, adds under 50 ms to the agent's turns, and uses constant memory in the number of concurrent sessions.
2. We train 54Humans, a 1.7B-parameter model fine tuned on real and synthetic coding-agent transcripts to abstain on the approximately 90% of turns that contain nothing worth retaining, and to extract rather than paraphrase when it does record.
3. We introduce a cross-session benchmark in which constraints are stated once during unrelated work and later sessions are scored by deterministic checks on the code the agent produces.
4. We show that 54Humans raises cross-session compliance from 63.0% with the agent's built-in memory to 94.5%, that most of the gain remains (87.0%) when its entries are delivered through the agent's own memory, and that across all conditions compliance is determined largely by whether a constraint is delivered to the later session.

2 Problem Formulation

A user interacts with an agent over a sequence of sessions $s_1, s_2, \ldots, s_T$ on a single repository. Each session begins without the conversational context of earlier sessions. During some session s_i the user states a constraint r, typically while requesting unrelated work. In a later session s_j, $j > i$, the agent performs a task whose correct execution depends on r. We say the task *exercises* r if the agent's changes touch the behaviour r governs, and that the agent *complies* with r if those changes respect it.

A *memory writer* W observes session s_i and may produce a set of memory entries. A *delivery mechanism* D determines which stored entries appear in the context of s_j. Compliance can fail at three points: W

[1]The name records the number of attempts it took: the model described here is the fifty-fourth.

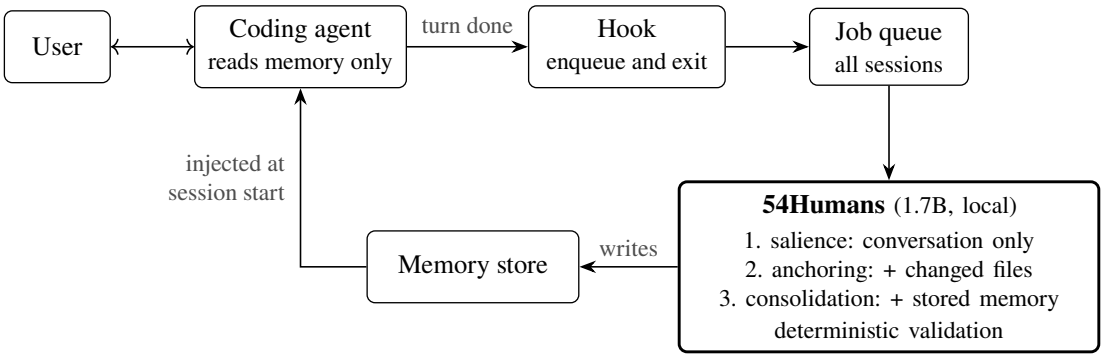

Figure 1: Architecture. The agent retains read access to memory; 54Humans is its only writer. The hook never loads a model, so the user never waits on it, and a single resident process serves every session.

may not record r, D may not deliver it, and the agent may disregard it once delivered. Our primary metric is the *compliance rate*, the fraction of exercised (task, constraint) pairs in which the agent complies. We additionally report the *delivery rate*, the fraction of exercised pairs in which r was present in the context of s_j.

We restrict attention to constraints that cannot be inferred from the repository. An item is worth retaining if and only if forgetting it would force the user to restate it or force the agent to rediscover it at cost. We distinguish five kinds: *laws* (standing rules), *hazards* (past failures to avoid), *rituals* (procedures bound to a trigger), *corrections* (changes of direction), and *open loops* (work deliberately deferred).

3 System Design

3.1 Overview

Figure 1 shows the architecture. The agent runs unmodified except that it has no means of writing to memory. A lightweight hook fires on session events and enqueues the completed turn. A single resident process holds the 54Humans model, drains the queue, and writes accepted entries to the memory store. At the start of each new session the store injects repository-wide entries into the agent's context.

3.2 Triggering

A turn is judged when the user sends the following message. At that point the agent's reply is complete, and the new message frequently carries information about the previous one, such as a correction or an acknowledgement. An additional trigger at session end handles the final turn. The hook extracts only the visible text of the turn, discarding tool invocations and their output, which in our transcript corpus account for the majority of each turn's bytes. It then enqueues a job and exits without loading a model, so the user never waits on it.

