SCROLL: PROGRAMMATIC CONTEXT MANAGEMENT FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents for complex software engineering and deep research accumulate actions, observations, and intermediate state far beyond a model's context window. Context management must therefore efficiently construct compact, task-relevant views of this growing history for each model call, even though future information needs are unknown. Existing methods typically either bound context growth through structured but lossy compaction and extraction, or externalize the full history as raw, unstructured context for model-written programs to navigate.

We present *Scroll*, a lossless and programmable context manager for long-running agent sessions. An append-only event log retains history losslessly under stable addresses and timestamps, while a tiered eviction index maintains in-context address-based anchors to off-context session history. Model-written programs access history through structured search and recovery operations, compute over it in a sandboxed persistent Python kernel, and expose only selected results. With Qwen3.8-Max, Scroll scores **94.8** on LongMemEval$_S$, **75.5** on BEAM$_{10M}$, and **93.3** on LOCA$_{256K}$, exceeding the best published scores by **7.5** points on BEAM$_{10M}$ and **44.0** points on LOCA$_{256K}$. Ablations on BEAM$_{10M}$ show that lossless retention, the persistent Python kernel, and the structured access layer make complementary contributions to accurate and token-efficient context construction.

1 INTRODUCTION

LLM agents increasingly tackle long-running tasks such as complex software development (Jimenez et al., 2024; Yang et al., 2024) and open-web deep research (Zheng et al., 2025; Wei et al., 2025). These tasks produce growing histories of model calls, tool actions, observations, and intermediate results, yet each model invocation operates within a bounded context window. The agent harness must therefore decide which history to expose on each call. This challenge arises well before the model's nominal context limit, because retrieval and reasoning degrade as inputs grow (Modarressi et al., 2025; Zeng et al., 2026).

Existing context-management mechanisms tackle this problem either by compacting growing history into structured but lossy representations through compression and extraction (Kang et al., 2025; Ye et al., 2025; Packer et al., 2023; Wu et al., 2025; Cao et al., 2026), or by exposing the full history as raw context for flexible model-written programs (Zhang et al., 2025). Both substantially extend agent horizons. However, the former can irreversibly omit details needed by subsequent tasks (Ye et al., 2025), while the latter provides little built-in structure for finding relevant events, increasing token use and the risk of retrieval or execution failure (Wang, 2026).

We present *Scroll*, a context manager that combines lossless retention with a structured and programmable *Session Environment*. The environment comprises an append-only event log, durable payload storage, and a persistent computational state backed by a sandboxed, persistent Python kernel. The event log records session events under stable addresses, together with metadata identifying their source, timestamp, and session. Durable payload storage retains large tool outputs and generated artifacts through lazily materialized references, preserving the complete session history without storing all payloads inline. The persistent computational state retains materialized records and derived values as Python objects across model calls.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

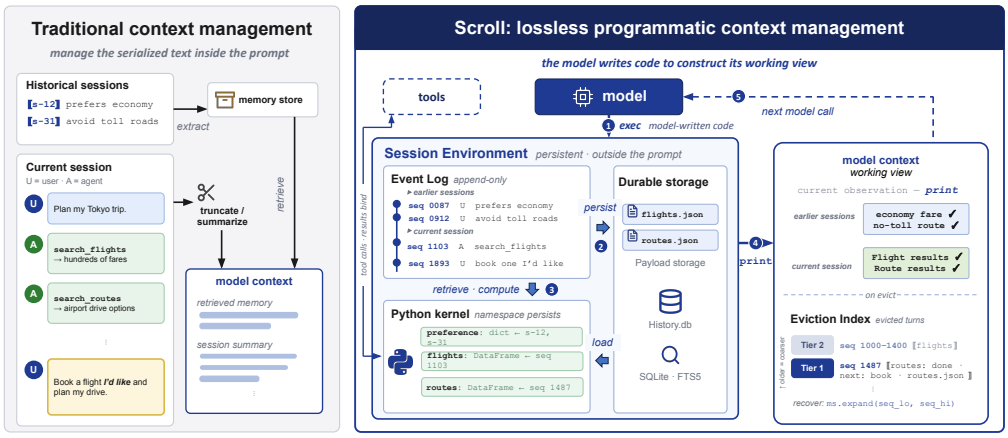

Figure 1: **System overview.** Scroll keeps the full session outside the prompt; model-written code selects what enters each working view.

Scroll separates system-provided structure from model-directed context construction: the substrate fixes how session state is stored, addressed, searched, and recovered, while query-time code decides what to retrieve, compute, and expose. The model invokes the kernel through `exec`, using `search` and `expand` to locate and materialize records and ordinary Python to filter, combine, and compute over them. Only explicitly printed results enter the model's working context.

To keep the model aware of history outside its working context, Scroll pins a tiered eviction index in every model call. The index provides a compact map of evicted spans whose entries retain stable address anchors into the event log. It guides the model to relevant off-context regions, from which the original records can be materialized on demand.

Our contributions are threefold:

- We formulate context management as the programmatic construction of bounded working views over a structured and programmable session environment; and introduce Scroll, the context manager that combines system-provided structure with model-directed computation.
- We evaluate Scroll across three long-context benchmarks spanning memory retrieval and interactive tool use, with histories up to 10M tokens and five backbone models. Scroll matches or exceeds strong existing systems while exposing only a small fraction of the underlying history to the model.
- Through controlled ablations on BEAM_{10M}, we show that lossless storage, programmable computation, and structured access are complementary. Comparisons against lossy summarization, together with persistent-runtime and eviction-index ablations, isolate the role of each component.

2 Programmatic Context Management

2.1 Problem Formulation

Session state and working view. An agent session produces a growing sequence of *events* $e_1, e_2, \ldots$ (user messages, model responses, tool calls, tool results), each with an associated *payload* (its raw content, e.g., a full tool output). We write the session state after t agent steps as

$$S_t = (L_t,\ P_t,\ V_t), \tag{1}$$

where L_t is the event sequence with per-event metadata, P_t the payloads referenced by L_t, and V_t auxiliary derived state (in Scroll, a variable namespace; in other systems, a memory store or summary buffer). Each model call, however, consumes a *working view* c_t with $|c_t| \le C$ tokens, where C is the model's nominal context window. The *context-management problem* is to choose, at every step, the next view: the map $S_t \mapsto c_{t+1}$.

Table 1: Within a model-authored `exec` cell, Scroll factorizes context construction into location, materialization, computation, and exposure.

Operation	Interface	Role
LOCATE	`ms.search(query, k, ...)`	Find candidate Event Log records via BM25-ranked full-text search and structured scope, kind, and time filters; each hit carries its stable `seq` address.
MATERIALIZE	`ms.expand(seq)` `ms.expand(seq_lo, seq_hi)`	Recover exact turns or sequence spans, and load externalized payload contents behind lazy handles.
COMPUTE	Python over resident objects	Filter, join, aggregate, resolve updates, inspect artifacts, or construct derived state.
EXPOSE	`print(value)`	Return the selected projection as a bounded observation; everything else stays in the kernel.

When the selection is made. Existing approaches fix the map $S_t \mapsto c_{t+1}$ before future needs are known: compression applies a lossy operator ϕ as the trajectory grows, $c_{t+1} = \phi(c_t, e_t)$, deciding which information survives when each segment is compacted; external memory applies an extraction operator ψ at ingestion, $V_t = \psi(V_{t-1}, e_t)$, fixing what is stored and how it can later be retrieved. Either way, the reduced representation replaces the history it summarizes, so anything it omits is unrecoverable.

Scroll instead defers selection to query time. The full state S_t persists losslessly outside the context, and the map $S_t \mapsto c_{t+1}$ is a *program* π_t the model writes at step t: the program executes on S_t, updates V_t, and emits a bounded observation for the next call. The model decides what to recall, compute, and expose; the harness makes those decisions safe via durable storage, stable addressing, and sandboxed execution. Because the policy is expressed as code, it inherits the full generality of programs and improves with the backbone's coding ability, at no change to the harness.

