TRANSITION-PARAMETRIC MEMORY FOR LANGUAGE AGENTS: REVERSIBLE COMPILED MEMORY FIELDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon language agents can reuse past experience, but combining record-level editability with compact memory access remains challenging. Textual memories typically require retrieval and in-context processing, while experience absorbed into shared parameters can be difficult to audit or selectively remove. We formulate *transition-parametric memory*, which preserves each decision transition as an identifiable record with a separate parametric contribution. We introduce *Reversible Compiled Memory Fields* (RCMF) to aggregate these contributions into a fixed-shape field queried by the current agent state to modulate a frozen language model. Online field reads avoid textual replay and have computational cost independent of memory count. Individual records can be inserted or removed through additive updates without retraining the model or recompiling other records. Experiments on AppWorld, ALFWorld, and WebShop show favorable task performance against multiple baselines. Core implementation code is provided in Appendix D.

1 INTRODUCTION

Large language models (LLMs) increasingly act as agents that observe environments, take actions, and incorporate feedback (Yao et al., 2023). Their interaction histories record not only what was attempted, but also the preceding states and resulting observations, providing experience that can inform decisions on subsequent tasks.

A common approach stores trajectories or summaries as textual memories for later retrieval (Shinn et al., 2023; Zhao et al., 2024). These records remain human-readable and directly editable, but retrieved text competes with the current task for context and must be processed again at inference time. Retrieval also typically exposes only a small subset of the memory bank to each decision.

A complementary approach encodes experience into model parameters, adapters, or compact latent states, enabling memory access without replaying raw text (Das et al., 2024; Chen et al., 2025). However, when multiple experiences are absorbed through shared parameter updates, their individual contributions can be difficult to isolate, and selective removal may require further optimization.

Our goal is to preserve experience identity and provenance while enabling compact, state-conditioned memory access without textual replay. We use an *agent decision transition*—the pre-action state, executed action, and resulting observation—as the atomic memory unit. This granularity separates the heterogeneous decisions within a trajectory while retaining the context that more abstract summaries may omit.

We formulate *transition-parametric memory*: each transition remains an identifiable record while contributing separately to a parametric memory representation. To realize this formulation, we introduce *Reversible Compiled Memory Fields* (RCMF), a framework that sums independently compiled record contributions into a fixed-shape field. At each decision step, the current state reads a compact signal from the field to modulate a frozen LLM, without retrieving memory text or scanning individual records. After training, a record is inserted by adding its field contribution and removed by subtracting it, without re-encoding other records; the original records remain available for provenance and auditing. Figure 1 illustrates this pipeline.

We evaluate RCMF on AppWorld, ALFWorld, and WebShop against ReAct, ReMe, MemGen, and δ-mem, representing a no-memory control and procedural, generative latent, and fast-weight mem-

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.

ory systems, respectively. RCMF achieves the highest observed task success across the three benchmarks in our evaluations. Additional AppWorld experiments measure execution cost and the scaling of field-read latency and storage, while ablations and field- and record-level interventions examine state-conditioned addressing and memory specificity.

Our contributions are:

- We formulate *transition-parametric memory*, using identifiable state–action–observation transitions as the units of parametric experience memory. This preserves both record identity and decision context while enabling experience to be accessed parametrically rather than replayed as memory text.
- We introduce *Reversible Compiled Memory Fields* (RCMF), combining independently editable record contributions with a fixed-shape, state-conditioned field read for a frozen LLM without textual memory replay.
- We evaluate RCMF on three agent benchmarks and analyze its execution cost, computational and storage scaling. Ablations and controlled interventions further examine state–conditioned addressing and memory specificity.

2 Related Work

2.1 Explicit and Structured Memory

Retrieval-augmented generation supplies selected external text to a language model (Lewis et al., 2020). Agent memory systems extend this interface to observations, dialogue events, reflections, and distilled lessons (Park et al., 2023; Packer et al., 2024; Zhong et al., 2024; Shinn et al., 2023; Zhao et al., 2024). Later systems improve memory organization through reflection, linking, refinement, and learned management (Tan et al., 2025; Xu et al., 2025). Task-oriented approaches organize experience into workflows, subgoal summaries, state-specific cues, reusable procedures, and knowledge structures (Wang et al., 2025b; Hu et al., 2025; Wang et al., 2026a; Cao et al., 2026; Yang et al., 2026). These methods improve the organization and relevance of explicit memories, but using them still requires selecting and processing textual or symbolic content.

RCMF retains human-readable transition records for provenance and editing, but uses their compiled contributions rather than their text during inference, avoiding repeated textual processing and per-record runtime access.

2.2 Parametric and Latent Memory

Parameter-efficient methods provide compact adaptation mechanisms through adapters, prefixes, soft prompts, and low-rank updates (Houlsby et al., 2019; Li & Liang, 2021; Lester et al., 2021; Hu et al., 2022). These are not memory systems by themselves, but underpin later methods that store knowledge, demonstrations, or experience in auxiliary parameters. Model editing modifies selected behaviors through learned updates or targeted changes to factual associations (Mitchell et al., 2022a; Meng et al., 2022; 2023); semi-parametric and lifelong editing methods use side memories, routing, or constrained updates to improve locality and reduce interference (Mitchell et al., 2022b; Wang et al., 2024a; Fang et al., 2025).

Dedicated latent-memory systems support episodic updates and selective forgetting (Das et al., 2024), persistent memory pools (Wang et al., 2024b; 2025a), amortized context modulations (Tack et al., 2024), and context-generated adapters (Chen et al., 2025). Memory Decoder distills retrieval behavior into a parametric module (Cao et al., 2025), while sparsely accessed key–value parameters expand pretrained knowledge capacity (Lample et al., 2019; Berges et al., 2025).

For language agents, MemGen generates state-conditioned latent memory tokens through a trigger and memory weaver (Zhang et al., 2026); ParamMem learns parametric reflection patterns (Yao et al., 2026); and TMEM and agentic test-time training absorb episode experience into LoRA parameters (Ren et al., 2026; Wang et al., 2026b).

RCMF instead centers on independently attributable decision transitions: after training, each record can be compiled, added, or removed without updating the policy or re-encoding the remaining bank.

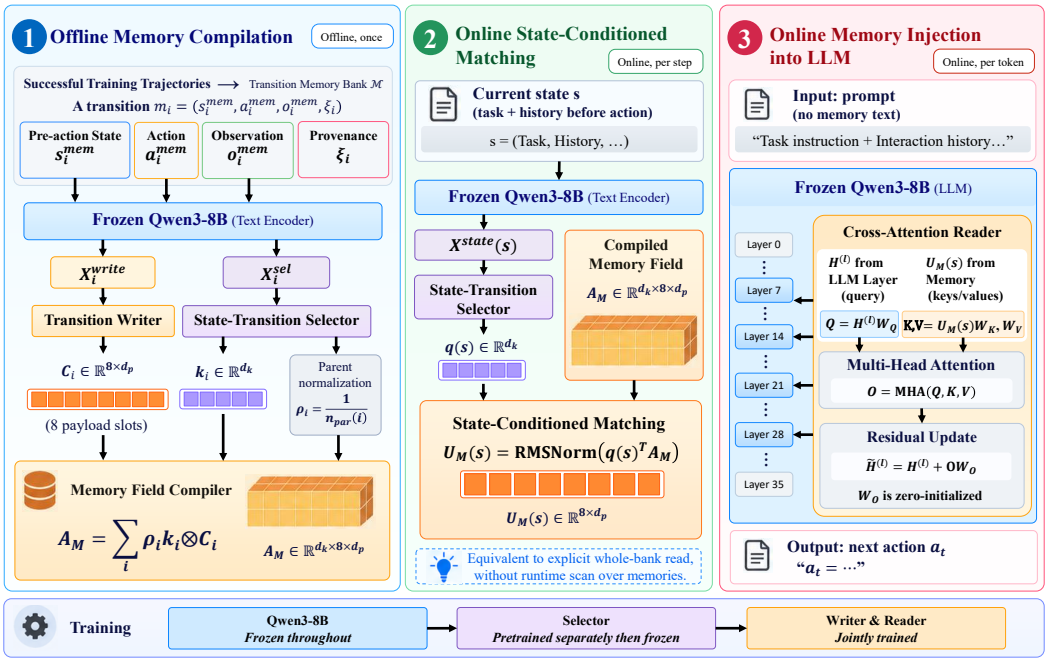

Figure 1: Overview of RCMF. Historical transitions are compiled offline into a fixed-shape memory field. At each decision step, the current state produces a query that reads eight state-conditioned memory slots from the field, which are injected into selected layers of the frozen LLM.

2.3 Fast Weights and Test-Time Learning

Fast-weight systems use rapidly updated auxiliary parameters to retain recent information (Schmidhuber, 1992; Ba et al., 2016). Linearized attention admits an outer-product fast-weight interpretation, with delta-rule and recurrent extensions enriching the update mechanism (Schlag et al., 2021; Irie et al., 2021). Test-Time Training generalizes the recurrent state to a model updated through self-supervised optimization (Sun et al., 2025), while Titans and ATLAS develop learned neural memories for long contexts (Behrouz et al., 2025b;a). More directly related, δ-mem uses an online delta-rule associative state to modulate a frozen attention backbone (Lei et al., 2026).

RCMF shares the associative outer-product representation, but differs in the unit and lifecycle of memory: persistent cross-task decision transitions contribute independently to a compiled field rather than being absorbed sequentially into a token- or chunk-level state. Its focus is transition-level provenance and reversible record editing with a fixed-size read, not a new attention operation.

3 Method

3.1 Problem Setting and Transition Memory Bank

We consider a frozen Qwen3-8B language-model policy π_θ. At decision step t, the state s_t contains the task specification and interaction history available before the current action. The policy generates a_t, receives observation o_t, and forms the next state as $s_{t+1} = \text{Update}(s_t, a_t, o_t)$.

RCMF stores each decision transition as a structured memory record m_i. The transition memory bank is

$$\mathcal{M} = \{m_i\}_{i=1}^{N}, \qquad m_i = (s_i^{\text{mem}}, a_i^{\text{mem}}, o_i^{\text{mem}}, \xi_i)\,, \tag{1}$$

where s_i^{mem}, a_i^{mem}, and o_i^{mem} are the historical pre-action state, complete action, and resulting observation, respectively. The metadata ξ_i records provenance, including task, episode, trajectory, step index, replay lineage, and record identity. Both current state s_t and stored state s_i^{mem} are pre-action decision states, referring to the current and historical interactions, respectively.

Thus, m_i is an experience record, not a tensor produced by the memory model. In our experiments, the memory bank is constructed from successful training trajectories. The original structured records remain the authoritative memory ledger throughout the system's lifetime.