A single long-running process owns the model and consumes the queue in priority order, one inference at a time. Because the model is loaded once and shared, memory consumption is independent of the number of active sessions. Jobs are deduplicated by turn and session, and a per-session watermark ensures each turn is judged exactly once.

3.3 Input

Each judgment operates on a bounded packet containing the user's message and the agent's reply, the set of files modified during the turn, and the entries already stored for the repository. Modified files are taken from the memory store's own event log rather than from the agent's account of its work.

3.4 Judgment

We decompose each judgment into three inferences, each given only the information relevant to its question. The first, *salience*, decides whether the turn contains anything worth retaining, and is given the conversation alone, since whether a statement constitutes a standing constraint is a property of what the user said rather than of the files that changed or the memories already stored. The second, *anchoring*, runs only when the salience pass produced an entry and the turn modified code. It sees the modified files and may attach an entry to a specific file or declaration, but it cannot add, remove or rephrase entries. The third, *consolidation*, runs only when memories already exist, sees the full packet, and may update or retire an existing entry. On turns where the salience pass abstains, which is the common case, the judgment is a single inference.

3.5 Deterministic Validation

Every judgment passes through a set of checks that involve no model. An anchor must refer to a file that exists or was modified in the turn, or to a declaration named in the packet, and must be named in the entry itself; otherwise the entry is stored at repository scope. In the training labels, 91% of non-repository anchors satisfy the naming condition. At least 30% of an entry's content words must occur in the session text, a threshold that rejects 0.2% of correct entries in our labelled data while excluding entries whose content originates elsewhere, including restatements of stored memories. Near duplicates of existing entries and entries with a kind outside the fixed vocabulary are discarded. An entry is attributed to the user only if the user stated it; entries inferred by the agent are attributed to the agent and treated as lower confidence. If the store cannot resolve an entry's anchor at write time, the entry is recorded again at repository scope.

3.6 Isolation

54Humans reads the store's event log in read-only mode, maintains its own state in a separate database, and writes through the store's standard interface. The component is additive: disabling it leaves all previously written memory readable, and the agent's behaviour degrades to that of an agent without new memory.

4 Model and Training Data

4.1 Transcript Corpus

We collected 2,132 Claude Code session transcripts from routine development work. Of the messages attributed to the user, approximately 21% were not written by the user but were injected by tooling, including benchmark task descriptions, harness notifications and command wrappers. After removing these, 5,810 user turns remained. Because an entry attributed to the user carries the user's authority, correct attribution was treated as a first-class requirement of the data pipeline.

The resulting distribution is highly skewed. Approximately 90% of turns contain nothing worth re-

taining. User messages are bimodal in length, consisting either of short conversational turns or of long turns dominated by pasted logs and plans. Positive turns are frequently marked by the user ("remember that", "for now", "going forward"), which makes extraction feasible for a small model.

4.2 Training Set

The training set contains 2,835 labelled turns from the corpus and 1,566 synthetic turns designed to cover behaviour that real data exhibits rarely: multiple entries in a single turn, constraints owned by a single declaration, repository-wide constraints, and turns that resemble constraints but are scoped to the current task. Synthetic user messages span clean, casual and noisy registers, with register sampled independently of the label so that writing style carries no information about the target. Approximately 72% of capture examples are labelled as containing nothing to retain.

4.3 Output Format

The model emits one line per entry or the token `NONE`. Each entry specifies a kind, a source (user or agent), an anchor, and a body, for example:

```
MEMORY | kind=law | source=HUMAN | anchor=general | Env vars get read in exactly one place,
    config/env.js.
```

The body is copied from the user's statement rather than composed. This choice matches the relative strengths of a small model, which is considerably more reliable at locating and extracting a span than at generating a faithful paraphrase, and it prevents the model from attributing to the user content the user did not provide.

4.4 Training

We fine tune Qwen3-1.7B [10] with LoRA [4] (rank 16, $\alpha = 32$) for two epochs on a single NVIDIA T4, computing the loss only on the target. The model is used in its non-thinking mode, and inference reproduces the training template exactly. The quantised model (Q4_K_M) occupies 1.3 GB.