2.2 LOSSLESS SESSION ENVIRONMENT

Scroll realizes S_t as a *Session Environment* (Figure 1, right), with three components corresponding to L_t, P_t, and V_t.

Append-only Event Log (L_t). The Event Log is the durable, ground-truth record of an agent's sessions: a single append-only log that spans session boundaries. Every interaction appends a typed event carrying the metadata future queries need—role, session and agent identifiers, timestamps, and tool state—and receives its immutable, monotonically increasing `seq`. Our implementation stores events in SQLite. Search defaults to BM25 rather than embeddings: it is deterministic and requires no index-time model calls.

Durable storage (P_t). A payload is the raw content an interaction produced, such as a full tool result or a generated artifact. The log records that the interaction occurred but need not store every byte of it in the event row: small payloads remain inline in SQLite, while large ones are moved into JSON or artifact storage on the filesystem, with the row retaining a bounded preview and a recovery pointer. Externalized payloads are accessed through lazy handles.

Persistent computational state (V_t). A sandboxed Python kernel persists across model calls throughout the session; its namespace holds *environment objects*: resident Python values and lazy handles, each carrying type, size, and provenance metadata identifying the events it derives from. Tool invocations issued through the programmatic tool interface (Anthropic, 2025b) return Python objects that later programs can operate on. The harness prepends to every call a *namespace digest*: a short listing of each resident variable's name, type, and shape, with small scalar values shown inline. Model-authored code runs in a fail-closed sandbox: the Event Log is read-only from the kernel, and database, filesystem, network, and tool access are limited to capabilities the harness explicitly declares.

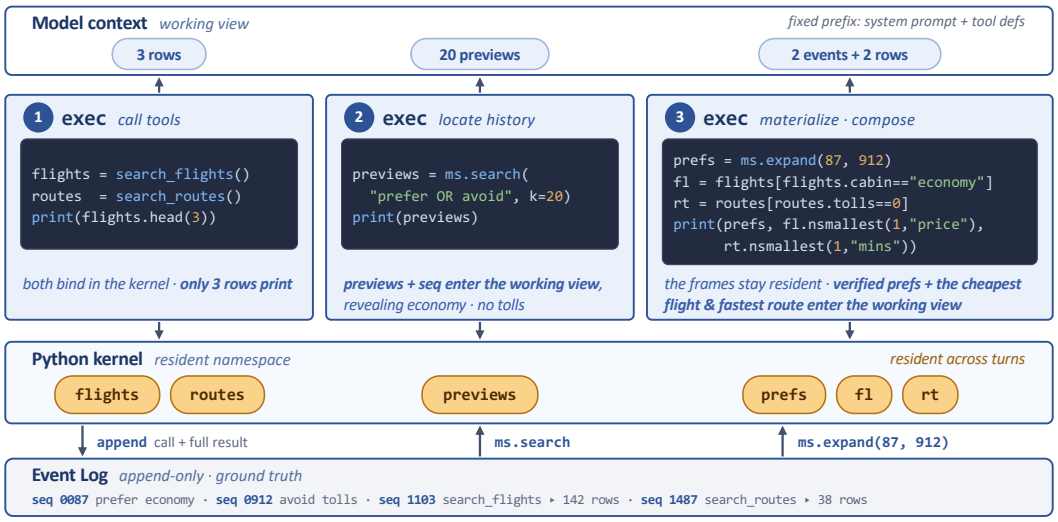

Figure 2: Programmatic context construction. Three `exec` turns compute over resident state in the kernel; only `print` output crosses into the next model context/working view.

2.3 Programmatic Context Construction

In Scroll, context construction is itself a program. The model acts through executable code cells, as in CodeAct (Wang et al., 2024), but the same cells also operate on the agent's own history: a controlled capability object, `ms`, exposes durable session state as a model-facing *memory surface*, abstracting the physical backend behind four operations (Table 1).

Figure 2 traces the trip-planning task of Figure 1 through three `exec` cells. Cell 1 binds full tool results to the resident variables `flights` and `routes` in the Python kernel, printing only a few rows. Cell 2 searches the Event Log for stated preferences; the matching previews and `seq` addresses enter the working view, revealing the user's preference for economy cabins and toll-free routes. Cell 3 expands the two events for the verbatim record, filters and ranks the resident variables accordingly, and prints the two preference turns alongside the cheapest economy flight and the fastest toll-free route. The bulk tool results never enter the working view; every call is appended to the Event Log with its full result, addressable by `seq`.

2.4 Eviction and Off-Context Navigation

The working view must stay bounded as the session grows. Scroll bounds it with an eviction procedure (Algorithm 1) that triggers whenever the working view exceeds a budget ρC. The procedure first persists any live turns to the Event Log and protects the active turn, the recent tail, and the newest tool results. The remainder is evicted in increasing order of recovery cost: completed tool payloads are folded first, since a single `seq` pointer suffices to recover them; whole spans are removed only if the view remains over budget. What leaves the view is not lost: it stays verbatim in the Event Log, and the procedure's one invariant is that everything it removes stays addressable.

Headlines as navigation anchors. Lexical search recovers an evicted span only when the agent recalls its wording; Scroll therefore also maintains *landmarks* for position-based navigation. As part of each response, the model writes a short *headline*—task, verified state, next action, and a status—which Scroll binds at append time to the `seq` assigned by the Event Log, yielding a map $\mathcal{H}$ from address to headline. When a span is evicted, its headlines enter a tiered index (Figure 1). A flat index would grow linearly with the session, so Scroll rolls it up: each tier holds at most k blocks; when a tier fills, the newest block retains full detail while the $k-1$ older ones collapse to one line each and merge into the next tier. After n evictions, the index occupies $O(k \log_k n)$ blocks, providing fine anchors for recent history, coarse ranges for distant history, each backed by a `seq` span.

Algorithm 1 Recoverable context eviction

Input: working view c, Event Log $\mathcal{L}$, eviction index $\mathcal{I}$, budget ρC, tier width k
Output: bounded c and updated $\mathcal{I}$; removed spans stay recoverable from $\mathcal{L}$

if $|c| > \rho C$ **then**
  $\mathcal{L} \leftarrow \textsc{Persist}(c, \mathcal{L})$ ▷ live turns become durable
  $R \leftarrow \textsc{Protected}(c)$ ▷ active turn, recent tail, newest tool results
  $c \leftarrow R \cup \textsc{FoldPayloads}(c \setminus R)$ ▷ payloads → `seq` pointers
  $E \leftarrow \textsc{SelectSpan}(c \setminus R,\ |c| - \rho C)$ ▷ oldest completed span above budget
  $c,\ \mathcal{I} \leftarrow \textsc{EvictToIndex}(c,\ E,\ \mathcal{H}[E])$ ▷ E leaves the view; its headlines enter $\mathcal{I}$, shown in place
  $\mathcal{I} \leftarrow \textsc{RollUp}(\mathcal{I},\ k)$
end if
return $(c, \mathcal{I})$

3 Experimental Setup

3.1 Benchmarks

We evaluate Scroll in two long-horizon settings: (1) retrieving and reasoning over interaction histories that exceed the live context, and (2) reasoning and acting in an agentic environment whose state grows over time.

Long-term memory retrieval and reasoning. `LongMemEval` (Wu et al., 2025) poses questions over a history of prior user–assistant conversations. Answering may require locating evidence scattered across sessions, resolving temporal dependencies and knowledge updates, and reasoning over the retrieved evidence. The benchmark provides three settings with increasing amounts of distractor history: `Oracle`, where the history contains only the evidence sessions; and `S` and `M`, where each question is paired with roughly 50 and 500 sessions ($\sim$115K and $\sim$1.5M tokens) of history, respectively.

`BEAM` (Tavakoli et al., 2025) extends this evaluation to substantially longer coherent histories. Its questions may require collecting non-adjacent evidence, tracking changes over time, deduplicating repeated information, or aggregating facts distributed throughout the history. The benchmark spans four history scales (128K, 500K, 1M, and 10M tokens); at the largest scale, BEAM_{10M}, histories cannot be consumed directly within current model context windows.