3.2 RCMF Overview

Figure 1 shows the RCMF pipeline. The framework separates two questions about a historical transition. First, when should the transition influence the current decision? A **State-Transition Selector** maps each stored transition m_i to a key $k_i \in \mathbb{R}^{d_k}$ and the current state s to a query $q(s) \in \mathbb{R}^{d_k}$. Their inner product $q(s)^\top k_i$ controls the strength and sign of the transition's contribution. Second, what information should the transition contribute? A **Transition Writer** maps its semantic content to eight payload slots $C_i \in \mathbb{R}^{8\times d_p}$.

The **Field Compiler** combines these key–payload pairs (k_i, C_i) into a fixed-shape memory field. Each transition contributes an additive tensor ΔA_i, and the complete bank is represented by

$$\Delta A_i \propto k_i \otimes C_i, \qquad A_{\mathcal{M}} = \sum_i \Delta A_i \in \mathbb{R}^{d_k\times 8\times d_p}, \tag{2}$$

where $\otimes$ denotes the outer product. Section 3.5 specifies the per-record weighting.

At each decision step, the Selector obtains $q(s)$ from the current state. Contracting this query with the field and normalizing each payload slot produces

$$U_{\mathcal{M}}(s) = \mathrm{RMSNorm}\left(q(s)^\top A_{\mathcal{M}}\right) \in \mathbb{R}^{8\times d_p}. \tag{3}$$

The **Cross-Attention Reader** integrates these slots into intermediate hidden states of the frozen language model. This online path requires neither top-k retrieval nor a scan of individual memory records. Algorithm 1 in Appendix C summarizes the procedure.

3.3 Multi-View State and Transition Representations

Each transition memory m_i is converted into a structured text sequence containing four semantic sections: the task goal x_i^{goal} and pre-action state x_i^{sta} extracted from s_i^{mem}, the complete action x_i^{act} from a_i^{mem}, and the resulting observation x_i^{obs} from o_i^{mem}. We denote this deterministic serialization by $x_i = \mathrm{Render}(m_i)$, which concatenates x_i^{goal}, x_i^{sta}, x_i^{act}, and x_i^{obs} under a canonical structured format. The complete sequence x_i is processed once by the frozen Qwen3-8B backbone. Let

$$H_i = H_\theta(x_i) = (h_1, \ldots, h_T) \in \mathbb{R}^{T\times d_{\mathrm{model}}}, \tag{4}$$

denote its final-layer token hidden states. Because the canonical rendering preserves the token boundaries of the four semantic sections, each section corresponds to a contiguous subsequence of H_i. For example, the hidden states corresponding to the action section are obtained simply by taking from H_i the positions occupied by x_i^{act}. We denote the four resulting hidden-state sequences by H_i^{goal}, H_i^{sta}, H_i^{act}, and H_i^{obs}.

For any hidden-state sequence $H = (h_1, \ldots, h_n)$, we use two complementary readouts,

$$\mathrm{MeanPool}(H) = \frac{1}{n}\sum_{h\in H} h, \qquad \mathrm{FinalPool}(H) = h_n. \tag{5}$$

Mean pooling summarizes information across the complete span, while the final-token hidden state provides a contextualized representation after the causal backbone has processed the preceding tokens. For each of the four semantic sections, we extract both readouts. For example, the action section contributes $\mathrm{MeanPool}(H_i^{\mathrm{act}})$ and $\mathrm{FinalPool}(H_i^{\mathrm{act}})$. Applying the same operations to the goal, pre-action state, action, and observation yields eight d_{model}-dimensional representations, collectively denoted as $X_i^{\mathrm{write}} \in \mathbb{R}^{8\times d_{\mathrm{model}}}$ and used as input to the **Writer**.

The Selector uses a slightly richer representation of the stored transition. It augments the eight section-level representations with the mean-pooled and final-token representations of the complete transition H_i, yielding $X_i^{\mathrm{sel}} \in \mathbb{R}^{10\times d_{\mathrm{model}}}$. On the current-state side, it analogously encodes only the task and interaction information available before the current action and uses the resulting representations $X^{\mathrm{state}}(s) \in \mathbb{R}^{10\times d_{\mathrm{model}}}$. These two representations form the inputs to the **Selector** introduced in Section 3.4. The construction of $X^{\mathrm{state}}(s)$ is detailed in Appendix A.1.

3.4 TRANSITION WRITER AND STATE-TRANSITION SELECTOR

Transition Writer. The Writer is a section-aligned feed-forward encoder that maps the eight rows of X_i^{write} to eight payload slots. Let $h_{i,e}^r \in \mathbb{R}^{d_{\text{model}}}$ denote the row of X_i^{write} associated with section $e \in \{\text{goal}, \text{sta}, \text{act}, \text{obs}\}$ and readout type $r \in \{\text{mean}, \text{final}\}$. Each section has its own LayerNorm and two affine maps, $f_{e,1} : \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{d_w}$ and $f_{e,2} : \mathbb{R}^{d_w} \to \mathbb{R}^{d_p}$, where d_w is the Writer hidden dimension. A learned embedding $\eta_r \in \mathbb{R}^{d_w}$ identifies the readout type and is shared across sections. The row-wise transformation and subsequent stacking are

$$z_{i,e}^r = f_{e,1}\left(\text{LN}_e\left(h_{i,e}^r\right)\right) + \eta_r \in \mathbb{R}^{d_w}, \qquad C_{i,e}^r = f_{e,2}\left(\text{GELU}\left(z_{i,e}^r\right)\right) \in \mathbb{R}^{d_p}, \tag{6}$$

Applying Equation 6 to the mean-pooled and final-token rows of all four semantic sections produces eight payload vectors. Stacking them in the same section–readout order as X_i^{write} gives

$$C_i = \text{Stack}_{e,r}\left(C_{i,e}^r\right) = W_\phi\left(X_i^{\text{write}}\right) \in \mathbb{R}^{8 \times d_p}. \tag{7}$$

Here $W_\phi\left(X_i^{\text{write}}\right)$ is compact notation for applying the Writer in Equation 6 to the eight rows of X_i^{write} and stacking the resulting payloads; it is not an additional transformation applied after the $C_{i,e}^r$ vectors are obtained. The exact Writer structure is detailed in Appendix A.2.

State-Transition Selector. The Selector ensembles $J = 3$ independently trained scoring models with the same architecture but separate parameters. For one model, omitting its index, let $x_v^{\text{state}}(s)$, $x_{i,w}^{\text{sel}} \in \mathbb{R}^{d_{\text{model}}}$ denote rows v and w of $X^{\text{state}}(s)$ and X_i^{sel}, where $v, w \in \{1, \ldots, 10\}$. Each view undergoes two bias-free linear projections: a view-specific projection $P_v^{\text{state}}, P_w^{\text{sel}} \in \mathbb{R}^{d_{\text{proj}} \times d_{\text{model}}}$ followed by a side-shared projection $R_{\text{state}}, R_{\text{sel}} \in \mathbb{R}^{d_{\text{int}} \times d_{\text{proj}}}$, where d_{proj} and d_{int} are the intermediate and interaction dimensions. The state and transition sides use separate shared projections:

$$\begin{aligned} \hat{x}_v^{\text{state}} &= R_{\text{state}} P_v^{\text{state}} x_v^{\text{state}}(s) \in \mathbb{R}^{d_{\text{int}}}, \\ \hat{x}_{i,w}^{\text{sel}} &= R_{\text{sel}} P_w^{\text{sel}} x_{i,w}^{\text{sel}} \in \mathbb{R}^{d_{\text{int}}}. \end{aligned} \tag{8}$$

A learned tensor $G \in \mathbb{R}^{10 \times 10 \times d_{\text{int}}}$ weights interactions between all state–transition view pairs. With $\odot$ denoting element-wise multiplication, the model's raw score is

$$\text{score}_{\text{raw}}(s, m_i) = \frac{1}{\sqrt{10 \cdot 10 \cdot d_{\text{int}}}} \sum_{v=1}^{10} \sum_{w=1}^{10} \left(\hat{x}_v^{\text{state}}\right)^\top \left(G_{v,w,:} \odot \hat{x}_{i,w}^{\text{sel}}\right). \tag{9}$$

Each view pair contributes a learned weighted inner product. The J models are pretrained on training-only state–transition supervision; their raw scores are standardized using training-set statistics and averaged. Model parameters and calibration statistics are frozen during Writer–Reader optimization. Since each score is bilinear, the calibrated ensemble admits the exact decomposition

$$\text{score}(s, m_i) = q(s)^\top k_i + b, \qquad q(s), k_i \in \mathbb{R}^{d_k}, \tag{10}$$

where d_k is the common query/key dimension and b is a global calibration intercept. The query concatenates projected state views across models, whereas the key combines projected transition views with the interaction tensors and calibration scales without using the current state. Hence $q(s)$ depends only on $X^{\text{state}}(s)$ and k_i only on X_i^{sel}. Appendix A.3 gives the implementation dimensions and derives these factors. RCMF uses the bilinear component as its continuous state–transition coefficient, without discrete selection or top-k retrieval:

$$\alpha(s, m_i) = \text{score}(s, m_i) - b = q(s)^\top k_i. \tag{11}$$

3.5 FIELD COMPILATION AND STATE-CONDITIONED READ

Groups of related memories can contain different numbers of transitions, so larger groups would otherwise contribute more terms simply because they contain more records. We define normalization groups from the provenance metadata ξ_i introduced in Section 3.1. Let $\text{par}(i)$ denote the group of memory m_i, and let $n_{\text{par}(i)}$ be the number of records in that group when the record weights ρ_i are assigned. Before slot-wise RMS normalization, an explicit read at state s would be

$$\sum_{i=1}^{N} \rho_i \alpha(s, m_i) C_i = \sum_{i=1}^{N} \rho_i \left(q(s)^\top k_i\right) C_i \in \mathbb{R}^{8 \times d_p}, \qquad \rho_i = \frac{1}{n_{\text{par}(i)}}. \tag{12}$$

Thus, each group has unit total base weight when ρ_i is assigned, before applying the state-dependent coefficient $\alpha(s, m_i)$. Direct evaluation of Equation 12 scans the memory bank at every decision step.