5 Offline Evaluation

Table 1 reports performance on 260 held-out capture examples, evaluated with the full judgment pipeline of Section 3.4 and its deterministic checks. Field-level accuracies are computed over examples in which the predicted and reference entry counts agree. The held-out examples come from the same transcript corpus and synthetic process as the training set, so these figures describe in-distribution performance; the cross-session benchmark of Section 6 is the more demanding test. Table 2 summarises runtime cost on a laptop-class machine. Because the hook only enqueues, its latency is the only overhead visible to the user.

6 Cross-Session Benchmark

6.1 Codebases and Constraints

We use two codebases in different languages: a production Node.js backend service of approximately 2,400 files, and a widely used open-source Python web framework of approximately 110 source files. For

Measure	Result
Abstention, nothing to retain	99.4% (160/161)
Recall, something to retain	91.9% (91/99)
Kind accuracy	94.4%
Source accuracy	100.0%
Anchor accuracy	92.1%

Table 1: Held-out capture performance.

Measure	Result
Hook latency, median / p90	47 ms / 51 ms
Inference per judgment, median	0.49 s
Throughput, 1/4/16 sessions	1.4/1.9/2.0 turns/s
Resident memory	approx. 2.2 GB

Table 2: Runtime cost of a single resident process.

each we wrote five constraints that cannot be inferred from the code: in total four laws, two hazards, two corrections and two rituals. Each constraint deliberately contradicts the prevailing pattern in its codebase. In the Node.js service, for example, environment variables are read directly in 111 files and console logging appears at 393 call sites; in the Python framework one path library is used 35 times for every 3 uses of the alternative. An agent without memory that imitates surrounding code will therefore tend to violate the constraint. We regard this as the situation in which memory is most consequential rather than as a representative sample of all constraints.

6.2 Protocol

Each constraint is stated in a *first session*: a real agent session in which the user requests a small unrelated change and mentions the constraint in passing. Phrasing varies across constraints in ways typical of developer speech ("one standing rule going forward", "fyi, we're moving off", "heads up, this bit us before", "remember this for the future"). Two additional first sessions contain directive language but no standing constraint, and serve to detect spurious recording.

Second sessions are fifteen tasks, ten on the Node.js service and five on the Python framework, each exercising two to four constraints without mentioning them. For example, implementing a client for an external API whose timeout is configured through an environment variable exercises the constraints on environment access, HTTP clients and logging.

The agent is Claude Code with Claude Sonnet 5 at medium effort, run non-interactively with a limit of 40 turns. Each session starts from a fresh copy of the repository. Dependency installation and web access are disabled.

6.3 Conditions

We compare four conditions, which differ only in who writes memory and how it is delivered.

No memory. Each second session begins without access to anything recorded earlier.

Built-in memory. Unmodified Claude Code. All sessions run at the same project path, so the agent's memory directory persists between sessions, and any instruction files the agent writes (`CLAUDE.md`, `AGENTS.md`, or files under `.claude`) are carried into subsequent sessions as they would be in a working repository.

54Humans. The agent has no means of writing memory. After each first session, 54Humans judges the transcript and writes to the memory store, which injects repository-scope entries at the start of each

Condition	Compliance	95% CI	Sessions fully compliant
No memory	29.6% (16/54)	[19.1, 42.8]	0/18
Built-in memory	63.0% (29/46)	[48.6, 75.5]	3/15
54Humans via built-in memory	87.0% (40/46)	[74.3, 93.9]	9/15
54Humans	94.5% (52/55)	[85.1, 98.1]	16/18

Table 3: Cross-session compliance. Counts are exercised (task, constraint) pairs.

second session, up to eight entries within 2,000 characters. The agent's built-in memory is not carried between sessions in this condition, so its only persistent knowledge is what 54Humans recorded.

54Humans via built-in memory. This condition separates the writer from the delivery channel. The entries 54Humans wrote from the same first sessions are placed, verbatim, in the agent's built-in memory directory, in the format the agent uses for its own entries: an index file listing each entry and one note per entry. Second sessions then run as in the built-in memory condition, with no memory store and no injection. The agent therefore receives 54Humans' entries through exactly the channel it uses for its own.