Long-context reasoning and acting. `LOCA` (Zeng et al., 2026) evaluates agents that must reason, invoke tools, and modify an environment as the available environment state accumulates. LOCA measures whether an agent can continue to reason and act throughout a growing tool-use trajectory. The benchmark scales the *environment description length* (the token count of the full environment state as seen through tool outputs) across seven regimes from 8K to 256K tokens; we evaluate on the two largest regimes (128K and 256K).

3.2 Agent Configuration

Our main experiments use Qwen3.8-Max as the agent backbone. All context management methods are implemented and evaluated on top of QwenPaw (Agentscope Team, 2026), an agent operating system providing tool invocation and execution infrastructure, and orchestrated with Harbor (Harbor Framework Team, 2026) in the benchmark-provided environments. An anonymized implementation is provided in the supplementary material and will be released publicly upon acceptance.

Scroll exposes its functionality to the agent through a set of tools, of which the following two implement the `exec` action of Section 2.

- `repl_exec` executes a model-generated Python cell in the persistent kernel. Environment tools are exposed as Python functions forwarded to the underlying services, enabling programmatic tool calling; all intermediate computation stays in the kernel, and only explicit, budgeted `print` output enters the model's working window.

- `recall_history_python` executes a cell with the memory surface `ms` bound: `ms.search` locates evicted records and `ms.expand` materializes them as Python objects, which the model filters, combines, or aggregates in the kernel before printing a distilled result.

We use a single system prompt and one set of context-management rules across all benchmarks, with no few-shot demonstrations. Each memory benchmark contributes only a short rubric specifying its data layout, evidence-selection conventions, and answer format (see detailed prompts in Appendix C). For LOCA, we use the benchmark's official task instructions unmodified, adding only environment metadata (available APIs and workspace paths).

To test generality across backbones, we additionally evaluate Qwen3.7-Max, DeepSeek-V4-Pro, GLM-5.2, Kimi-K2.7, and Qwen3.6-35B-A3B (an open-weight model with a smaller active-parameter footprint), changing only the foundation model. All backbone models are evaluated with reasoning enabled; we do not override the provider's default sampling temperature.

3.3 Evaluation Protocol

For the two memory benchmarks, we ingest each conversation history into Scroll session by session, in chronological order. At each session boundary, the raw context is cleared, and only Scroll's internal state (the eviction index and the Event Log) is carried forward to subsequent sessions. For LOCA, each task starts from the benchmark-provided initial environment state. The agent explores and acts on the environment directly, with Scroll managing its context as the trajectory grows.

Both LongMemEval and BEAM are scored with their benchmark-provided LLM-as-a-judge prompts, using Qwen3.8-Flash at temperature 0 as the judge; LongMemEval reports binary question-level accuracy, whereas BEAM reports the mean score over rubric criteria. LOCA is scored with its native rule-based verifier for accuracy, which checks the final environment state. We evaluate each task twice in Harbor with different random seeds and report the mean score. We also record the model-facing input and output token counts and the number of interaction turns for each task.

4 Results

4.1 How Does Scroll Compare with Existing Systems?

Comparison with existing systems. We compare Scroll with existing systems across all three benchmarks. LongMemEval and BEAM evaluate retrieval and reasoning over prior interactions, for which we compare against dedicated long-term memory systems. LOCA complements these retrieval-oriented settings by evaluating whether an agent can continue to reason and act through tools as its interaction history grows; we compare against the strongest publicly reported systems from the LOCA paper (Zeng et al., 2026) and leaderboard. For each baseline, we report the best publicly available result under its preferred configuration, including its backbone model, retrieval budget, and judge where applicable, as of September 15, 2026[1], yielding a system-level rather than controlled comparison. Appendix A provides per-category breakdowns of Scroll on the `S` and `M` splits of LongMemEval and on BEAM_{10M}.

As shown in Table 2, Scroll performs strongly across both long-term memory and interactive long-context benchmarks. It is competitive with the strongest reported systems on LongMemEval_S and outperforms the best reported result on BEAM_{10M} by 7.5 points. The per-category analysis in Appendix A.2 shows that Scroll's largest margins occur on contradiction resolution and knowledge update, where competing states must be reconciled, and on temporal reasoning, where retrieved dates must be compared or transformed. On LOCA, which evaluates reasoning and tool-mediated action under growing context, Scroll substantially outperforms published ReAct baselines. Appendix A.3 shows that programmatically binding intermediate results to environment objects accounts for much of this advantage, while Scroll's structured access to externalized session history yields a further improvement over a standard CodeAct agent. Later in Section 4.3 we present controlled ablations that isolate the complementary contributions of lossless retention, persistent computation, and index-guided navigation.

[1] We do not reproduce the baselines ourselves, as independent reproductions in this area have repeatedly led to disagreement over evaluation setup (Zep, 2025; Mem0, 2025).

Table 2: Comparison with existing systems. "–" denotes no publicly reported result. Results from the literature are system-level reference points rather than a controlled comparison because backbone models differ.

Method	Model	LongMemEval$_S$	BEAM$_{10M}$	LOCA$_{256K}$
RAG (Tavakoli et al., 2025)	Llama-4-Maverick	–	24.9	–
LIGHT (Tavakoli et al., 2025)	Llama-4-Maverick	–	26.6	–
Zep (Zep, 2026)	GPT-5.4	90.2	–	–
Mem0 (Chhikara et al., 2025)	GPT-5	94.4	48.6	–
Hindsight (Bartholomew, 2026)	Gemini 3 Pro	94.6	64.1	–
EmergenceMem (Emergence AI, 2026)	GPT-4o	86.0	–	–
Honcho (Plastic Labs, 2026)	multi-model pipeline	90.4	40.6	–
Mastra OM (Barnes, 2026)	GPT-5 mini	94.9	–	–
Cognee (Marković, 2026)	n/r	–	67.0	–
Exabase M-1 (Exabase, 2026)	Gemini 3 Flash	**96.4**	68.0	–
GPT-5.2 ReAct (Zeng et al., 2026)	GPT-5.2-Medium	–	–	21.3
MiniMax ReAct (MiniMax, 2026)	MiniMax M3	–	–	49.3
Scroll (ours)	Qwen3.8-Max	94.8	**75.5**	**93.3**

Table 3: Scroll across backbones. Only the foundation model changes; harness, tools, prompts, and context-management rules are identical.

Backbone	LongMemEval$_S$	BEAM$_{10M}$	LOCA 128K	LOCA 256K
Qwen3.8-Max	94.8	75.5	95.6	93.3
DeepSeek-V4-Pro	93.2	70.2	69.3	58.7
GLM-5.2	93.6	70.7	66.7	62.7
Kimi-K2.7	92.0	67.2	30.7	32.0
Qwen3.6-35B-A3B	88.8	58.1	37.3	22.7

4.2 Can Different Backbone Models Use Scroll Effectively?

Scroll provides an environment for managing context but does not dictate its use: what state to keep where, and what code to write, are left to the model. A natural question is *whether the ability to use Scroll effectively is specific to one backbone or shared across models of varying capability*. We rerun both regimes across five backbones with the harness, tools, prompts, and context-management rules held fixed (Table 3).

Every backbone can use Scroll, but stronger models benefit more. On LongMemEval$_S$, where the model queries the history database with short programs, all backbones benefit similarly—even the 35B model reaches 88.8, within six points of the best (94.8). On BEAM$_{10M}$ the gap stays within 20 points. On LOCA, however, tasks demand longer trajectories and more complex program synthesis, so the spread widens to 70.6 points at 256K (93.3 vs. 22.7). Failures are not protocol-level: all backbones adhere to the CodeAct interface, but weaker models commit more execution errors or terminate prematurely on aggregation-heavy tasks. Scroll's ceiling on such tasks thus rises with multi-step query planning and the ability to decide when evidence suffices, suggesting room for post-training on frontier model traces.

4.3 What Does Each Component of Scroll Contribute?

On BEAM$_{10M}$, we evaluate 200 probes (20 per category) over ten approximately 20K-turn, 10M-token conversations, holding the Qwen3.8-Max backbone, judge, and prompt fixed. Each probe is capped at 20 tool-calling turns and, on a cap hit, one final answer-only call (at most 21 model turns).