The **Field Compiler** avoids this per-step scan by compiling the weighted key–payload outer products into a single fixed-shape tensor:

$$A_{\mathcal{M}} = \sum_{i=1}^{N} \rho_i k_i \otimes C_i \in \mathbb{R}^{d_k \times 8 \times d_p}, \tag{13}$$

$$U_{\mathcal{M}}(s) = \mathrm{RMSNorm}\left(q(s)^\top A_{\mathcal{M}}\right) \in \mathbb{R}^{8 \times d_p}. \tag{14}$$

Here $\otimes$ denotes the outer product, and RMS normalization is applied independently across the d_p coordinates of each slot. By distributivity, the unnormalized contraction $q(s)^\top A_{\mathcal{M}}$ equals the explicit sum in Equation 12. The serving shape of $A_{\mathcal{M}}$ is therefore independent of N, while the archival memory ledger grows with the number of records. Appendix A.4 gives the exact tensor algebra, normalization, and implementation dimensions.

3.6 Cross-Attention Reader

The field read $U_{\mathcal{M}}(s)$ provides eight state-conditioned memory slots. The **Cross-Attention Reader** injects them after selected Transformer layers without extending the language-model prompt. Let $H^{(\ell)} \in \mathbb{R}^{T_{\mathrm{ctx}} \times d_{\mathrm{model}}}$ denote language model hidden states after layer ℓ, where T_{ctx} counts token positions processed in the current forward pass. With learned Reader projections $W_Q^{(\ell)}, W_K^{(\ell)}, W_V^{(\ell)}$, and $W_O^{(\ell)}$, and separate LayerNorms for the hidden states and memory slots, the Reader computes

$$\begin{gathered} Q^{(\ell)} = \mathrm{LN}_Q^{(\ell)}\left(H^{(\ell)}\right) W_Q^{(\ell)}, \qquad K^{(\ell)} = \mathrm{LN}_M^{(\ell)}\left(U_{\mathcal{M}}(s)\right) W_K^{(\ell)}, \qquad V^{(\ell)} = \mathrm{LN}_M^{(\ell)}\left(U_{\mathcal{M}}(s)\right) W_V^{(\ell)}, \\ \widetilde{H}^{(\ell)} = H^{(\ell)} + \mathrm{MHA}\left(Q^{(\ell)}, K^{(\ell)}, V^{(\ell)}\right) W_O^{(\ell)}. \end{gathered} \tag{15}$$

Here d_{attn} is the total Reader attention dimension. The language-model states supply the queries, while the memory slots supply the keys and values. The output projection $W_O^{(\ell)}$ is initialized to zero, so the initial Reader update is zero and the augmented model reproduces the frozen language model. The memory slots are neither prompt tokens nor entries in the language model's self-attention KV cache. Appendix A.5 gives the attention structure, dimensions, and execution details.

3.7 Record-Level Editing and Task-Legal Fields

The additive field gives each memory an attributable contribution ΔA_i. Let $\tau(s)$ denote the task identity of the current state, A_{all} the field formed from all memories otherwise eligible for that state, and $A_{\tau(s)}$ the summed contribution from the current task. Then

$$\Delta A_i = \rho_i k_i \otimes C_i, \tag{16}$$

$$A_{\mathrm{legal}}(s) = A_{\mathrm{all}} - A_{\tau(s)}. \tag{17}$$

A record is inserted or removed by adding or subtracting ΔA_i; replacement subtracts the original contribution before adding the replacement. These edits do not require recompiling unrelated memories. During Writer–Reader optimization and validation, $A_{\mathrm{legal}}(s)$ prevents the current task from reading its own stored transitions. For a new task absent from the memory bank, $A_{\tau(s)} = 0$ and the complete otherwise eligible field is used. Provenance metadata ξ_i determines memory eligibility before compilation. Appendix A.6 details the editing semantics, provenance handling, and differentiable construction of task-legal fields.

3.8 Joint Writer–Reader Optimization

The Qwen3-8B policy and pretrained Selector remain frozen while the Writer and Reader are optimized jointly. For each training state, the Writer computes payloads for the complete eligible memory bank, from which the task-legal field $A_{\mathrm{legal}}(s)$ is constructed differentiably. The frozen Selector produces the state-dependent query $q(s)$, which reads this field into eight memory slots. Supervision on the resulting policy output backpropagates through the Reader, field read, and Writer; auxiliary

Table 1: End-to-end performance on the fixed evaluation sets. Higher is better.

Method	WebShop Reward	WebShop Full Success	AppWorld Success	ALFWorld Success	Average Success
ReAct	0.5803	39/200 (19.50%)	32/168 (19.05%)	43/134 (32.09%)	23.55%
ReMe	0.5819	44/200 (22.00%)	38/168 (22.62%)	43/134 (32.09%)	25.57%
MemGen	0.5728	36/200 (18.00%)	34/168 (20.24%)	50/134 (37.31%)	25.18%
δ-mem	0.6017	45/200 (22.50%)	41/168 (24.40%)	47/134 (35.07%)	27.33%
RCMF	**0.6320**	**51/200 (25.50%)**	**48/168 (28.57%)**	**52/134 (38.81%)**	**30.96%**

penalties regularize the Reader residual and Writer payload magnitude. Every training forward uses the complete task-legal field, and no raw memory text is added to the policy prompt. Exact training targets, loss terms, optimization settings, and checkpoint selection are given in Appendix B.3.

After optimization, the Writer and Reader are fixed. Each eligible transition is compiled once into its record contribution ΔA_i, and the deployment field is obtained by summing these contributions. The field is reused across decision steps until a record is edited; runtime then requires only the current-state query, compiled-field read, and Reader.

4 Experiments

4.1 Experimental Setup

Benchmarks. AppWorld (Trivedi et al., 2024) tests multi-step reasoning via code-based API interactions across simulated applications. **ALFWorld** (Shridhar et al., 2021) tests sequential decision making in household environments via textual actions and observations. **WebShop** (Yao et al., 2022; Liu et al., 2024) tests search, navigation, and attribute-constrained shopping; we use the AgentBench-FC runtime and its 200-task standard evaluation set. Memory construction and training use training-side experience, excluding evaluation trajectories and outcomes from memory and supervision. Appendix B.1 gives data splits, runtime details, and evaluation-history qualifications.

Baselines. ReAct (Yao et al., 2023) retains the current-episode history without cross-task memory. **ReMe** (Cao et al., 2026) represents explicit and structured procedural memory. **MemGen** (Zhang et al., 2026) represents generative latent agent memory through a learned trigger and memory weaver that generate state-conditioned latent memory tokens. **δ-mem** (Lei et al., 2026) represents fast-weight memory through an online associative state updated by a delta rule. All methods use Qwen3-8B and share the evaluation tasks, environments, and evaluators within each benchmark. We rerun the baselines in these environments rather than use published scores from different models or runtimes. Appendix B.2 describes the adaptations.

RCMF configuration and evaluation. RCMF uses the architecture in Section 3 with a frozen Qwen3-8B backbone. Each transition contributes eight 256-dimensional payload slots. The Selector query/key dimension is 960, giving a compiled field of shape $960 \times 8 \times 256$. Readers use eight heads with total attention dimension 512 after zero-indexed Qwen layers 7, 14, 21, and 28. Runtime memory access uses the compiled field without raw-memory prompt insertion, retrieval, or per-record scanning. Appendix B.3 gives implementation, training, and evaluation details.

4.2 Effectiveness

Table 1 reports task success on AppWorld and ALFWorld, and mean reward plus full success (reward 1.0) on WebShop. Average success is the unweighted mean of the three task-success rates.

RCMF shows the strongest overall performance across the three benchmarks, suggesting that its design is effective across different interaction structures rather than specific to one environment. Other memory mechanisms do not improve uniformly over ReAct: MemGen improves on ALFWorld but underperforms on WebShop, while ReMe improves on AppWorld and WebShop but not ALFWorld. These differences suggest that adding memory alone does not ensure consistent gains, motivating RCMF's emphasis on conditioning stored experience on the current decision state.

Table 2: End-to-end execution cost on the 168-task AppWorld evaluation.

Method	Total Steps	Evaluation Wall Time (h)	Wall Time / Step (s)
ReAct	4,791	5.20	3.91
ReMe	5,071	7.18	5.10
MemGen	4,398	11.25	9.21
δ-mem	4,529	10.59	8.42
RCMF	4,800	6.14	4.61

Table 3: Ablation study. Δ denotes the change in AppWorld success rate relative to Full RCMF.

Variant	Success	Δ vs.Full
Full RCMF	48/168 (28.57%)	–
Mean pooling only	38/168 (22.62%)	-5.95 pp
Uniform record weights	41/168 (24.40%)	-4.17 pp
State-independent addressing	27/168 (16.07%)	-12.50 pp

4.3 Efficiency

Table 2 reports steps and wall time for the 168-task AppWorld evaluations. Because different memory mechanisms can change the generated trajectories, these measurements include behavior-dependent differences in trajectory length, generated outputs, and environment interaction; they therefore characterize realized agent execution cost rather than isolating memory-access latency. Controlled efficiency measurements are provided in Appendix B.4.

RCMF adds moderate overhead over ReAct but remains less expensive per decision step than the memory baselines. This is consistent with amortizing transition encoding and field compilation across decisions: online memory use requires current-state querying, a fixed-size field read, and Reader operations, rather than textual memory replay, latent-memory generation, or fast-weight updates. Together with the task-performance results and controlled measurements, these findings support offline compilation and online reuse as a practical way to balance memory effectiveness and execution cost.

4.4 Ablation Study

We ablate three RCMF design choices, fixing each intervention before final evaluation and retraining the affected trainable components under the same protocol.

Mean pooling only removes final-token information by replacing every final-token view with a copy of its corresponding mean-pooled view while preserving tensor shapes and downstream interfaces. The affected Selector and Writer–Reader components are retrained under the same protocol.

Uniform record weights removes provenance-based normalization by setting $\rho_i = 1$ for every eligible record. The frozen Selector is retained, while the Writer and Reader are retrained using the modified field throughout training and deployment.

State-independent addressing replaces the current-state query $q(s)$ with a fixed query $\bar{q}$, the arithmetic mean of frozen Selector queries over the 366 Writer–Reader training states. The same $\bar{q}$ is used for training and evaluation, while the Reader still processes state-dependent language-model hidden states. The Writer and Reader are retrained under this fixed-query read.

All three interventions reduce the observed AppWorld success rate in Table 3. Among the three tested design choices, the results therefore identify state-conditioned addressing as having the largest empirical contribution in the fixed AppWorld configuration, while dual mean/final representations and record normalization provide smaller positive contributions.

Table 4: Memory-specificity analysis on AppWorld. Field rows report task success; record rows report mean/median change in successful-action log-probability after inserting the selected record.