6.4 Scoring and Analysis

A script examines the lines added by the agent in each second session. A constraint is scored only if the task exercised it, determined by the presence of the relevant behaviour in the added code (for example, a read of an environment variable); it is then scored as complied with or violated. No language model is involved in scoring. Every violation was additionally checked by hand.

Observations are (task, constraint) pairs. Because pairs within a session are not independent, we report Wilson 95% confidence intervals and Fisher's exact test at the pair level together with two analyses at coarser granularity: the proportion of sessions in which every exercised constraint was respected, and a paired comparison over tasks. The four conditions comprise 90 agent sessions (median 11 turns) at a total cost of USD 23.65. Three second-session tasks were run twice in the no-memory and 54Humans conditions to estimate run-to-run variation.

7 Results

7.1 Compliance

Figure 2 and Table 3 report compliance by condition. Built-in memory roughly doubles compliance relative to no memory (29.6% to 63.0%, $p = 0.0012$), and 54Humans raises it further to 94.5% ($p = 9.5\times10^{-5}$ against built-in memory). The same ordering holds at the session level: every exercised constraint was respected in 16 of 18 second sessions under 54Humans, 9 of 15 under 54Humans via built-in memory, 3 of 15 under built-in memory, and none of 18 without memory. In a paired comparison over the fifteen tasks, 54Humans achieved higher compliance than built-in memory on 10 tasks, equal compliance on 5, and lower on none (two-sided sign test, $p = 0.002$). In the repeated runs, all 19 constraint outcomes matched the first run.

One constraint, requiring use of a shared logger, was respected in 10 of 11 cases even without memory, because code surrounding the relevant call sites already used the logger. Excluding it, compliance is 14.0% (6/43) without memory, 52.8% (19/36) with built-in memory, 83.3% (30/36) with 54Humans via built-in

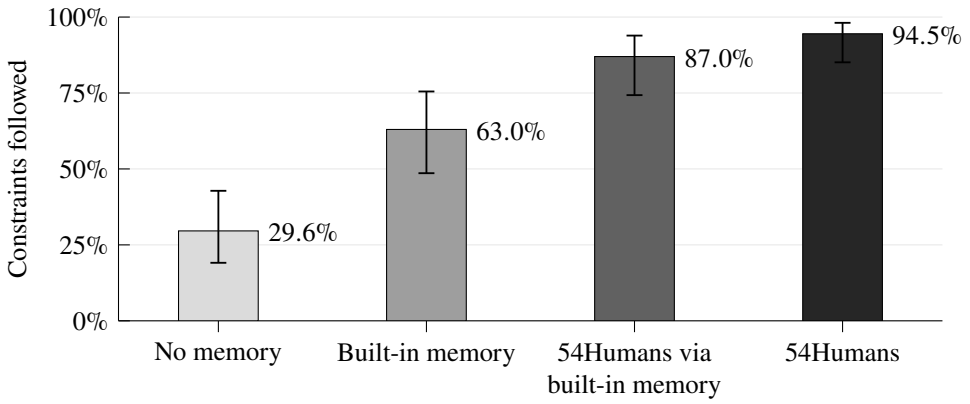

Figure 2: Share of exercised constraints followed in a later session, pooled over both codebases. Whiskers show Wilson 95% intervals: no memory 16/54 [19.1, 42.8]; built-in memory 29/46 [48.6, 75.5]; 54Humans via built-in memory 40/46 [74.3, 93.9]; 54Humans 52/55 [85.1, 98.1].

memory, and 92.9% (39/42) with 54Humans.

7.2 Writer or Channel?

54Humans differs from the built-in memory condition in two respects: who writes the entries, and how they reach the next session. The fourth condition holds the channel fixed. With 54Humans' entries delivered through the agent's own memory directory, compliance is 87.0% (40/46), against 63.0% when the agent writes that directory itself ($p = 0.015$). Over the fifteen tasks, 54Humans' entries produced higher compliance than the agent's own on 9, equal on 6, and lower on none (two-sided sign test, $p = 0.004$). Moving the same entries to the memory store raises compliance further to 94.5%, a difference that is not significant at this sample size ($p = 0.29$). Most of the effect therefore lies in what is recorded.