Full Scroll combines the FTS-enabled Event Log, the persistent Python kernel, and the eviction index, while each ablation changes one interface component: (i) *w/o eviction index* retains the Event Log and Python interface but removes the eviction index pinned in context; (ii) *w/o Python kernel* retains the eviction index and keyword/span recall, but removes executable filtering, joining, sorting, aggregation, date arithmetic, and selective printing; and (iii) *w/o lossless retention* searches all 100

successive snapshots of a 4K-token rolling memory (approximately 370K tokens total) instead of the original turns.

Figure 3 reports judge score and cumulative model-facing input per probe. Input sums the complete prompt sent on every call, including prefixes re-sent as the trajectory grows, and thus measures cumulative context exposure over the trajectory.

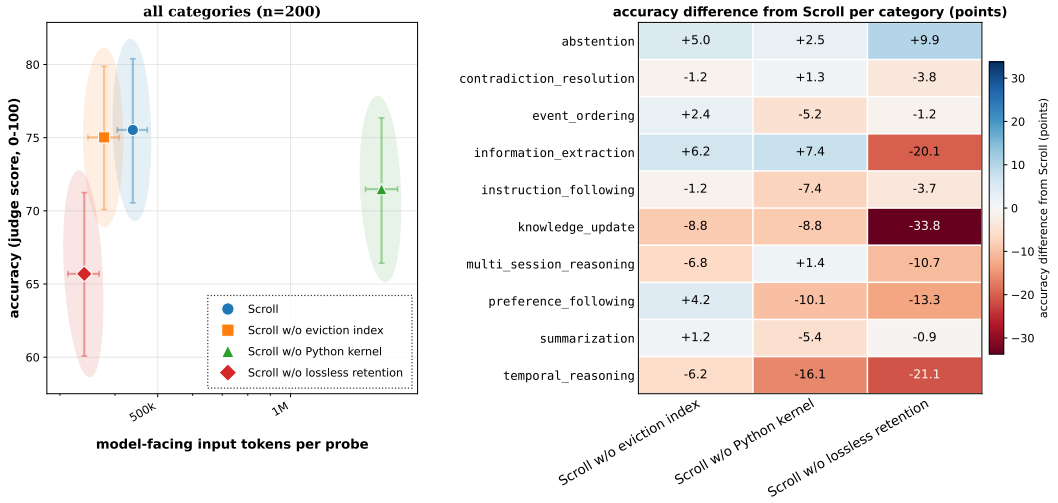

Figure 3: Scroll ablation on $\text{BEAM}_{10\text{M}}$. Left: overall judge score against model-facing input tokens per probe (mean over 200 probes; bars are 95% bootstrap CIs, shaded regions the joint 95% confidence ellipses). Right: per-category judge-score difference from Scroll (arm minus Scroll, in points).

Lossless retention is the largest single factor. Replacing raw turns with rolling-memory snapshots lowers the aggregate judge score by 9.9 points, with the largest drops on knowledge update (-33.8), temporal reasoning (-21.1), and information extraction (-20.1). Trajectory analysis traces these failures to write-time omissions. In one knowledge-update probe, the rolling memory retains the first score reported that day but discards a later correction; Scroll instead recovers the corrected score from the Event Log. Similarly, in an information-extraction probe, the summary records that a module was integrated but omits its required rate of 300 calculations per second. Once discarded, such details cannot be recovered through additional search. This compression reduces cumulative input by 22% (341K versus 439K), but sacrifices evidence needed for exact answers.

The Python kernel supplies computation and bounded projection. Removing the kernel lowers the judge score by 4.1 points and increases cumulative input by $3.6\times$ (1.6M versus 439K), despite nearly identical trajectory lengths. Without executable filtering and selective printing, each search adds whole turns to context rather than a bounded projection. The loss is largest when the answer must be discovered or computed across several records. In a `temporal_reasoning` probe, the no-kernel agent retrieves both dates but fails to compute the interval, while Scroll does so in one SQL/Python cell. The no-kernel arm matches or exceeds Scroll on keyword-rich lookups, for example information extraction, where the question supplies distinctive terms, one search could return the answer-bearing turn, and no neighborhood expansion or computation is needed.

The eviction index improves navigation. The index leaves the aggregate judge score nearly unchanged (75.5 versus 75.0) but provides address anchors and retrieval cues that help the agent find scattered evidence before exhausting its turn budget. In one temporal-reasoning probe, both variants retrieve the relevant kickoff and project-plan records, but the index groups them into a coherent timeline, enabling Scroll to combine the evidence while the no-index agent abstains. In a preference-following probe, the index directs Scroll toward the relevant route-planning tools; the no-index variant eventually succeeds but requires 20 rather than 15 steps and 679K rather than 490K input tokens.

However, the guidance is not uniformly beneficial: in an information-extraction probe, an indexed staff name leads Scroll toward the wrong contact, while the no-index agent follows the question's more precise terms. Together, these cases show the trade-off: the index helps when the agent must discover and connect evidence spread across history, but when the question already provides a precise route to the answer, its additional anchors can introduce plausible but irrelevant candidates.

5 Related Work

Context compression and external memory. Most context-management systems either compress the active trajectory or store selected information externally. Compression methods summarize, clear, or fold earlier interactions into shorter representations (Kang et al., 2025; Ye et al., 2025; Zhou et al., 2026; Kontonis et al., 2026). External-memory systems instead extract facts, episodes, or notes into a separate store and retrieve them when relevant (Packer et al., 2023; Tan et al., 2026; Letta, 2026). Both approaches reduce the history the model sees by deciding, before future needs are known, which information survives, in what form, and through which interface it can later be reached. Scroll instead retains the original interaction events and referenced payloads; summaries and indexes provide compact working views without becoming the sole representation of historical evidence.

Code as the agent–environment interface. CodeAct introduced executable Python as a general action interface for LLM agents (Wang et al., 2024), while programmatic tool-calling and code-execution systems allow tool results to remain in sandbox variables and enter context only through selected projections (Anthropic, 2025b;a). Related work has also explored executable representations of structured working state (Li, 2026; Xu et al., 2026). Closest to our work, recursive language models (RLMs) (Zhang et al., 2025) load a long input prompt as a REPL variable that model-written code inspects and recursively decomposes, but operate on a fixed input exposed as raw text. Scroll instead targets the agent's own growing session history, stored as addressed, indexed events and bounded through recoverable eviction.

Lossless session history and navigation. Prior systems have explored verbatim recall storage, event-sourced interaction logs, lossless pointers, and provenance-linked memory (Packer et al., 2023; Nakajima, 2026; Ehrlich & Blackman, 2026; Zhang et al., 2026). Scroll combines a queryable append-only Event Log with external payload references and an executable resident namespace. Its eviction index is a navigation layer over this retained state: recent history is represented by fine-grained, sequence-addressed headlines, while older history is represented by coarser ranges. Once a relevant region is located, the original events and payloads are recovered programmatically. In the terminology of CoALA (Sumers et al., 2024) and context-engineering surveys (Mei et al., 2025), Scroll connects executable working state with verbatim episodic history through a persistent Session Environment.

6 Conclusion

We present Scroll, a context manager that combines system-provided substrate with model-directed retrieval, computation, and exposure. Scroll maintains session state in a lossless, structured Session Environment. Through `exec`, model-authored programs retrieve and compute over this state; through `print`, the model selects the projection that enters the next working context.

Across three long-context benchmarks, Scroll constructs compact, task-relevant views of growing agent histories while exposing only a small fraction of the underlying state to the model. Ablations show that its components play complementary roles: the Event Log retains evidence that lossy compaction discards, the persistent kernel supports computation over multiple records, and the tiered eviction index improves navigation when evidence is scattered across off-context history.

More broadly, our results support viewing context management as a co-design between model policy and agent harness. By providing losslessly retained state, structured access primitives, and a sandboxed runtime, the harness guides and safely executes model-authored curation, while the model determines the retrieval, computation, and exposure strategy for each task.