Level	Condition	Result
Field	Correct	48/168 (28.57%)
Field	Key–payload shuffle	42/168 (25.00%)
Field	State-query shuffle	39/168 (23.21%)
Record	Related	$+0.1901/+0.1064$
Record	Unrelated	$+0.0025/+0.0009$

4.5 Mechanistic Analysis

We examine state–memory specificity at both the **Field** and **Record** levels. At the field level, we perturb state–memory correspondence without retraining.

Key–Payload shuffle deterministically reassigns payloads C_i across fixed keys k_i while preserving the same sets of keys, payloads, record weights, and Reader parameters.

State-Query shuffle instead keeps the compiled field fixed but replaces the current-state query $q(s)$ with a query from a different held-out training state.

These controls respectively break the key–payload binding $k_i \to C_i$ and the state–query correspondence $s \to q(s)$ in the normal addressing chain $s \to q(s) \to k_i \to C_i$. Their lower success in Table 4 supports that RCMF depends on the learned correspondence among state queries, keys, and payloads rather than merely on the aggregate content stored in the field.

At the record level, we isolate individual contributions using 20 related and 20 unrelated state–memory pairs that are unseen during training. For pair u, let s_u denote the current state, i_u the index of the selected memory record, and a_u^* the recorded successful action. Let A_u^- contain all other eligible records and $A_u^+ = A_u^- + \Delta A_{i_u}$. Let $p_{\mathrm{RCMF}}(a \mid s, A)$ denote the probability assigned to action sequence a by the RCMF-conditioned language model under field A. We measure

$$\Delta_u = \log p_{\mathrm{RCMF}}(a_u^* \mid s_u, A_u^+) - \log p_{\mathrm{RCMF}}(a_u^* \mid s_u, A_u^-). \tag{18}$$

A positive value means that inserting the selected record increases the likelihood assigned to the recorded successful action. Related records produce a positive shift, whereas unrelated records remain close to zero. This separation is consistent with record-specific influence: relevant records steer the policy toward the successful action without a comparable directional effect from unrelated records.

5 Conclusion

We introduced *transition-parametric memory*, in which agent decision transitions remain identifiable records while contributing separately to a parametric memory representation. RCMF realizes this formulation by compiling transition-specific key–payload contributions into a fixed-shape field queried by the current state and injected into a frozen language model. The resulting interface avoids textual memory replay and per-record runtime scanning, while additive field contributions support record-level insertion and removal without retraining the model or recompiling unrelated memories.

Across AppWorld, ALFWorld, and WebShop, RCMF performs favorably against explicit, latent, and fast-weight memory baselines. Efficiency measurements show that field-read shape and active tensor storage are independent of memory-bank size, while ablations and controlled interventions provide evidence that performance depends on state-conditioned addressing and correctly associated transition content. These results suggest that separately editable transition records and compact parametric access can coexist in a unified agent-memory interface. The present study is limited to a single frozen backbone, three agent benchmarks, and the memory-bank scales evaluated here. Future work should test substantially larger and continually growing memory banks, additional backbones and agent domains, and alternative compilation and Reader architectures.

AI USE STATEMENT

We used generative AI tools for polishing writing, summarizing literature, and implementing and debugging code. We have not used generative AI tools for proposing ideas, drafting sections of the paper, generating datasets, proving mathematical claims, or generating scientific claims; all of these decisions and the final manuscript content were made entirely by the authors.

We have reviewed all AI-assisted work. AI-assisted code and analyses were reviewed against the implemented system and experimental artifacts, and literature and citation claims used in the paper were checked against their original sources. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Our experiments use simulated agent environments and existing benchmark data; we conduct no human-subject study and deploy no system to real users. In real-world deployments, however, persistent agent memories may store sensitive, erroneous, or adversarial interaction records that can influence future actions. RCMF's provenance tracking and record-level removal improve auditability and controllability but do not by themselves address privacy, poisoning, security, or misuse risks. Practical deployment would additionally require appropriate data minimization, access control, consent and retention policies, write validation, and independent verification of requested deletions.

REPRODUCIBILITY STATEMENT

The mathematical definition of RCMF and its update, read, and editing procedures are provided in Section 3. Appendix A gives architecture, tensor-algebra, and implementation details; Appendix B documents benchmark settings, baseline adaptations, training and evaluation protocols, and efficiency measurements; Appendix C provides the benchmark-independent procedure; and Appendix D provides the corresponding core implementation.

REFERENCES

Jimmy Ba, Geoffrey E. Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Ali Behrouz, Zeman Li, Praneeth Kacham, Majid Daliri, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. ATLAS: Learning to optimally memorize the context at test time. *arXiv preprint arXiv:2505.23735*, 2025a.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b.

Vincent-Pierre Berges, Barlas Oguz, Daniel Haziza, Wen-Tau Yih, Luke Zettlemoyer, and Gargi Ghosh. Memory layers at scale. In *International Conference on Machine Learning (ICML)*, 2025.

Jiaqi Cao, Jiarui Wang, Rubin Wei, Qipeng Guo, Kai Chen, Bowen Zhou, and Zhouhan Lin. Memory decoder: A pretrained, plug-and-play memory for large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

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.

Tong Chen, Hao Fang, Patrick Xia, Xiaodong Liu, Benjamin Van Durme, Luke Zettlemoyer, Jianfeng Gao, and Hao Cheng. Generative adapter: Contextualizing language models in parameters with a single forward pass. In *International Conference on Learning Representations (ICLR)*, 2025.

Payel Das, Subhajit Chaudhury, Elliot Nelson, Igor Melnyk, Sarathkrishna Swaminathan, Sihui Dai, Aurelie Lozano, Georgios Kollias, Vijil Chenthamarakshan, Jiri Navratil, Soham Dan, and Pin-Yu Chen. Larimar: Large language models with episodic memory control. In *International Conference on Machine Learning (ICML)*, 2024.

Junfeng Fang, Houcheng Jiang, Kun Wang, Yunshan Ma, Jie Shi, Xiang Wang, Xiangnan He, and Tat-Seng Chua. AlphaEdit: Null-space constrained knowledge editing for language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning (ICML)*, 2019.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Jingdi Lei, Di Zhang, Junxian Li, Weida Wang, Kaixuan Fan, Xiang Liu, Qihan Liu, Xiaoteng Ma, Baian Chen, and Soujanya Poria. δ-mem: Efficient online memory for large language models. *arXiv preprint arXiv:2605.12357*, 2026.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Xiang Lisa Li and Percy Liang. Prefix-Tuning: Optimizing continuous prompts for generation. In *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, 2021.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Kevin Meng, Arnab Sen Sharma, Alex J. Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer. In *International Conference on Learning Representations (ICLR)*, 2023.

Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D. Manning. Fast model editing at scale. In *International Conference on Learning Representations (ICLR)*, 2022a.

Eric Mitchell, Charles Lin, Antoine Bosselut, Christopher D. Manning, and Chelsea Finn. Memory-based model editing at scale. In *International Conference on Machine Learning (ICML)*, 2022b.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2024.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023.

Tao Ren, Weiyao Luo, Hui Yang, Rongzhi Zhu, Xiang Huang, Yuchuan Wu, Bingxue Chou, Jieping Ye, Jiafeng Liang, Yongbin Li, and Yijie Peng. Scaling self-evolving agents via parametric memory. *arXiv preprint arXiv:2606.04536*, 2026.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning (ICML)*, 2021.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations (ICLR)*, 2021.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (Learn at test time): RNNs with expressive hidden states. In *International Conference on Machine Learning (ICML)*, 2025.

Jihoon Tack, Jaehyung Kim, Eric Mitchell, Jinwoo Shin, Yee Whye Teh, and Jonathan Richard Schwarz. Online adaptation of language models with a memory of amortized contexts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, Anand Rajan Iyer, Tianlong Chen, Huan Liu, Chen-Yu Lee, and Tomas Pfister. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Peng Wang, Zexi Li, Ningyu Zhang, Ziwen Xu, Yunzhi Yao, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. WISE: Rethinking the knowledge memory for lifelong model editing of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024a.

Tong Wang, Pei Xu, Shiyue Cao, Likun Yang, Daipeng Li, Jianbin Jiao, and Kaiqi Huang. SAMem: State-aware memory as a fine-grained memory for llm agents in decision-making. In *Findings of the Association for Computational Linguistics (ACL)*, 2026a.

Yanbo Wang, Jinhua Hao, Yuze Shi, Kun Yuan, and Ming Sun. No time like the present: Agentic test-time training for llm agents. *arXiv preprint arXiv:2607.03441*, 2026b.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, Jingbo Shang, and Julian McAuley. MEMORYLLM: Towards self-updatable large language models. In *International Conference on Machine Learning (ICML)*, 2024b.

Yu Wang, Dmitry Krotov, Yuanzhe Hu, Yifan Gao, Wangchunshu Zhou, Julian McAuley, Dan Gutfreund, Rogerio Feris, and Zexue He. M+: Extending MemoryLLM with scalable long-term memory. In *International Conference on Machine Learning (ICML)*, 2025a.

Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning (ICML)*, 2025b.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Ke Yang, Zixi Chen, Xuan He, Jize Jiang, Michel Galley, Chenglong Wang, Jianfeng Gao, Jiawei Han, and ChengXiang Zhai. PlugMem: A task-agnostic plugin memory module for llm agents. In *International Conference on Machine Learning (ICML)*, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

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

Tianjun Yao, Yongqiang Chen, Yujia Zheng, Pan Li, Zhiqiang Shen, and Kun Zhang. ParamMem: Augmenting language agents with parametric reflective memory. In *International Conference on Machine Learning (ICML)*, 2026.

Guibin Zhang, Muxin Fu, and Shuicheng Yan. Memgen: Weaving generative latent memory for self-evolving agents. In *International Conference on Learning Representations (ICLR)*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: Llm agents are experiential learners. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

A RCMF Implementation Details

A.1 Current-State Representation

Section 3.3 defines the transition representations X_i^{write} and X_i^{sel} extracted from a stored transition. The current state is represented in the same way: a structured target-free state text is passed once through the frozen Qwen3-8B backbone, semantic token spans are identified in the resulting final-layer hidden states, and mean-pooled and final-token readouts are extracted from each span. We use five state spans and therefore obtain

$$X^{\text{state}}(s) \in \mathbb{R}^{10\times d_{\text{model}}}. \tag{19}$$

The semantic content of the five spans is defined by the benchmark adapter, but each span is represented by the same pair of mean-pooled and final-token readouts. In all cases, the state representation contains only information available before the current action and excludes the target action and future observations. The reported implementation uses $d_{\text{model}} = 4096$. The resulting $X^{\text{state}}(s)$ is used together with X_i^{sel} by the Selector in Section 3.4.