The remaining gap is concentrated in a single constraint. Four of the six violations in this condition concern the rule on environment variables, whose entry reads as a description ("Env vars get read in exactly one place, config/env.js.") rather than an instruction. Presented as one note among the agent's memories it was read and not applied; presented by the memory store, under a header stating that the entries are rules given by the developer, it was followed in all nine sessions that exercised it (Section 8.2).

7.3 An Example

Figure 3 follows one task through three of the conditions. In an earlier session the developer stated two constraints relevant to it: that new code must use the built-in fetch rather than axios, introduced with "fyi", and that environment variables are read only in a single configuration module. The task asks for a notifier that posts to a Slack webhook whose URL is given by an environment variable.

Without memory the agent reproduced the prevailing patterns of the codebase. With its built-in memory it followed the one relevant convention it had recorded, routing its messages through the shared logger, and violated the two it had not. With 54Humans every relevant constraint was present at the start of the session, and the agent added the variable to the configuration module, imported it, and used fetch. The pattern is general, as Section 7.5 shows: the agent follows what it is given, and the conditions differ in what it is given.

First session (developer, while asking for a retry helper): *"Also, fyi: we're moving off axios. Don't use it in anything new, built-in fetch only from now on."*

Condition	What reached the second session	What the agent wrote (`slack-notifier.js`)

No memory	nothing	`const axios = require('axios');`
		`const webhookUrl = process.env.SLACK_WEBHOOK_URL;`
		`await axios.post(webhookUrl, { text }, { timeout: 5000 });`

Built-in memory	the agent's own notes: use the shared logger; update the endpoint list	`const axios = require('axios');`
		`const webhookUrl = process.env.SLACK_WEBHOOK_URL;`
		`logger.warn('slack-notifier.notify: ... not configured');`
		`await axios.post(webhookUrl, { text });`

54Humans	all five stored constraints, including *"We are moving off axios; don't use it in anything new, built-in fetch only from now on"* and *"Env vars get read in exactly one place, config/env.js"*	`const { SLACK_WEBHOOK_URL } = require('../config/env');`
		`const res = await fetch(SLACK_WEBHOOK_URL, { ... });`
		`logger.error('Slack notify failed', { status: res.status });`

Figure 3: One task under the three conditions. Code excerpts are taken verbatim from the agent's changes, with long arguments elided.

7.4 Compliance by Kind of Constraint

Figure 4 breaks results down by kind. Rituals are the most dependent on memory: without it they were never performed, since nothing in the code indicates that a given change entails an additional step. All three memory conditions performed well on rituals, which reflects that both writers recorded them (Section 7.6). The largest differences between the writers arise for laws, hazards and corrections.

7.5 The Role of Delivery

For each exercised pair we determined from the session transcript whether the constraint was present in the agent's context, whether through the memory store's injection, the agent's memory directory, or an instruction file. Pooled over the four conditions, delivered constraints were followed in 95.0% of cases (114/120, 95% CI [89.5, 97.7]) and undelivered constraints in 28.4% (23/81, [19.7, 39.0]); Fisher's exact test gives $p = 1.4 \times 10^{-24}$.

Once delivered, a constraint was almost always respected. The undelivered constraints that were nonetheless respected were predominantly cases in which the task itself favoured compliant code. We interpret this as evidence that, for a current frontier agent, the dominant failure in cross-session compliance is not disregard of stated constraints but their absence from context. Figure 5 makes the relationship visible: in each condition the compliance rate sits close to the delivery rate: 0% of constraints were delivered without memory, 50% with built-in memory, and 96% in both 54Humans conditions.

7.6 What Each Writer Recorded

Table 4 compares the entries produced by the two memory writers from the twelve first sessions, and Figure 6 shows the individual constraints.