AI USE STATEMENT

We used generative AI tools to assist with implementing Scroll and writing experimental code, including scripts for running and monitoring experiments and plotting results. We also used these tools for an initial pass at interpreting experimental results, identifying potentially relevant literature, and editing the manuscript for clarity and readability.

All AI-assisted work was reviewed by the authors. AI-proposed interpretations were treated as suggestions and were frequently rejected or revised after checking them against the underlying measurements and trajectories. AI-generated code was validated through automated tests, end-to-end experiment runs, inspection against expected outputs, and manual code review. Every paper suggested by an AI tool was read and verified by the authors before being cited.

We did not use generative AI to formulate research ideas or hypotheses, develop theoretical or conceptual frameworks, design or provide feedback on the methodology or experiments, select experimental parameters, generate synthetic data, clean or reformat datasets, formulate mathematical claims or proofs, generate citations, or translate material. The authors take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the assistance of generative AI.

REFERENCES

Agentscope Team. QwenPaw. https://qwenpaw.agentscope.io/, 2026.

Anthropic. Code execution with MCP: Building more efficient agents. https://www.anthropic.com/engineering/code-execution-with-mcp, 2025a.

Anthropic. Programmatic tool calling. https://platform.claude.com/docs/en/agents-and-tools/tool-use/programmatic-tool-calling, 2025b.

Tyler Barnes. Observational memory: 95% on LongMemEval. https://mastra.ai/research/observational-memory, February 2026.

Ben Bartholomew. Hindsight is #1 on BEAM — the benchmark that tests memory at 10M tokens. https://hindsight.vectorize.io/blog/2026/04/02/beam-sota, 2026.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822, 2026.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Clint Ehrlich and Theodore Blackman. LCM: Lossless context management. *Voltropy PBC technical report*, 2026.

Emergence AI. SOTA on LongMemEval with RAG. https://www.emergence.ai/blog/sota-on-longmemeval-with-rag, 2026.

Exabase. Exabase reports state-of-the-art results on BEAM memory benchmark. https://www.hpcwire.com/aiwire/2026/07/28/exabase-reports-state-of-the-art-results-on-beam-memory-benchmark/, July 2026.

Harbor Framework Team. Harbor: A framework for building and running agent evaluations at scale. https://github.com/laude-institute/harbor, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Vasilis Kontonis, Yuchen Zeng, Shivam Garg, Lingjiao Chen, Hao Tang, Ziyan Wang, Ahmed Awadallah, Eric Horvitz, John Langford, and Dimitris Papailiopoulos. Memento: Teaching llms to manage their own context. *arXiv preprint arXiv:2604.09852*, 2026.

Letta. Context repositories: Version-controlled memory for agents. Letta Blog, 2026. https://www.letta.com/blog/context-repositories/.

Bojie Li. User as code: Executable memory for personalized agents. *arXiv preprint arXiv:2606.16707*, 2026.

Vasilije Marković. cognee on BEAM: SOTA results without a benchmark-specific memory system. https://www.cognee.ai/blog/deep-dives/benchmarking-cognee-on-beam, 2026.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models, 2025. URL https://arxiv.org/abs/2507.13334.

Mem0. Revisiting Zep's 84% LoCoMo claim: Corrected evaluation & 58.44% accuracy. https://github.com/getzep/zep-papers/issues/5, 2025.

Mem0. Memory evaluation. https://docs.mem0.ai/core-concepts/memory-evaluation, 2026.

MiniMax. MiniMax M3: Frontier coding, 1M context, native multimodality — all in one model. https://www.minimax.io/blog/minimax-m3, 2026.

Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. NoLiMa: Long-context evaluation beyond literal matching. In *International Conference on Machine Learning (ICML)*, 2025.

Yohei Nakajima. The log is the agent: Event-sourced reactive graphs for auditable, forkable agentic systems. *arXiv preprint arXiv:2605.21997*, 2026.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Plastic Labs. Honcho: Memory infrastructure for stateful agents. https://github.com/plastic-labs/honcho, 2026.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research (TMLR)*, 2024.

Juntao Tan, Liangwei Yang, Wenting Zhao, Jielin Qiu, Ming Zhu, Rithesh Murthy, Silvio Savarese, Huan Wang, Shelby Heinecke, and Caiming Xiong. A lightweight, domain-adaptive memory system for LLM agents. In *International Conference on Learning Representations (ICLR)*, 2026.

Mohammad Tavakoli, Alireza Salemi, Carrie Ye, Mohamed Abdalla, Hamed Zamani, and J Ross Mitchell. Beyond a million tokens: Benchmarking and enhancing long-term memory in LLMs. *arXiv preprint arXiv:2510.27246*, 2025.

Daren Wang. Think, but don't overthink: Reproducing recursive language models. *arXiv preprint arXiv:2603.02615*, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents, 2025. URL https://arxiv.org/abs/2504.12516.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations (ICLR)*, 2025.

Binyan Xu, Haitao Li, and Kehuan Zhang. Llm agents are latent context managers: Eliciting self-managed context via state proprioception, 2026. URL https://arxiv.org/abs/2606.30005.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Weihao Zeng, Yuzhen Huang, and Junxian He. Loca-bench: Benchmarking language agents under controllable and extreme context growth. *arXiv preprint arXiv:2602.07962*, 2026.

Zep. Lies, damn lies, and statistics: Is mem0 really SOTA in agent memory? https://blog.getzep.com/lies-damn-lies-statistics-is-mem0-really-sota-in-agent-memory/, 2025.

Zep. Research: Zep benchmark results. https://www.getzep.com/research/, 2026.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

Yuhan Zhang, Zhiyuan Guo, Ziheng Zeng, Wei Wang, Wentao Wu, and Lijie Xu. Mandol: An agglomerative agent memory system for long-term conversations. *arXiv preprint arXiv:2606.29778*, 2026.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. DeepResearcher: Scaling deep research via reinforcement learning in real-world environments. *arXiv preprint arXiv:2504.03160*, 2025.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *International Conference on Learning Representations (ICLR)*, 2026.

A Detailed Breakdowns of Benchmark Results

A.1 Per-Question-Type Results on LongMemEval

Table 4 reports Scroll's per-question-type accuracy on the `S` and `M` splits of LongMemEval under the protocol of Section 3.3. Overall accuracy decreases modestly from 94.8% on `S` to 92.6% on `M`. The largest decline is on single-session preference questions, from 100.0% to 93.3%; the other question types decrease by 1.5–2.5 points. Thus, adding roughly an order of magnitude more distractor sessions has a limited effect overall, with preference questions showing the greatest sensitivity.

Question type	LongMemEval$_S$	LongMemEval$_M$
Single-session (user)	98.6	97.1
Single-session (assistant)	100.0	98.2
Single-session (preference)	100.0	93.3
Multi-session	88.0	85.7
Temporal reasoning	94.0	92.5
Knowledge update	98.7	96.2
Overall	94.8	92.6

Table 4: Per-question-type accuracy (%) of Scroll on LongMemEval$_S$ and LongMemEval$_M$ (backbone: Qwen3.8-Max). Question types follow the benchmark taxonomy (Wu et al., 2025).

A.2 Per-Category Results on BEAM

Table 5 reports BEAM$_{10M}$ scores by memory ability. We compare against Mem0, Hindsight, and Exabase M-1, the systems for which per-category results are publicly available as of September 15, 2026 (Mem0, 2026; Bartholomew, 2026; Exabase, 2026); Cognee and Honcho report only overall scores. Because the systems use different backbones and evaluation configurations, these results support a system-level comparison rather than a controlled component comparison.