A.2 Transition Writer Architecture

Section 3.4 defines the Writer as a section-aligned mapping from $X_i^{\text{write}} \in \mathbb{R}^{8\times d_{\text{model}}}$ to $C_i \in \mathbb{R}^{8\times d_p}$. In the reported implementation, $d_{\text{model}} = 4096$, $d_w = 512$, and $d_p = 256$. For each semantic section $e \in \{\text{goal}, \text{sta}, \text{act}, \text{obs}\}$, the two corresponding rows of X_i^{write} are

$$h_{i,e}^{\text{mean}} = \text{MeanPool}\,(H_i^e) \in \mathbb{R}^{4096}, \qquad h_{i,e}^{\text{final}} = \text{FinalPool}\,(H_i^e) \in \mathbb{R}^{4096}. \tag{20}$$

Thus, the eight rows of X_i^{write} are precisely the vectors $h_{i,e}^r$ obtained from the four semantic sections and the two readout types.

The Writer contains four independent section-specific branches, one for each e. Within branch e, both $h_{i,e}^{\text{mean}} \in \mathbb{R}^{4096}$ and $h_{i,e}^{\text{final}} \in \mathbb{R}^{4096}$ are first processed by the same section-specific LayerNorm LN_e and the same affine map $f_{e,1} : \mathbb{R}^{4096} \to \mathbb{R}^{512}$. This produces $f_{e,1}\left(\text{LN}_e\left(h_{i,e}^r\right)\right) \in \mathbb{R}^{512}$. The Writer then adds the readout-type embedding $\eta_r \in \mathbb{R}^{512}$, using η_{mean} for the mean-pooled row and η_{final} for the final-token row. The two embeddings $\eta_{\text{mean}}, \eta_{\text{final}} \in \mathbb{R}^{512}$ are shared across all four section branches. After GELU, the same section-specific second affine map $f_{e,2} : \mathbb{R}^{512} \to \mathbb{R}^{256}$ produces $C_{i,e}^r \in \mathbb{R}^{256}$.

The two readouts of a semantic section share that section's normalization and feed-forward parameters but remain distinguishable through η_r, whereas different sections use different LayerNorm and affine parameters. The four branches produce eight vectors $C_{i,e}^r \in \mathbb{R}^{256}$, which are stacked in the same section–readout order as the rows of X_i^{write} to form $C_i \in \mathbb{R}^{8\times 256}$. No additional transformation follows this stacking: $W_\phi\left(X_i^{\text{write}}\right)$ in Equation 7 denotes the complete four-branch computation. The Writer itself does not mix different semantic sections. Its inputs may already contain information from preceding sections through the frozen causal backbone.

A.3 Selector Implementation and Exact Query–Key Decomposition

Section 3.4 defines the Selector's member architecture and raw score in Equations 8 and 9. The implementation uses $d_{\text{model}} = 4096$, $d_{\text{proj}} = 64$, and $d_{\text{int}} = 32$. All $J = 3$ members use these dimensions but have independently learned parameters. This subsection derives the query and key vectors in Equation 10, including the ensemble calibration.

Single-member factorization. For member $j \in \{1, \ldots, J\}$, let $\hat{x}_{j,v}^{\text{state}} \in \mathbb{R}^{d_{\text{int}}}$ and $\hat{x}_{i,j,w}^{\text{sel}} \in \mathbb{R}^{d_{\text{int}}}$ denote the projected views from Equation 8, and let $G_j \in \mathbb{R}^{10\times 10\times d_{\text{int}}}$ denote that member's interaction tensor. Its raw score, denoted by $\text{score}_j(s, m_i)$, follows Equation 9 with these member-specific quantities.

Concatenating the ten projected state views gives

$$q_j(s) = \text{concat}\left(\hat{x}_{j,1}^{\text{state}}, \ldots, \hat{x}_{j,10}^{\text{state}}\right) \in \mathbb{R}^{10d_{\text{int}}}. \tag{21}$$

On the transition side, collect the terms multiplying each state view v into a vector

$$\widetilde{x}_{i,j,v}^{\text{sel}} = \frac{1}{\sqrt{10 \cdot 10 \cdot d_{\text{int}}}} \sum_{w=1}^{10} [G_j]_{v,w,:} \odot \hat{x}_{i,j,w}^{\text{sel}} \in \mathbb{R}^{d_{\text{int}}}, \tag{22}$$

where $[G_j]_{v,w,:}$ is the interaction-weight vector for view pair (v, w) in member j. Although indexed by the state-view position v, $\widetilde{x}_{i,j,v}^{\text{sel}}$ uses no current-state features: it depends only on the stored transition and the member's parameters. Concatenating these ten vectors yields

$$k_{i,j} = \text{concat}\left(\widetilde{x}_{i,j,1}^{\text{sel}}, \ldots, \widetilde{x}_{i,j,10}^{\text{sel}}\right) \in \mathbb{R}^{10d_{\text{int}}}. \tag{23}$$

The raw score can therefore be rearranged as

$$\begin{aligned} \text{score}_j(s, m_i) &= \sum_{v=1}^{10} \left(\hat{x}_{j,v}^{\text{state}}\right)^\top \widetilde{x}_{i,j,v}^{\text{sel}} \\ &= q_j(s)^\top k_{i,j}. \end{aligned} \tag{24}$$

This factorization introduces no additional learned projection: it only regroups the terms in Equation 9. With $d_{\text{int}} = 32$, each member contributes 320 query and key coordinates.

Ensemble calibration. Let μ_j and $\sigma_j > 0$ be the mean and standard deviation of member j's raw scores computed from training data. These statistics are fixed before Writer–Reader optimization. The calibrated ensemble averages the standardized member scores:

$$\text{score}(s, m_i) = \frac{1}{J} \sum_{j=1}^{J} \frac{\text{score}_j(s, m_i) - \mu_j}{\sigma_j}. \tag{25}$$

To express this ensemble as one inner product, concatenate the member queries directly and incorporate the averaging and calibration scales into the member keys:

$$q(s) = \mathrm{concat}\left(q_1(s), \ldots, q_J(s)\right),$$
$$k_i = \mathrm{concat}\left(\frac{k_{i,1}}{J\sigma_1}, \ldots, \frac{k_{i,J}}{J\sigma_J}\right). \tag{26}$$

Both vectors have dimension $d_k = 10Jd_{\mathrm{int}} = 960$ in the reported implementation. Substituting Equation 24 into Equation 25 gives

$$\begin{aligned}\mathrm{score}(s, m_i) &= \frac{1}{J}\sum_{j=1}^{J}\frac{q_j(s)^\top k_{i,j} - \mu_j}{\sigma_j} \\ &= q(s)^\top k_i + b, \qquad b = -\frac{1}{J}\sum_{j=1}^{J}\frac{\mu_j}{\sigma_j}.\end{aligned} \tag{27}$$

As defined in Equation 11, the compiled field uses $\alpha(s, m_i) = q(s)^\top k_i$ rather than the full calibrated score including b. The frozen transition key can therefore be computed before deployment, while the current-state query is computed online.

A.4 Compiled-Field Algebra and Normalization

Section 3.5 defines the compiled field $A_{\mathcal{M}}$ and its state-conditioned read. Let $\kappa \in \{1, \ldots, d_k\}$ index the key dimension, $c \in \{1, \ldots, 8\}$ the payload slot, and $\nu \in \{1, \ldots, d_p\}$ the payload coordinate. For memory m_i, define its field contribution as $\Delta A_i = \rho_i k_i \otimes C_i$. Element-wise,

$$[\Delta A_i]_{\kappa c\nu} = \rho_i [k_i]_\kappa [C_i]_{c\nu}, \tag{28}$$

$$[A_{\mathcal{M}}]_{\kappa c\nu} = \sum_{i=1}^{N} \rho_i [k_i]_\kappa [C_i]_{c\nu}. \tag{29}$$

Contracting the key dimension with the current-state query gives

$$\left[q(s)^\top A_{\mathcal{M}}\right]_{c\nu} = \sum_{\kappa=1}^{d_k} [q(s)]_\kappa [A_{\mathcal{M}}]_{\kappa c\nu} \tag{30}$$

$$= \sum_{i=1}^{N} \rho_i \left(\sum_{\kappa=1}^{d_k} [q(s)]_\kappa [k_i]_\kappa\right) [C_i]_{c\nu} \tag{31}$$

$$= \sum_{i=1}^{N} \rho_i \left(q(s)^\top k_i\right) [C_i]_{c\nu}. \tag{32}$$

This is the explicit read in Equation 12. The equality is exact in real arithmetic; compiled and explicit floating-point evaluations can differ slightly because they accumulate terms in different orders.

Normalization is applied separately to each payload slot. For $Y \in \mathbb{R}^{8 \times d_p}$,

$$[\mathrm{RMSNorm}(Y)]_{c\nu} = \frac{Y_{c\nu}}{\sqrt{\frac{1}{d_p}\sum_{\nu'=1}^{d_p} Y_{c\nu'}^2 + \varepsilon}}, \qquad \varepsilon = 10^{-6}. \tag{33}$$

Applying this operation to $Y = q(s)^\top A_{\mathcal{M}}$ gives Equation 14. The normalization has no learned scale, and an empty field returns eight zero slots directly.

In the reported implementation, $d_k = 960$ and $d_p = 256$, so $A_{\mathcal{M}}$ has shape $960 \times 8 \times 256$ regardless of the number of stored memories N. The field implementation also supports an additional query-independent accumulator, but its per-record coefficient is zero in all reported runs; the accumulator is therefore identically zero and is omitted from the Method equations.

A.5 Cross-Attention Reader Architecture

Section 3.6 defines the Reader projections and residual update in Equation 15. The reported implementation uses $d_{\text{model}} = 4096$, $d_p = 256$, and $d_{\text{attn}} = 512$ with eight attention heads, giving head dimension $d_h = 64$. The projection matrices are

$$W_Q^{(\ell)} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{attn}}}, \qquad W_K^{(\ell)}, W_V^{(\ell)} \in \mathbb{R}^{d_p \times d_{\text{attn}}}, \qquad W_O^{(\ell)} \in \mathbb{R}^{d_{\text{attn}} \times d_{\text{model}}}, \tag{34}$$

and all are bias-free. The hidden states and memory slots use separate LayerNorms; the same memory-side normalization $\text{LN}_M^{(\ell)}$ is shared by the key and value projections.