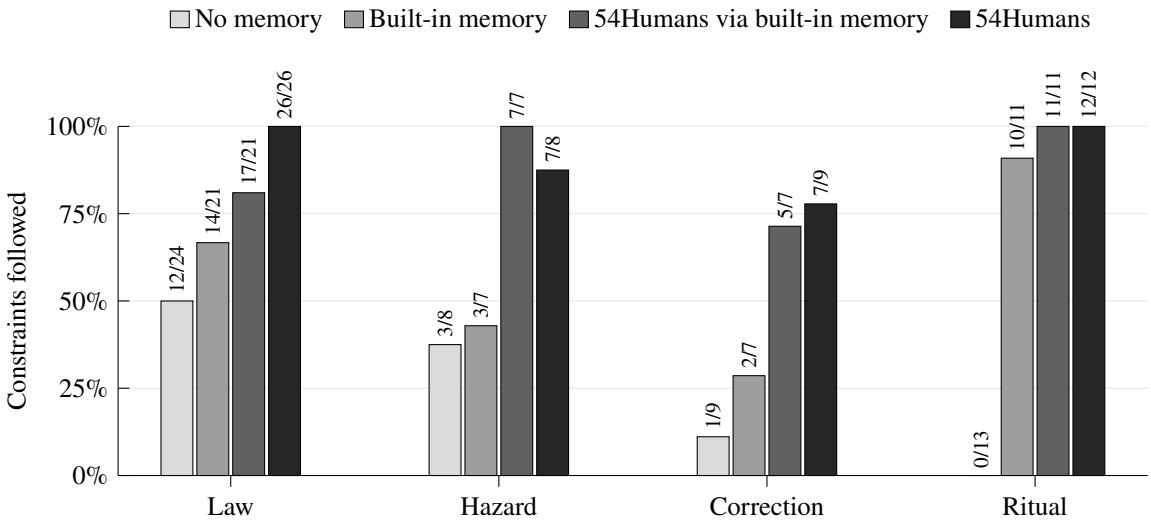

Figure 4: Compliance by kind of constraint. Labels give followed / exercised pairs.

	54Humans	Built-in memory
Constraints recorded	9/10	4/10
Constraints delivered to later sessions	9/10	4/10
Entries recorded from the two noise sessions	0	0
Entries not corresponding to any constraint	1	0

Table 4: Entries written from the twelve first sessions.

The agent's own entries were of high quality. Each restated the constraint, noted that the user had requested it, and described how to apply it, and the four were followed in 22 of the 23 later cases that exercised them. The difference between the conditions lies in coverage rather than in the quality of individual entries. 54Humans produced one entry that did not correspond to a constraint, derived from a task description phrased as a procedure; it was anchored to a single file and was not injected into any later session.

8 Discussion: Where Memory Is Lost

The results above describe the complete systems. In this section we examine why constraints failed to reach later sessions or, once delivered, were not followed.

8.1 Selective Recording by the Agent

All four constraints the agent recorded on its own were phrased explicitly as durable rules ("a standing rule for this codebase", "remember this for the future"). Of the six it did not record, five were phrased incidentally ("fyi, we're moving off", "heads up, this bit us before", "also, going forward"); the sixth was introduced as "one standing rule going forward" but embedded in the description of the task (Figure 6). This suggests that an agent engaged in a task records a constraint when the user clearly signals that it should be recorded and the statement stands apart from the work, and otherwise treats it as part of the

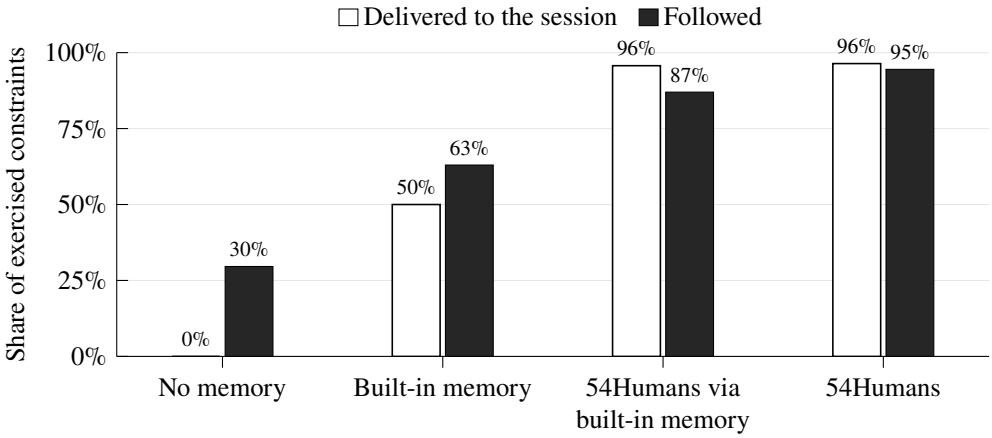

Figure 5: Delivery rate and compliance rate per condition. Compliance tracks delivery; the residual without delivery comes from tasks that happened to favour compliant code.

current task. 54Humans, whose only function is to make this judgment, recorded five of these six.