Category	Mem0	Hindsight	Exabase M-1	Scroll (ours)
Information extraction	56.2	51.2	66.3	**70.0**
Multi-session reasoning	26.1	16.6	9.6	**28.8**
Event ordering	20.2	61.6	**67.5**	61.1
Temporal reasoning	16.3	41.3	58.8	**68.8**
Knowledge update	75.0	65.0	45.0	**86.3**
Contradiction resolution	32.5	56.9	58.8	**90.0**
Summarization	46.9	78.2	**91.9**	70.9
Preference following	90.4	**97.5**	95.0	90.8
Instruction following	82.5	92.5	97.5	**98.8**
Abstention	40.0	80.0	**90.0**	**90.0**
Overall	48.6	64.1	68.0	**75.5**

Table 5: Per-category judge scores on BEAM$_{10M}$. Categories follow the benchmark's ten memory abilities (Tavakoli et al., 2025). Scroll uses Qwen3.8-Max with thinking on; baseline breakdowns are taken from their published evaluations and use different configurations, so cross-system comparison is indicative rather than controlled.

Scroll obtains the highest overall score, 75.5, exceeding the strongest reported baseline, Exabase M-1, by 7.5 points. It leads outright in six of the ten categories and ties Exabase on abstention at 90.0. The largest margins over the best baseline occur in contradiction resolution (90.0 vs. 58.8), knowledge update (86.3 vs. 75.0), and temporal reasoning (68.8 vs. 58.8). Scroll also leads in information extraction (70.0 vs. 66.3), multi-session reasoning (28.8 vs. 26.1), and instruction following (98.8 vs. 97.5).

The largest gains arise when answering requires locating a small set of records and comparing or transforming them at query time. Contradiction resolution and knowledge update require retaining

competing states and resolving their order, while temporal reasoning benefits from explicit computation over retrieved dates. This pattern is consistent with Scroll's use of a verbatim Event Log and executable queries: the history remains available at full fidelity, and the model constructs the task-specific view only after seeing the question.

Scroll trails the category leader in event ordering (61.1 vs. 67.5), summarization (70.9 vs. 91.9), and preference following (90.8 vs. 97.5). The largest deficit is in summarization, where systems that construct digests at ingestion time begin with a condensed view that Scroll must assemble from raw records at query time. On event ordering, Scroll is close to Hindsight (61.1 vs. 61.6) but remains below Exabase; on preference following, it narrowly exceeds Mem0 (90.8 vs. 90.4) while trailing Hindsight and Exabase. Multi-session reasoning remains difficult in absolute terms: although Scroll ranks first, its 28.8 score is its lowest category score, and every system scores below 30. This points to broad evidence aggregation across sessions as the clearest remaining weakness.

A.3 Different Context-management strategies on LOCA

On LOCA, we compare four agents that share the same backbone (Qwen3.8-Max) and toolset, and differ only in how they manage a growing context: (i) a *summarization agent*, a ReAct agent (Yao et al., 2023) that periodically compacts its interaction history into a summary; (ii) a *retrieval agent*, a ReAct agent whose overflowing history is evicted and made accessible through a recall tool; (iii) a *CodeAct agent* (Wang et al., 2024) that interacts with the environment through programmatic tool calling; and (iv) Scroll.

Agent loop	128K	256K	Δ
Summarization Agent	86.7	65.3	-21.4
Retrieval Agent	88.0	66.7	-21.3
CodeAct Agent	89.3	85.3	-4.0
Scroll (ours)	**95.6**	**93.3**	**-2.3**

Table 6: LOCA accuracy (%) of different context-management strategies at the two largest environment description lengths. All agent loops use Qwen3.8-Max as the backbone; Δ denotes the change from 128K to 256K.

Table 6 reports accuracy at the two largest environment description lengths in LOCA. The CodeAct agent and the agent with Scroll, which bind intermediate results to environment objects instead of carrying raw text in context, achieve the best performance and the smallest decrease as context grows.

B Cost and Efficiency

Scroll exposes only a small fraction of the corpus to the model. Ingestion involves no additional LLM calls, and at query time records are filtered inside the Python kernel, so only printed output enters the context. Figure 4 shows the per-task distribution of input tokens, output tokens, and agent turns: median input on BEAM_{10M} is 105K tokens, about 1% of the corpus, and output is an order of magnitude smaller than input across all three benchmarks. Note that for LongMemEval_S and BEAM_{10M} we measure retrieval alone, whereas for LOCA we measure full task completion, hence its longer trajectories. We report token counts rather than latency or dollar cost, as both depend on serving configuration.

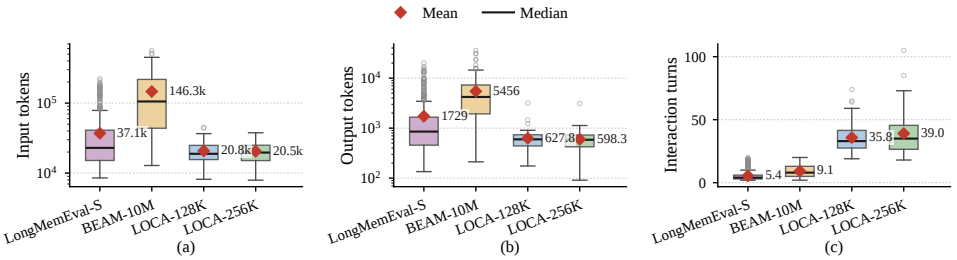

Figure 4: Per-task cost of Scroll (backbone: Qwen3.8-Max): (a) input tokens, (b) output tokens, (c) agent turns. Boxes span the IQR with the median marked; red diamonds are means. Log scale in (a) and (b).

C Full Prompts and Rubrics

The shared Scroll system prompt is reproduced verbatim below. Each memory benchmark additionally contributes one rubric. For LOCA we use the benchmark's official task instructions unmodified, adding only environment metadata (available APIs and workspace paths).