For head $h \in \{1, \ldots, 8\}$, let $Q_h^{(\ell)} \in \mathbb{R}^{T_{\text{ctx}} \times d_h}$ and $K_h^{(\ell)}, V_h^{(\ell)} \in \mathbb{R}^{8 \times d_h}$ denote the corresponding projected slices. The attention weights $\Pi_h^{(\ell)}$ and head output $Z_h^{(\ell)}$ are

$$\Pi_h^{(\ell)} = \text{softmax}\left(\frac{Q_h^{(\ell)}\left(K_h^{(\ell)}\right)^\top}{\sqrt{d_h}}\right) \in \mathbb{R}^{T_{\text{ctx}} \times 8}, \qquad Z_h^{(\ell)} = \Pi_h^{(\ell)} V_h^{(\ell)} \in \mathbb{R}^{T_{\text{ctx}} \times d_h}. \tag{35}$$

Softmax is applied over the eight memory slots. The eight $Z_h^{(\ell)}$ are concatenated to form the $T_{\text{ctx}} \times d_{\text{attn}}$ output denoted by MHA in Equation 15, before multiplication by $W_O^{(\ell)}$. Attention scores and softmax are computed in float32, after which the attention weights are cast to the value-tensor dtype.

Separate Readers are attached after zero-indexed Qwen3-8B layers 7, 14, 21, and 28, with independent normalization and projection parameters at each insertion. Every $W_O^{(\ell)}$ is initialized to zero. The same state-conditioned slot tensor $U_{\mathcal{M}}(s)$ is supplied throughout generation of an action. During prompt prefill, T_{ctx} counts the positions processed together; under single-token cached decoding, $T_{\text{ctx}} = 1$ even though earlier tokens remain in the self-attention cache.

If $U_{\mathcal{M}}(s)$ is exactly zero, the deployment hook bypasses the Reader update and returns $H^{(\ell)}$ unchanged. This runtime bypass is distinct from zero initialization of $W_O^{(\ell)}$: it preserves the zero-field behavior after Reader parameters have been trained. The Reader keys and values are temporary cross-attention tensors and are never added to the language model's self-attention KV cache.

A.6 Editing, Provenance, and Leakage Controls

Section 3.7 defines record editing through ΔA_i and task-level exclusion through $A_{\text{legal}}(s)$. The authoritative memory ledger and compiled serving representation remain separate. The ledger retains each structured transition m_i and its provenance metadata ξ_i, while the derived field record stores the identifiers and fixed quantities needed to reproduce its contribution, including k_i, C_i, and ρ_i. The tensor ΔA_i need not be stored separately because it is reconstructed as $\rho_i k_i \otimes C_i$ when the record is applied.

Once the Writer is fixed, insertion and removal directly add or subtract ΔA_i, and replacement removes the original contribution before adding the replacement. The weight ρ_i remains fixed, so editing one record does not renormalize unrelated records. Reversibility refers to the identified contribution to the compiled field: removal and restoration are exact algebraically, subject to floating-point roundoff. It does not imply reconstruction of the original transition text or removal of any influence the record may have had during earlier Writer–Reader optimization.

During Writer–Reader optimization, the payloads C_i change with the Writer parameters, so $A_{\text{legal}}(s)$ cannot be treated as a fixed stored tensor. For each training state, A_{all} is constructed differentiably from the current payloads, and $A_{\tau(s)}$ is constructed from the same payload tensor using records from the current task. Equation 17 is then applied, allowing gradients to propagate through both field construction and task subtraction. After optimization, the Writer is fixed and record contributions can be compiled once for deployment.

The provenance metadata ξ_i is not semantic input to the Transition Writer or State-Transition Selector. It is used for record identity, normalization grouping, and eligibility checks before field construction. The available provenance identifiers depend on the benchmark ledger and can include task, episode, replay, and lineage information. These checks operate on the archival records; the subsequent state-conditioned read uses only the compiled field.

B Additional Experimental Details

B.1 Benchmark Details

Section 4.1 introduces the three evaluation benchmarks. Table 5 summarizes the benchmark-side data partitions used for training-side experience and formal evaluation. Evaluation populations are frozen before final method evaluation and are not used for memory construction, checkpoint selection, prompt selection, or other method configuration.

Table 5: Benchmark data used for training-side experience and formal evaluation.

Benchmark	Training / Validation Source	Formal Evaluation
AppWorld	Train (90 tasks)	Test_normal (168)
ALFWorld	Train (3,553 games)	Valid_unseen (134)
WebShop	Train $[1500, 12000)$; validation $[500, 1500)$	Standard $[0, 200)$

AppWorld. A transition contains the task goal, complete pre-action state, executed code action, and resulting environment observation. We use the full 90-task training split as the source of training-side experience, retaining successful replay-validated trajectories for memory construction and method training. Formal evaluation uses all 168 tasks in the AppWorld 0.1.0 `test_normal` split under the frozen prompt. No `test_normal` outcome is used for memory construction, training, or configuration.

ALFWorld. Transition boundaries follow individual textual environment actions and their returned observations. Training-side experience is constructed only from successful replay-validated training trajectories. Formal evaluation uses all 134 official `valid_unseen` games in a frozen order. The benchmark adapter preserves the upstream task identities and evaluator semantics, including the fixed method-neutral action-boundary translation required by the pinned ALFWorld runtime. No `valid_unseen` outcome is used for training or configuration.

WebShop. We use the maintained AgentBench-FC (Liu et al., 2024) WebShop runtime rather than the original WebShop execution harness. The frozen task manifest assigns standard evaluation to indices $[0, 200)$, validation to $[500, 1500)$, and training to $[1500, 12000)$. RCMF construction uses only training-side agent trajectories, with validation restricted to a fixed training-independent subset and the standard 200 tasks remaining inaccessible until the method is frozen. A transition consists of a complete search or click action paired with its pre-action state and resulting page observation. Mean raw reward is the primary evaluation metric; exact reward 1.0 is reported as a secondary full-success metric.

Across all benchmarks, the authoritative human-readable transition ledger is kept separate from derived memory tensors. Evaluation task identities, trajectories, rewards, and ground truth are excluded from memory construction. Complete prompt assets, environment versions, task manifests, transition renderers, maximum-step settings, and leakage checks are included in the released experiment manifests.

B.2 Baseline Adaptation

Section 4.1 compares RCMF with the ReAct no-cross-task-memory control and three external memory systems. Table 6 summarizes their memory interfaces. Adaptation preserves each method's defining mechanism rather than imposing the RCMF transition representation on every baseline.

Table 6: External memory baselines and their retained memory interfaces.

Method	Family	Memory Interface
ReMe	Explicit / structured	Procedural text
MemGen	Generative latent	Generated latent memory tokens
δ-mem	Fast-weight / test-time	Associative fast-weight state

ReMe. We preserve its procedural-memory representation and retrieval/refinement lifecycle. The adapter converts benchmark trajectories and decision states into the inputs required by the memory procedure and connects its output to the agent runtime.

MemGen. We preserve the trigger–weaver organization and generation of state-conditioned latent memory tokens. Adaptation supplies trajectory conversion, environment interaction, and the training and inference interfaces needed to use this mechanism with Qwen3-8B.

δ-mem. We preserve the online delta-rule update and memory-conditioned correction mechanism. The adapter maps the environment's interaction stream into the required update and read interfaces without substituting textual retrieval or another memory type.

Within each benchmark, methods share the evaluation tasks, environment, evaluator, and non-memory prompt scaffold. Memory-specific inputs and operations follow the corresponding method. Where benchmark conversion or runtime support is missing, we implement it from the paper and released code. Adaptation choices are fixed before the final comparison runs. All reported baseline scores come from executions in our benchmark environments, not from published results under different backbones or runtimes.

B.3 Implementation, Training, and Evaluation Details

The shared modules and tensor dimensions are specified in Section 3 and Appendix A. Here we detail the AppWorld training instance: Selector pretraining followed by the joint Writer–Reader optimization introduced in Section 3.8.

AppWorld Selector pretraining. Training pairs carry procedural-compatibility labels, and transitions sharing a canonical procedure are grouped into classes. A class score is the mean of its member-transition scores. The objective combines listwise supervision over classes, pairwise ranking, and hard-negative ranking between classes with the same coarse action type. Auxiliary binary cross-entropy terms supervise exact API-sequence matching and compatibility with the current task stage. The respective weights are 1.0, 0.75, 0.75, 0.1, and 0.1. The listwise target distribution is obtained by applying softmax to the compatibility labels with temperature 0.75.

Training-only grouped cross-validation selects the configuration before the final three members are fitted. Each member is trained for 120 epochs with learning rate 3×10^{-4}, weight decay 10^{-4}, and batches of eight states. The member seeds are 25071, 25072, and 25073. Ensemble calibration follows Appendix A.3. The Selector parameters and calibration statistics are then frozen throughout Writer–Reader optimization and checkpoint selection.

AppWorld Writer–Reader data. This stage uses 366 supervised decision states and a memory pool of 401 transitions from 29 training-split tasks. Eight other tasks from the same training split provide 98 held-out decision states and 98 transitions. The held-out states are used for checkpoint selection, whereas their transitions are excluded from both the optimization and checkpoint-selection fields. For a correct-field unit at state s, $A_{\mathrm{legal}}(s)$ is constructed from the 401-record pool using the task exclusion in Equation 17. Parent normalization follows the ledger identities, with record weights ρ_i held fixed during exclusion and editing as described in Appendix A.6.

Behavioral targets. For each supervised state s, the frozen Selector identifies one eligible transition m_s^{sel} before the behavioral audit. This discrete selection is used only to construct offline supervision, not for the student's field read. A paired one-step audit compares two reference policies obtained from the same frozen language model π_θ: $\pi_{\mathrm{bare}}(\cdot \mid s)$ receives no memory text, whereas $\pi_{\mathrm{raw}}(\cdot \mid s, m_s^{\mathrm{sel}})$ additionally receives the selected transition as raw text. Their executed outcomes define

$$y_s^{\mathrm{pair}} := y(s, m_s^{\mathrm{sel}}) \in \{\mathrm{beneficial}, \mathrm{neutral}, \mathrm{harmful}\}.$$

This is a label for the state–selected-transition pair, not an intrinsic property of either the state or the transition. The behavioral target for the correct-field unit is

$$\pi_{\mathrm{tgt}}(\cdot \mid s) = \begin{cases} \pi_{\mathrm{raw}}(\cdot \mid s, m_s^{\mathrm{sel}}), & y_s^{\mathrm{pair}} = \mathrm{beneficial}, \\ \pi_{\mathrm{bare}}(\cdot \mid s), & y_s^{\mathrm{pair}} \in \{\mathrm{neutral}, \mathrm{harmful}\}. \end{cases} \tag{36}$$

The student receives the complete correct or controlled field specified by its training unit, never m_s^{sel} as prompt text.