We also evaluated a variant in which the agent was given an explicit tool for writing to the same memory store used by 54Humans. The tool's description instructs the agent to call it whenever the developer states a rule, corrects it, or asks it to remember something. The variant repeated the full protocol, twelve first sessions and fifteen second sessions, with the tool available throughout and the store delivering entries at session start exactly as in the 54Humans condition. Compliance was 46.7% (21/45). When the agent used the tool, it frequently anchored an entry to a directory or to a file that did not yet exist; the store correctly declined to treat such entries as verified and did not deliver them, while reporting the write as successful to the agent.

8.2 Delivered but Not Followed

Six delivered constraints were violated across all conditions. In the 54Humans condition the agent raised a bare `RuntimeError` despite a delivered prohibition. In the built-in memory condition it once skipped the endpoint-list ritual it had itself recorded. The remaining four were the environment-variable rule in the 54Humans via built-in memory condition, discussed in Section 7.2. The contrast with the memory store, where the identical entry was always followed, suggests that the framing of delivered memory matters for entries phrased as descriptions: an agent that is told a list of entries are instructions from the developer treats them as such, while an agent that finds the same sentence among its own notes may treat it as context. Recording entries in imperative form would make them robust to either channel.

9 Limits and Open Questions

What follows is where the evidence stops: the places where the design or the experiment is narrower than the claims it might seem to support, and the questions we would want answered before claiming more.

Ten constraints, written by us. Each codebase received five constraints of our own choosing, and each was chosen to contradict the prevailing pattern in the code. This isolates the contribution of memory, since an agent without memory will reliably violate such a constraint, but it also means the absolute

Constraint (kind)	How it was introduced	Built-in memory	54Humans
Shared logger only (law)	*"standing rule for this codebase"*	◉	●
Endpoint list ritual (ritual)	*"remember this for the future"*	◉	●
pathlib only (law)	*"a standing rule for this codebase"*	◉	●
Changelog entry (ritual)	*"remember this for the future"*	◉	●
Config module for env (law)	*"one standing rule going forward"*	○	●
No axios (correction)	*"fyi, we're moving off"*	○	●
No sync fs calls (hazard)	*"heads up on a hazard that bit us"*	○	●
No bare RuntimeError (hazard)	*"heads up, this bit us before"*	○	●
UTC timestamps (law)	*"also going forward"*	○	●
JSON provider only (correction)	*"fyi we moved away from"*	○	○
		4/10	9/10

Figure 6: Which constraints each writer recorded. A filled circle means the constraint was recorded and delivered to later sessions; an open circle means it was not. The agent recorded only the four constraints introduced as explicit rules (above the dashed line).

figures describe that situation rather than the typical constraint a developer states. A constraint that agrees with the surrounding code would be followed without memory, and a benchmark built from those would show a smaller effect. Whether the gap between writers persists on constraints drawn from real sessions rather than written for the experiment is the first thing we would test next.

Small numbers. The four conditions rest on ninety sessions and fifteen tasks. The effects are large and survive the session-level and task-paired analyses, but the intervals are wide, and the difference between the two 54Humans conditions cannot be resolved at this size. The ordering of the conditions is well supported; the precise magnitudes are not.

One agent. Every session used Claude Code with a single model at a single effort level. The results for built-in memory describe how that agent used its memory at the time of the experiments, and a different agent, or the same agent with a different memory policy, might record more. The finding that delivered constraints are followed is likely to hold more widely, since it concerns instruction following rather than memory; the finding that agents record too little is specific to what we measured.

A single interval. The benchmark measures one gap between a first session and a later one. It does not test a store that grows over months, entries that become stale as the code changes, or the cost of delivering many entries at once, which prior work suggests is real [5]. The memory layer retires entries whose subject disappears [7], but how a store written by 54Humans behaves at that scale is an open question and, we think, the more important one.

Scoring by pattern. Compliance is determined by pattern checks over the lines an agent added, and delivery by inspecting transcripts for the constraint's text. Every violation was verified by hand, and one error in the delivery check was found and corrected in that process, but an unusual compliant idiom could still be misclassified, and delivery through a channel we did not anticipate would be missed.