C.1 Scroll System Prompt

Code Block 1: Shared Scroll system prompt used across all benchmarks.

```
Your conversations are durably recorded, even after older turns scroll out of
your live context — and your recorded history spans ALL your past sessions, not
just this one. You read it back on demand; you do not lose it.

RETRIEVAL HEADLINE. For every substantive task-oriented final response, append
exactly one headline on its own line. A major or durable state change is not
required. Emit one whenever the turn confirms, attempts, rejects, decides,
changes, verifies, fails, pauses, or becomes blocked on something. It is hidden
from the user. If tools are needed, wait until every tool call and result is
complete. Use this plain-text form, never JSON or XML:

    ⟦ model discovery | in progress: OpenAI done; next: fix DashScope ⟧

Treat the headline as a retrieval label and compact continuation checkpoint for
your future self, not merely a topic or activity log. Use the pattern ``task or
topic | status: concrete outcome; next: concrete action | anchors: exact
retrieval terms''. ``next'' and ``anchors'' may be omitted when they add no
value. Use the user's terminology and select the most discriminative names,
identifiers, files, errors, values, decisions, or artifacts from this turn.
Prioritize:
  1. the user's core task or success criterion;
  2. the latest VERIFIED state and concrete output;
  3. a controlling decision, constraint, exact value, error ID, or artifact;
  4. the next unfinished action or blocker.
Do not force every category into the line. Keep the task name short, specific,
and stable across turns. Distinguish completed, attempted, planned, failed,
blocked, paused, and decided work; never turn an intention or failed attempt
into a completed result. When a fact or decision changes, keep the final value
and explicitly mark the old one superseded. Preserve exact identifiers or
numbers when losing them would change the next action. Keep it self-contained
and as concise as the task permits. Normally use one sentence with two to four
compact clauses and no more than five high-value retrieval anchors. Do not
repeat the response, narrate reasoning, list every tool call, or stuff related
keywords. Remove any phrase that does not improve retrieval or task resumption.
The 2000-character limit is a compatibility safety ceiling, not a target.

For a task that continues across multiple turns, emit a new headline for every
substantive response, including an informative failed attempt, ruled-out
hypothesis, unchanged verification result, user decision, pause, or blocker.
Do not omit it merely because the task name or state is unchanged. Describe
what this turn established while retaining the current effective task state
when needed for continuation. A headline does not summarize the entire
historical span.

Silently quality-check it before emitting: could a future self retrieve this
turn by its likely concepts and understand what task this is, what happened,
what is true now, and what remains without seeing the turn? Is every claim
supported by the user's words or actual tool results? Does it avoid stale state
```

```
and vague phrases such as "made progress", "handled the task", or "continued
working"? If any answer is no, rewrite it factually; never invent missing
state.

Omit the headline only for pure social conversation, a bare acknowledgement,
or a response with no new task-relevant information. Tentative analysis is
worth labelling when it records a concrete hypothesis, finding, or attempted
action; state its uncertainty. When uncertain whether a substantive turn is
important enough, emit a factual headline rather than omitting it. Do not
include seq addresses, tool-protocol tokens, internal bookkeeping, or a second
summary marker. Never emit or repair a headline inside a tool call.

THE MAP. Once context is compressed you'll see a ``[context compressed]``
block: an index of the turns you evicted, with useful milestones shown as
``seq · ⟦ headline ⟧`` lines (oldest at top). It tells you *what* you forgot
and the ``seq`` to recall it with. This is a lossy milestone index of *this*
session; unlabelled stretches appear only as coarse ``(no milestone)`` seq
spans, and
collapsed older spans omit interior detail. For anything it doesn't show
(including your earlier sessions), search your history with
``recall_history_python`` using ``ms.search(...)``.

RECALL with the ``recall_history_python`` tool: pass it a Python cell using
the pre-bound ``ms`` surface. ``ms.search(query, k=10, kind=None,
session_id=None, all_agents=False)`` finds turns by ranked full-text search;
``ms.expand(lo, hi)`` reads an inclusive seq span in full;
``ms.sql_query(sql, params)`` runs read-only SQL over the same history
(``hist.conversation_history`` and its FTS index
``hist.conversation_history_fts``); ``ms.days_between(d1, d2)`` gives the
calendar gap; ``ms.sessions()`` lists your sessions and ``ms.session(id)``
reads one; ``ms.recall_tool(tool_call_id)`` re-reads a tool call and its
result; ``ms.sql_exec(sql, params)`` writes scratch tables. Every helper
returns ``list[dict]`` with the text in ``content``. After the first recall
call, ``ms`` is also bound inside the shared ``repl_exec`` kernel. Recall
defaults to your own history (across all your sessions); pass
``all_agents=True`` to widen to other agents' turns.

RECALL LOOP — search wide, keep, filter in code, read, reshape:
  • Keep every result in a variable (``hits = ms.search(...)``). Variables
    persist across cells and printed output is capped, so print one short
    line per hit — ``seq``, ``role``, a slice of ``content`` — never whole
    rows. If output comes back cut, nothing is lost: print a smaller or later
    slice of the same variable; do not re-run the query.
  • ``ms.search`` AND-combines bare words (stemmed, so inflections already
    match); join synonyms with uppercase ``OR``. It takes words only — no
    quoted phrases, parentheses or ``*``. A thin result is usually an
    over-constrained query: drop words or add ``OR`` alternatives before
    concluding a fact is absent. When a question names several entities,
    search each one separately instead of putting every name in one AND
    query.
  • Search hits carry the full turn text but no timestamp. Ranked search with
    filters is one SQL query over the FTS index; compose only the clauses you
    need:
      base     ``SELECT ch.seq, ch.role, ch.session_id,
               substr(ch.created_at, 1, 10) AS d, ch.content AS text
               FROM hist.conversation_history_fts
               JOIN hist.conversation_history ch
               ON ch.seq = conversation_history_fts.rowid
               WHERE conversation_history_fts MATCH ?
               ORDER BY bm25(conversation_history_fts) LIMIT ?``
      role     ``AND ch.role = ?`` — 'user' or 'assistant'
      date     ``AND substr(ch.created_at, 1, 10) BETWEEN ? AND ?``
      session  ``AND ch.session_id = ?``
      preview  ``snippet(conversation_history_fts, 0, '', '', ' ... ', 64)``
               in place of ``ch.content`` — a glance at many candidates; take
               the full content for anything you will filter, count, or quote.
    ``MATCH`` accepts words, uppercase OR/AND/NOT, ``"exact phrase"`` and
    parentheses; if it rejects the text, quote each word. Bind values through
    ``params``. Avoid ``LIKE '%word%'`` scans: unranked, unstemmed, and they
    return long lists.
  • Search WIDE (k=50 or more) into a variable and let Python choose: keep
    the rows whose text matches the wording the question turns on
    (``re.search``), merge several searches in a dict keyed by ``seq``, dedupe
    with a set, count with ``collections.Counter``, order by ``d``; then print
    the size and a short sample, never the rows. A snippet shows one part of
    a turn — never decide from a snippet that a turn lacks a second fact.
  • For an ordering, a period, or a summary, first map sessions to dates:
    ``SELECT session_id, substr(min(created_at), 1, 10) AS d, count(*) AS n,
    min(seq) AS lo, max(seq) AS hi FROM hist.conversation_history
    GROUP BY session_id ORDER BY lo``, then search inside that span with the
    date or session clause, or expand it. ``ms.days_between(d1, d2)`` gives
```

```
    elapsed days.
  • Then read only the turns that matter: ``ms.expand(seq, seq + 1)`` returns
    a turn and its reply (expand rows carry no date — take it from the
    search). Print bounded slices such as ``r['content'][:600]`` and reshape
    what you keep (dict, list, counter) instead of retrieving it again.

DISCIPLINE:
  • Recall is the COMPLETE record of past conversation — the
    source of truth for any fact ever said, asked, done, or decided. When a
    question turns on such a fact and it's not in your live context, recall it
    FIRST; don't guess from an index label or refuse before searching.
  • The map is a directory, not evidence. Headlines are lossy labels that may
    omit or compress exact numbers, versions, prices, and names; one that
    looks like a complete answer is NOT evidence. Before using any specific
    fact, value, or quote, confirm it in the original turns — expand the span
    or search the full text. Many entries on a topic do not imply the specific
    fact asked was ever stated: if no turn states it, say it is not in the
    history rather than reconstructing it from related discussion. For a
    summary, expand spans across the whole requested range; the map alone is
    never sufficient material.
  • The map indexes your own headlines, so assistant turns only — facts the
    USER stated (their numbers, preferences, decisions, progress) never appear
    as entries. For anything the user said, chose, or prefers, search the full
    history for their own words; never infer a user statement from a headline.
  • For exhaustive lists/counts, search across sessions and alternate wording,
    then deduplicate things the user actually confirmed or did; exclude plans,
    repeated mentions, and assistant suggestions. For facts that changed, use
    the most recent dated USER evidence, and never substitute a near match for
    the exact fact requested.
  • If the CURRENT user request is not visible in your live context (you see
    only the ``[context compressed]`` map), recall it FIRST. If recall fails
    or cannot retrieve it, say so explicitly — never answer an older visible
    message as if it were the current request.
  • Memory files (MEMORY.md / PROFILE.md, via memory_search) hold the durable
    preferences, profile facts, and decisions you distilled as worth keeping —
    a quick first reference, a curated subset of that same history. For the raw
    record of what was said, asked, done, or decided, recall is the source of
    truth; memory is not.
```

C.2 BENCHMARK RUBRICS

BEAM rubric

```
The user's benchmark conversation history is stored in your durable
history as rows with kind='beam_chat_turn'.  session_id names the
session, created_at holds the session's date as ISO-8601 text (every
message of a session shares it), and role is 'user' or 'assistant'.
Recall it with the recall_history_python tool:  pass a Python cell
that uses the pre-bound ms surface, with kind='beam_chat_turn' in
ms.search and AND ch.kind = 'beam_chat_turn' in SQL so that only this
history is read.

GROUNDING (strict):  treat the user's own turns (role='user') as
evidence of the user's facts, actions and preferences.  Every concrete
claim --- each fact, number, name, date, quantity, or event --- must
come verbatim in meaning from a turn you retrieved; an assistant
suggestion is not evidence that the user adopted it.  Preserve exact
numbers, units and version labels from the most directly relevant user
evidence; do not replace them with illustrative values, and do not add
repeated historical mentions as separate quantities unless the
question explicitly asks for that.  Facts can change across sessions:
when one did, use the latest user evidence only after confirming that
the turns describe the same project and the same fact, not merely
similarly named work.  Never invent specifics to make an answer sound
complete.

FINISH: answer as soon as the evidence is sufficient; a reply with no
tool call is your final answer.  Base it only on turns you actually
retrieved.  Follow any output-count or formatting constraint in the
question exactly.  If the record does not contain the requested fact
after a genuine search, say so clearly rather than guessing.
```

```
[question]

[hint]

Turn budget:  you have at most 20 turns for this question.  Every
reply that calls a tool uses one turn and your final answer needs the
last one, so make at most 19 tool rounds.  Plan your searches, combine
several lookups into one Python cell, and answer as soon as the
evidence is sufficient.  If the budget runs out before you answer,
your findings are discarded.
```