Table 7: Loss weights for AppWorld Writer–Reader training. A beneficial pair satisfies $y_s^{\text{pair}} = \text{beneficial}$. Terms are zero outside their listed conditions. The unit-balancing factor multiplies the complete weighted sum.

Term	Applied to	Weight
Policy distillation	All units	1.0
Teacher-token cross-entropy	All units	0.25
Expert-action cross-entropy	Correct-field units	0.05
Zero-field hinge	Correct-field units	0.10
Key–payload-shuffle hinge	Correct-field units, beneficial pairs	0.10
Reader residual penalty	All units	0.001
Writer payload penalty	All units	0.0001

Training units and balancing. The 366 audited pairs comprise 105 beneficial, 235 neutral, and 26 harmful pairs. Each state contributes one correct-field unit targeting Equation 36. Each beneficial pair additionally contributes a key–payload-shuffled unit and a state-query-shuffled unit, both targeting $\pi_{\text{bare}}(\cdot \mid s)$. The former permutes payloads across fixed keys and record weights; the latter uses the query of a training state from a different task while retaining the original policy context. Both permutations are deterministic.

There are therefore $366 + 2 \times 105 = 576$ units per epoch: 105 raw-target correct-field units and 471 bare-target units. Their per-unit balancing factors are $576/(2 \times 105)$ and $576/(2 \times 471)$, respectively, giving the two groups equal summed unit weights. The balancing factor multiplies the entire weighted unit objective, not just its distillation term.

Writer–Reader objective. Each unit uses the cached response and token-level distributions of its assigned reference policy. Policy distillation minimizes teacher-to-student KL divergence over the teacher's 64 retained token IDs and one additional category containing the remaining vocabulary probability mass. Student probabilities are evaluated for the same retained IDs, with their remaining mass aggregated into the same additional category; the retained probabilities are not renormalized in isolation. A separate cross-entropy term supervises the actual teacher-response tokens. Correct-field units also receive cross-entropy supervision on the recorded expert action for the current state. Each prediction loss is averaged over the supervised tokens of its corresponding response under teacher forcing; prompt and padding tokens are not prediction targets.

Two one-sided comparison terms apply to correct-field units. Let $\text{NLL}^{\text{tgt}}_{\text{corr}}(s)$ and $\text{NLL}^{\text{tgt}}_{\text{zero}}(s)$ denote the mean negative log-likelihood of the same cached response from $\pi_{\text{tgt}}(\cdot \mid s)$ under the correct field and bare frozen model, respectively. For a beneficial pair, the earlier key–payload-shuffled unit additionally scores the raw-reference response and caches its mean negative log-likelihood as $\text{NLL}^{\text{raw}}_{\text{kp,cache}}(s)$; that unit's optimization target remains the bare reference policy. The comparison terms are

$$\begin{aligned} \mathcal{L}_{\text{zero}}(s) &= \left[\text{NLL}^{\text{tgt}}_{\text{corr}}(s) - \text{NLL}^{\text{tgt}}_{\text{zero}}(s)\right]_+, \\ \mathcal{L}_{\text{kp}}(s) &= \left[\text{NLL}^{\text{tgt}}_{\text{corr}}(s) - \text{NLL}^{\text{raw}}_{\text{kp,cache}}(s)\right]_+, \qquad y_s^{\text{pair}} = \text{beneficial}, \end{aligned} \tag{37}$$

where $[x]_+ = \max(x, 0)$. For beneficial pairs, Equation 36 selects the raw-reference response, so both sides of the second comparison score the same response. These are zero-margin hinges; no positive separation constant is imposed.

The zero-field reference is cached before optimization and remains fixed. The key–payload reference is detached from the earlier forward pass for the same state, evaluated before that unit's parameter update. It is not recomputed using the parameters of the later correct-field unit. Neither cached reference receives gradients in the correct-field update.

The Reader penalty is the mean squared entries of the residual $\widetilde{H}^{(\ell)} - H^{(\ell)}$ from Equation 15, first averaged within each insertion-layer tensor and then across insertion layers. Unlike the prediction losses, this penalty is not masked to response positions. The Writer penalty is the mean squared entries of the current training-pool payload tensor, before task subtraction. Table 7 gives the coefficients of the weighted sum.

Table 8: Optimization settings for AppWorld Writer–Reader training.

Setting	Value
Optimizer	AdamW
Writer learning rate	10^{-4}
Reader learning rate	10^{-4}
Weight decay	10^{-4}
Maximum joint gradient norm	1.0
Training epochs	2
Training units per epoch	576
Optimizer updates	1,152
Writer–Reader training seed	25101
Language-model forward autocast	bfloat16
Writer and Reader parameters	float32
Gradient checkpointing	Enabled
Recovery checkpoints	Every 25 units and at epoch boundaries

Optimization and checkpoint selection. Table 8 lists the optimization settings. Each unit produces one backward pass and one optimizer update, even when multiple response sequences are evaluated together. Correct-field units batch the assigned teacher response with the recorded expert action; key–payload-shuffled units batch the bare-reference response with the raw-reference response needed for the cached comparison. For each beneficial pair, units are ordered as key–payload shuffle, state-query shuffle, and correct field, with a separate update after each unit. Two epochs yield 1,152 optimizer updates. Only the Writer and Readers are optimized; Qwen and the Selector remain frozen. Recovery checkpoints are not additional candidates for model selection.

Both end-of-epoch checkpoints are evaluated on the 98 held-out states under zero-field, correct-field, key–payload-shuffled, and state-query-shuffled conditions. Selection uses executed one-step behavior: canonical action-signature agreement, semantic successor matching, and execution success. The preregistered behavioral categories and ranking prioritize correct-field improvements over zero and shuffled controls while accounting for execution regressions and task-level coverage. Within a behavioral category, checkpoints are ranked by the behavioral selection score; lower held-out correct-field policy KL breaks ties, followed by the earlier epoch. Teacher-forced loss alone does not determine selection.

This procedure selects epoch two. After selection, the 98 held-out-task transitions are feed-forward compiled and added without further optimization, giving the 499-record deployment field. The reported AppWorld checkpoint uses the original three-demonstration prompt during training and the original first complete demonstration during its associated deployment evaluation.

Evaluation. Training and evaluation seeds are recorded separately. Metrics and fixed task populations follow Section 4.2 and Appendix B.1. The average success in Table 1 is the unweighted mean of the three benchmark-specific task-success percentages, using WebShop's reward-1.0 success rate rather than its mean raw reward.

B.4 Efficiency and Scaling Details

Section 4.3 reports trajectory-dependent execution cost; here we profile RCMF ingestion, field reads, storage, and language-model-side time to first token (TTFT) on a single GPU equipped with 80GB of VRAM. Timed GPU operations use synchronized CUDA events. The three profiling prompts are the shortest, median-length, and longest prompts in the frozen held-out training records, selected without using evaluation outcomes.

Memory ingestion. Table 9 separates frozen-Qwen encoding of each complete transition, computation of k_i and C_i, and construction and addition of ΔA_i. Over all 499 records, the median per-record costs are 452.23, 3.25, and 0.063 ms, respectively. Text rendering, tokenization, and field allocation are outside these timings. No transition is truncated. Profiling uses separately re-encoded records, not the immutable field used for task evaluation.

Memory-bank scaling. We use deterministic prefixes $N \in \{1, 8, 32, 128, 256, 499\}$ of the frozen memory order, retaining the original record weights ρ_i. For each precomputed query $q(s)$, we compare the compiled read in Equation 14 with a sequential per-record implementation of Equation 12, including the same slot-wise normalization. Read times are medians of 1,000 repetitions after warm-up; query construction and Reader execution are excluded.

Table 9: RCMF scaling on AppWorld (milliseconds). Ingestion columns are per-record medians over each prefix; read columns are medians of 1,000 repetitions with a precomputed short-prompt query. Explicit read uses the sequential reference implementation.

N	Raw Encode	Writer/Key Compile	Field Update	Compiled Read	Explicit Read
1	420.97	3.161	0.073	0.111	0.130
8	423.48	3.242	0.068	0.116	0.571
32	432.04	3.272	0.065	0.122	2.028
128	439.12	3.260	0.063	0.131	7.823
256	440.27	3.250	0.063	0.141	15.598
499	452.23	3.252	0.063	0.153	30.287

The explicit computation grows approximately linearly with the number of records, increasing by more than two orders of magnitude between $N = 1$ and $N = 499$. In contrast, the compiled read remains below 0.16 ms throughout the sweep. Repeating the read benchmark under medium and long prompts gives the same qualitative behavior: at $N = 499$, median compiled-read latency is 0.150 and 0.149 ms, respectively, while the explicit computation remains approximately 30.29 ms.

Serving state. The float32 tensor $A_{\mathcal{M}}$ occupies $960 \times 8 \times 256 \times 4 = 7{,}864{,}320$ bytes. The implementation also retains the identically zero 8×256 accumulator described in Appendix A.4, adding 8,192 bytes. Total compiled-bank tensor storage is therefore 7,872,512 bytes (7.87 MB, with $1\,\mathrm{MB} = 10^6$ bytes) for any N. This excludes model weights, activations, the language-model KV cache, and archival records.

Fixed Reader and Writer parameter tensors occupy 71.44 and 35.80 MB, respectively. Selector checkpoint files occupy 189.24 MB, a file-size measurement rather than resident parameter memory. Estimated archival storage for ledger text, per-record deletion state, and proportionally allocated provenance grows from 0.06 MB at $N = 1$ to 31.09 MB at $N = 499$, excluding the separately archived representation cache. The constant-size bank-storage claim does not extend to total archival or runtime memory.

Time to first token. Table 10 measures GPU prefill through first-token selection for bare Qwen3-8B and RCMF at $N = 499$. Tokenization, state encoding, Selector query construction, and field reads occur before timing; RCMF slots are precomputed. This measures language-model-side latency with the Reader, not complete decision-step TTFT. Each condition uses 10 warm-up runs and 30 measured repetitions, validated against its own reference generated-token sequence rather than against another condition's outputs.

Table 10: Language-model-side TTFT (milliseconds) with precomputed RCMF slots ($N = 499$). Added latency is the difference of unrounded medians.

Prompt	Bare Qwen	RCMF	Added
Short	294.99	299.26	4.28
Medium	421.14	425.49	4.36
Long	809.21	815.94	6.73

With precomputed slots, RCMF adds little language-model-side first-token latency across the tested prompt lengths. The Reader attends to a fixed set of eight slots at each token position, so longer prompts can incur greater prefill overhead without requiring access to individual memory records.