10 Related Work

Memory for language model agents. MemGPT [8] treats the context window as a managed memory hierarchy controlled by the model. Generative Agents [9] maintain a memory stream with periodic reflection, and Reflexion [12] stores verbal lessons from failed attempts. Mem0 [2] and A-MEM [13] provide memory layers that extract and organise information from conversations, generally using a large model for extraction. Zep [11] maintains a temporal knowledge graph and uses a model to decide when a new fact supersedes an old one, and EA-Graph [3] anchors a coding agent's verification claims to the artifacts that established them. In these systems the decision of what to store is made by the acting model or by a large auxiliary model. 54Humans assigns this decision to a dedicated small model and removes write access from the acting agent.

Why accumulation is not free. Delivering stored material to an agent has a cost that grows with the store. Jaroslawicz et al. [5] found instruction-following degrading as the number of instructions rises, and Chakrabarti [1] observed agent instruction files more than tripling in length over their lifetime while their oldest entries became the hardest to remove. Both bear on the design choices here: 54Humans records rarely, extracts rather than elaborates, and relies on the memory layer's anchoring to retire entries whose subject is gone.

Instructions for coding agents. Instruction files such as `CLAUDE.md` and `AGENTS.md`, and the memory directory maintained by Claude Code, are the prevailing mechanism for persisting project knowledge. Our results suggest that these mechanisms are effective when populated and that population, rather than consumption, is the limiting step.

Small models. 54Humans is fine tuned with parameter-efficient methods [4] on a compact base model [10]. The broader position that inexpensive capture requires mechanical handling of forgetting follows our earlier argument [7].

11 Conclusion

We examined how a coding agent's knowledge of project constraints can persist across sessions, and argued that the task of deciding what to record should be separated from the task the agent is performing. 54Humans, a 1.7B-parameter local model with exclusive write access to the agent's memory, recorded nine of ten constraints stated during unrelated work, recorded nothing from turns that did not warrant it, and raised compliance in later sessions from 63.0% with the agent's built-in memory to 94.5%. Delivered through the agent's own memory, the same entries still reached 87.0%. In the terms of our earlier work, 54Humans lowers the authorship floor from the capture side: what a developer says in passing is recorded without anyone deciding to author it. Across conditions, compliance tracked delivery closely, which suggests that further gains in this setting will come from improving what is recorded and how it is delivered. Directions for future work include relevance-based delivery for large stores, training data drawn from many users, and evaluation over longer horizons in which stored constraints change or become obsolete.

References

[1] K. Chakrabarti. Why Does CLAUDE.md Keep Growing? Catastrophic Remembering in Agentic Coding. arXiv preprint arXiv:2608.11095, 2026.

[2] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv preprint arXiv:2504.19413, 2025.

[3] H.-J. Hsu, C.-J. Chi, and H. Everett. EA-Graph: Artifact-Anchored Verification Memory for Coding Agents under Upstream Drift. arXiv preprint arXiv:2608.04278, 2026.

[4] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR)*, 2022.

[5] D. Jaroslawicz, B. Whiting, P. Shah, and K. Maamari. How Many Instructions Can LLMs Follow at Once? arXiv preprint arXiv:2507.11538, 2025.

[6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations (ICLR)*, 2024.

[7] A. Katakam. The Authorship Floor: Why Project Knowledge Is Lost at Both Ends, and What Anchoring It to Code Recovers. 2026.

[8] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. arXiv preprint arXiv:2310.08560, 2023.

[9] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 2:1-2:22, 2023.

[10] Qwen Team. Qwen3 Technical Report. arXiv preprint arXiv:2505.09388, 2025.

[11] P. Rasmussen, P. Paliychuk, T. Beauvais, J. Ryan, and D. Chalef. Zep: A Temporal Knowledge Graph Architecture for Agent Memory. arXiv preprint arXiv:2501.13956, 2025.

[12] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[13] W. Xu, K. Mei, H. Gao, J. Tan, Z. Liang, and Y. Zhang. A-MEM: Agentic Memory for LLM Agents. arXiv preprint arXiv:2502.12110, 2025.