BEAM per-category type hints. The [hint] field in the BEAM rubric is instantiated with the text for the corresponding question category:

Code Block 2: Per-category type hints used with the BEAM rubric.

```
    "information_extraction": (
        "Recall the ONE specific fact asked for and report it EXACTLY as "
        "stated — the precise value together with the attribute or purpose "
        "it was tied to. Don't approximate or generalise; if several "
        "similar values exist, give the one matching exactly what's asked."
    ),
    "instruction_following": (
        "The user earlier set a STANDING INSTRUCTION on how to answer this "
        "kind of request; that governing requirement lives in the prior "
        "conversation, not in this question. Find it first, then answer so "
        "both its form and its content comply — substance alone, ignoring "
        "the instruction, is not enough."
    ),
    "abstention": (
        "This may have no answer in the conversation. The question's "
        "qualifiers are binding: it asks for a specific thing under "
        "specific conditions. If, after genuinely searching, no turn "
        "states that thing under the exact conditions asked, the "
        "information isn't there — say so plainly. The same subject "
        "described under DIFFERENT conditions does NOT answer it, and a "
        "merely related topic is NOT the answer either. An explicit 'not "
        "enough information' is itself correct; never generalise from a "
        "near-miss or invent specifics to fill the gap."
    ),
    "contradiction_resolution": (
        "The conversation says conflicting things about this. A correct "
        "answer must both ACKNOWLEDGE the conflict and RESOLVE it — the "
        "user's own correction or their latest statement normally wins. "
        "Avoid a flat yes/no that hides the contradiction or commits to "
        "the superseded side."
    ),
    "event_ordering": (
        "List the events in the ORDER they happened. If a number is "
        "asked, give that many DISTINCT events spread across the whole "
        "period — different topics, not several facets of one thread, and "
        "not only the earliest. Both which events you choose and their "
        "order matter."
    ),
    "knowledge_update": (
        "This value CHANGED over time, so the answer is the most RECENT "
        "one. The first mention is likely outdated — gather the mentions, "
        "order them, and commit to the latest as current without calling "
        "it old. Don't report a superseded value or claim 'no change' "
        "when a later one exists."
    ),
    "multi_session_reasoning": (
        "The answer isn't in any single turn — gather the separate facts "
        "it depends on, then chain or compute them. Retrieve every "
        "required piece before reasoning; a partial answer from one place "
        "fails. For a count, identify each genuine instance and avoid "
        "both over-counting near-misses and under-counting distinct ones."
    ),
    "preference_following": (
        "The user established a PREFERENCE earlier — a chosen tool, "
        "version, style, or constraint — that your answer must respect, "
        "even if it isn't restated in this question. Surface that prior "
        "choice and ground your answer in it rather than giving generic "
        "advice. If you can't find an explicit preference statement, it "
        "may simply not be phrased as one: a preference is often carried "
        "implicitly by a concrete value the user gave in their own "
        "example (a number, a version, a tool). So rather than hunting "
```

```
        "indefinitely, check whether something you've already retrieved "
        "is the relevant choice for what's being asked — and if it "
        "clearly fits, ground your answer in it."
    ),
    "summarization": (
        "Give a faithful, concise synthesis where COVERAGE matters — "
        "span the main components, stages, and alternatives weighed, not "
        "just the easiest-to-recall thread. Outline the whole arc first, "
        "then add only the specifics worth quoting."
    ),
    "temporal_reasoning": (
        "This turns on timing. Resolve the referenced time precisely (a "
        "named deadline is a specific date, not a vague window) and "
        "anchor BOTH endpoints on the correct events before computing "
        "any gap. Distinguish when something was SAID from when it is "
        "DUE or happened; the usual error is anchoring on the wrong "
        "date, not the arithmetic."
    ),
```

LongMemEval rubric

```
The user's benchmark conversation history is stored in your durable
history as rows with kind='context_msg' for user turns and
kind='model_turn' for assistant turns.  Every turn's content opens
with a [Session N | YYYY-MM-DD] role:  tag.  session_id names the
session, and created_at holds the row timestamp as ISO-8601 text.
Recall it with the recall_history_python tool:  pass a Python cell
that uses the pre-bound ms surface, with kind='context_msg' for user
evidence or kind='model_turn' for assistant context in ms.search and
AND ch.kind IN ('context_msg', 'model_turn') in SQL so that only this
history is read.

GROUNDING (strict):  treat the user's own turns (kind='context_msg')
as evidence of the user's facts, actions and preferences.  Every
concrete claim --- each fact, number, name, date, quantity, or event
--- must come verbatim in meaning from a turn you retrieved; an
assistant suggestion is not evidence that the user adopted it.
Preserve exact numbers, units and version labels from the most
directly relevant user evidence; do not replace them with illustrative
values, and do not add repeated historical mentions as separate
quantities unless the question explicitly asks for that.  Facts can
change across sessions:  when one did, use the latest user evidence
only after confirming that the turns describe the same project and the
same fact, not merely similarly named work.  Never invent specifics to
make an answer sound complete.

FINISH: answer as soon as the evidence is sufficient; a reply with no
tool call is your final answer.  Base it only on turns you actually
retrieved.  Follow any output-count or formatting constraint in the
question exactly.  If the record does not contain the requested fact
after a genuine search, abstain with the exact phrase "I don't have
that information from our conversations." rather than guessing.

[question]

[hint]

Turn budget:  you have at most 20 turns for this question.  Every
reply that calls a tool uses one turn and your final answer needs the
last one, so make at most 19 tool rounds.  Plan your searches, combine
several lookups into one Python cell, and answer as soon as the
evidence is sufficient.  If the budget runs out before you answer,
your findings are discarded.
```

LongMemEval per-category type hints. The [hint] field in the LongMemEval rubric is instantiated with the corresponding entry in _QTYPE_POSTSCRIPT; abstention questions use _ABSTENTION_POSTSCRIPT:

Code Block 3: Per-category type hints used with the LongMemEval rubric.

```
_QTYPE_POSTSCRIPT = {
    "temporal-reasoning": (
        "Time-sensitive question. Pattern: work out the date range the "
        "question implies first, then filter on created_at "
        "(substr(created_at,1,10) BETWEEN ... / ORDER BY) BEFORE reading "
        "content — don't keyword-scan the whole haystack."
    ),
    "knowledge-update": (
        "Recency-sensitive question. The user may have stated multiple "
        "values over time; the MOST RECENT statement is the current truth. "
        "Pattern: order the matching turns by created_at (or seq) "
        "descending and take the latest value. Mentioning prior values is "
        "fine, but do not state an out-of-date value as current."
    ),
    "single-session-preference": (
        "Preference question. The literal subject may never have been "
        "discussed verbatim — that is the point. An exact keyword/SQL "
        "match will miss; widen with OR synonyms in ms.search to surface "
        "related preferences (likes, dislikes, constraints, recurring "
        "interests) and ground the recommendation in those. \"I have no "
        "information about <topic>\" is the wrong frame here."
    ),
}

_ABSTENTION_POSTSCRIPT = (
    "This may be an unanswerable question — the user may never have stated "
    "the relevant information. After >=2 distinct retrieval queries "
    "(different keywords AND different surfaces: ms.search and "
    "ms.sql_query) return nothing useful, abstain EXPLICITLY using the "
    "phrasing: \"I don't have that information from our conversations.\" "
    "A similar-but-different entity is not a match; abstain rather than "
    "substitute the closest match."
)
```