The external memory baselines are not assigned an artificial common memory-count sweep because their stored units and update semantics differ: ReMe maintains procedural textual memories, MemGen produces latent memory states, and δ-mem maintains a fast-weight state. Their complete

Algorithm 1 Reversible Compiled Memory Field

Require: Transition memory bank $\mathcal{M} = \{m_i\}_{i=1}^{N}$, current state s, frozen language model, Transition Writer W_ϕ, frozen State-Transition Selector, Cross-Attention Readers

$A_{\mathcal{M}} \leftarrow 0$
for $m_i \in \mathcal{M}$ **do** ▷ offline compilation
  $X_i^{\text{write}}, X_i^{\text{sel}} \leftarrow \textsc{RepresentTransition}(m_i)$
  $C_i \leftarrow W_\phi(X_i^{\text{write}})$
  $k_i \leftarrow \textsc{SelectorKey}(X_i^{\text{sel}})$
  $A_{\mathcal{M}} \leftarrow A_{\mathcal{M}} + \rho_i k_i \otimes C_i$
end for
$X^{\text{state}}(s) \leftarrow \textsc{RepresentState}(s)$ ▷ online, once per decision step
$q(s) \leftarrow \textsc{SelectorQuery}(X^{\text{state}}(s))$
$U_{\mathcal{M}}(s) \leftarrow \mathrm{RMSNorm}\left(q(s)^\top A_{\mathcal{M}}\right)$
for selected language-model layers ℓ **do**
  $H^{(\ell)} \leftarrow H^{(\ell)} + \textsc{CrossAttentionReader}(H^{(\ell)}, U_{\mathcal{M}}(s))$
end for
return next action from the frozen language model

Edit: $A_{\mathcal{M}} \leftarrow A_{\mathcal{M}} \pm \rho_i k_i \otimes C_i$ ▷ insert / remove one memory

formal AppWorld wall-clock measurements are therefore reported in Table 2, while the controlled N-scaling experiment is restricted to the RCMF representation whose memory count is explicitly defined at the transition-record level.

C RCMF Core Procedure

Algorithm 1 summarizes the benchmark-independent RCMF computation. The benchmark supplies the transition and current-state representations and the per-record normalization weights ρ_i; the algorithm below covers the shared Writer, frozen Selector decomposition, field compilation and editing, state-conditioned read, and Cross-Attention Reader.

D Core Implementation

The following condensed implementation excerpt contains the benchmark-independent RCMF core shared by our AppWorld and WebShop implementations. We omit validation, bookkeeping, provenance handling, and the optional query-independent field accumulator, which is identically zero in all reported runs. The batched contraction in `read_compiled_field` is the vectorized implementation of $q(s)^\top A_{\mathcal{M}}$ in Equation 14.

Listing 1: Benchmark-independent core implementation of RCMF.

```
import math
from collections.abc import Sequence
import torch
from torch import Tensor, nn
import torch.nn.functional as F


def rms_norm(value: Tensor, *, eps: float = 1.0e-6) -> Tensor:
   work = value.to(torch.float32)
   normalized = work * torch.rsqrt(
      work.square().mean(dim=-1, keepdim=True) + eps
   )
   return normalized.to(value.dtype)


class SectionWriter(nn.Module):
   def __init__(
      self, *, input_dim=4096, hidden_dim=512, payload_dim=256
   ):
      super().__init__()
      self.norm = nn.LayerNorm(input_dim)
```

```
        self.input = nn.Linear(input_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, payload_dim)

    def forward(
        self, values: Tensor, pooling_embedding: Tensor
    ) -> Tensor:
        hidden = self.input(self.norm(values))
        hidden = hidden + pooling_embedding.unsqueeze(0).to(hidden.dtype)
        return self.output(F.gelu(hidden))


class AlignedTransitionWriter(nn.Module):
    section_names = ("goal", "pre_state", "action", "observation")

    def __init__(
        self, *, input_dim=4096, hidden_dim=512, payload_dim=256
    ):
        super().__init__()
        self.pooling_type_embedding = nn.Parameter(
            torch.empty(2, hidden_dim)
        )
        nn.init.normal_(
            self.pooling_type_embedding, mean=0.0, std=0.02
        )
        self.writers = nn.ModuleDict({
            name: SectionWriter(
                input_dim=input_dim,
                hidden_dim=hidden_dim,
                payload_dim=payload_dim,
            )
            for name in self.section_names
        })

    def forward(self, views: Tensor) -> Tensor:
        sections = []
        for index, name in enumerate(self.section_names):
            sections.append(self.writers[name](
                views[:, 2 * index : 2 * index + 2],
                self.pooling_type_embedding,
            ))
        return torch.cat(sections, dim=1)


class SignatureBalancedFieldSelector(nn.Module):
    def __init__(
        self, *, state_views, transition_views,
        input_dim, projection_dim, interaction_rank
    ):
        super().__init__()
        self.state_views = int(state_views)
        self.transition_views = int(transition_views)
        self.interaction_rank = int(interaction_rank)

        self.state_projection = nn.ModuleList(
            nn.Linear(input_dim, projection_dim, bias=False)
            for _ in range(state_views)
        )
        self.transition_projection = nn.ModuleList(
            nn.Linear(input_dim, projection_dim, bias=False)
            for _ in range(transition_views)
        )
        self.state_rank = nn.Linear(
            projection_dim, interaction_rank, bias=False
        )
        self.transition_rank = nn.Linear(
            projection_dim, interaction_rank, bias=False
        )
        self.tensor_core = nn.Parameter(torch.empty(
            state_views, transition_views, interaction_rank
        ))
        nn.init.normal_(
            self.tensor_core,
            std=1.0 / math.sqrt(interaction_rank),
        )

    @staticmethod
    def _project(
        values: Tensor, projections: nn.ModuleList
    ) -> Tensor:
        return torch.stack([
            layer(values[:, index])
```

```
            for index, layer in enumerate(projections)
        ], dim=1)

    def state_factors(self, state: Tensor) -> Tensor:
        return self.state_rank(
            self._project(state, self.state_projection)
        )

    def transition_factors(self, transition: Tensor) -> Tensor:
        return self.transition_rank(
            self._project(transition, self.transition_projection)
        )


class FrozenSelectorDecomposition(nn.Module):
    def __init__(
        self,
        *,
        models: Sequence[SignatureBalancedFieldSelector],
        train_means: Sequence[float],
        train_stds: Sequence[float],
    ):
        super().__init__()
        self.models = nn.ModuleList(models)
        self.train_means = tuple(
            float(value) for value in train_means
        )
        self.train_stds = tuple(
            float(value) for value in train_stds
        )
        for model in self.models:
            model.eval().requires_grad_(False)

    @property
    def intercept(self) -> float:
        return -sum(
            mean / std
            for mean, std in zip(
                self.train_means,
                self.train_stds,
                strict=True,
            )
        ) / len(self.models)

    @torch.no_grad()
    def query(self, state_views: Tensor) -> Tensor:
        return torch.cat([
            model.state_factors(state_views).reshape(
                state_views.shape[0], -1
            )
            for model in self.models
        ], dim=-1)

    @torch.no_grad()
    def key(self, transition_views: Tensor) -> Tensor:
        values = []
        ensemble_size = len(self.models)
        for model, std in zip(
            self.models, self.train_stds, strict=True
        ):
            factors = model.transition_factors(transition_views)
            scale = math.sqrt(
                model.state_views
                * model.transition_views
                * model.interaction_rank
            )
            effective = torch.einsum(
                "vwr,bwr->bvr",
                model.tensor_core,
                factors,
            ) / scale
            values.append(
                (effective / (ensemble_size * std))
                .flatten(start_dim=1)
            )
        return torch.cat(values, dim=-1)

    @torch.no_grad()
    def decomposed_scores(
        self,
        state_views: Tensor,
```

```
        transition_views: Tensor,
    ) -> Tensor:
        return (
            self.query(state_views)
            @ self.key(transition_views).T
            + self.intercept
        )


def compile_differentiable_field(
    keys: Tensor,
    payloads: Tensor,
    rho: Tensor,
) -> Tensor:
    weighted = payloads * rho[:, None, None].to(payloads.dtype)
    return torch.einsum(
        "nk,nsp->ksp",
        keys.to(payloads.dtype),
        weighted,
    )


def read_compiled_field(
    query: Tensor,
    A: Tensor,
) -> Tensor:
    raw = torch.einsum(
        "bk,ksp->bsp",
        query.to(A.dtype),
        A,
    )
    return rms_norm(raw)


def edit_field(
    A: Tensor,
    key: Tensor,
    payload: Tensor,
    rho: float,
    sign: float,
) -> None:
    A.add_(
        torch.einsum("k,sp->ksp", key, payload),
        alpha=float(sign) * float(rho),
    )


class LayerFieldCrossAttention(nn.Module):
    def __init__(
        self,
        *,
        model_dim=4096,
        payload_dim=256,
        attention_dim=512,
        heads=8,
    ):
        super().__init__()
        self.attention_dim = int(attention_dim)
        self.heads = int(heads)
        self.head_dim = self.attention_dim // self.heads

        self.query_norm = nn.LayerNorm(model_dim)
        self.memory_norm = nn.LayerNorm(payload_dim)
        self.query = nn.Linear(
            model_dim, attention_dim, bias=False
        )
        self.key = nn.Linear(
            payload_dim, attention_dim, bias=False
        )
        self.value = nn.Linear(
            payload_dim, attention_dim, bias=False
        )
        self.output = nn.Linear(
            attention_dim, model_dim, bias=False
        )
        nn.init.zeros_(self.output.weight)

    def forward(self, hidden: Tensor, slots: Tensor):
        q = self.query(self.query_norm(hidden))
        memory = self.memory_norm(
            slots.to(hidden.device, hidden.dtype)
```

```
        )
        k = self.key(memory)
        v = self.value(memory)

        batch, query_tokens = q.shape[:2]
        q = q.view(
           batch, query_tokens, self.heads, self.head_dim
        ).transpose(1, 2)
        k = k.view(
           batch, 8, self.heads, self.head_dim
        ).transpose(1, 2)
        v = v.view(
           batch, 8, self.heads, self.head_dim
        ).transpose(1, 2)

        scores = torch.matmul(
           q.to(torch.float32),
           k.to(torch.float32).transpose(-2, -1),
        ) / math.sqrt(self.head_dim)

        probabilities = F.softmax(
           scores, dim=-1, dtype=torch.float32
        ).to(v.dtype)
        context = torch.matmul(probabilities, v)

        context = (
           context.transpose(1, 2)
           .contiguous()
           .view(batch, query_tokens, self.attention_dim)
        )
        delta = self.output(
           context.to(self.output.weight.dtype)
        ).to(hidden.dtype)

        return hidden + delta, probabilities, delta
```