PROGRAM-CENTRIC MEMORY: A REPRESENTATION-FOR-FUNCTION APPROACH TO EXPERIENCE REUSE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents can construct memory from past interaction trajectories and reuse it to support future action selection and outcome prediction. However, when these memory functions are carried primarily by language, predictive information may remain implicit, while reusable action patterns require repeated LLM reasoning at runtime. This motivates a memory-form perspective that we call **Representation for Function**: memory representations should align with the functions they serve at reuse time. By aligning memory form with function, we propose **Program-Centric Memory (PCM)**, which represents explicit probabilistic action–outcome relationships as **Predictive Beliefs** and reusable action patterns as **Executable Programs**. PCM achieves consistently strong performance across ALFWorld, WebShop, and TravelPlanner. Controlled experiments further show that, with the underlying information held fixed, using executable rather than textual representations of the same action patterns increases success from 79.9% to 97.8%, while reducing online tokens from 98.4k to 2.5k per task. These results establish memory form as a key dimension of experience reuse.

1 INTRODUCTION

Large language model (LLM) agents continuously accumulate trajectories as they interact with environments. A key function of memory is to transform past experience into reusable knowledge that can guide future actions. Among the information contained in trajectories, two types are particularly important for action: probabilistic relationships between actions and outcomes, and reusable patterns for carrying out actions. Effectively preserving and reusing these two forms of action-relevant information is therefore a central challenge for agent memory.

Many existing agent memory methods reuse past experience through linguistic context. Two common forms are **Retrieval-Oriented Memory**, which retrieves relevant trajectories, cases, or summaries to support current decisions (Chhikara et al., 2025; Xu et al., 2025), and **Procedure-Oriented Memory**, which stores linguistic guidance on how actions should be carried out (Wang et al., 2025; Forouzandeh et al., 2026; Cao et al., 2026). In many agent systems, such memories are exposed to the agent through language, which we broadly refer to as **Language-Centric Memory**. *While language is flexible in describing rich experience, linguistic memory may not be equally suited to all types of information.* Probabilistic relationships preserved in descriptive language may remain implicit or underspecified. Action patterns preserved in descriptive language require the LLM to reinterpret them for concrete cases, making reuse dependent on online reasoning. As a result, even when past experience contains useful predictive relationships and reusable execution patterns, the agent may still need to reconstruct how to use this information each time the memory is reused. Figure 1 illustrates this limitation with a simple example. Suppose an agent is searching for a target among several boxes. Memory may state that "Box A is likely to contain the target" or "Check Box A first." However, such descriptions do not directly specify how Box A should be compared with other boxes, or what action should follow if Box A is empty. The agent must still interpret the stored description, compare alternatives, and adapt it to the current state. This suggests a basic principle: both what information is preserved in memory and the form in which it is preserved matter.

Based on the above discussion, we ask a more fundamental question: What makes a good memory representation for future reuse? Our key idea is simple: ***Representation for Function***—*memory should be represented in a form aligned with its function at reuse time.* For probabilistic relation-

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

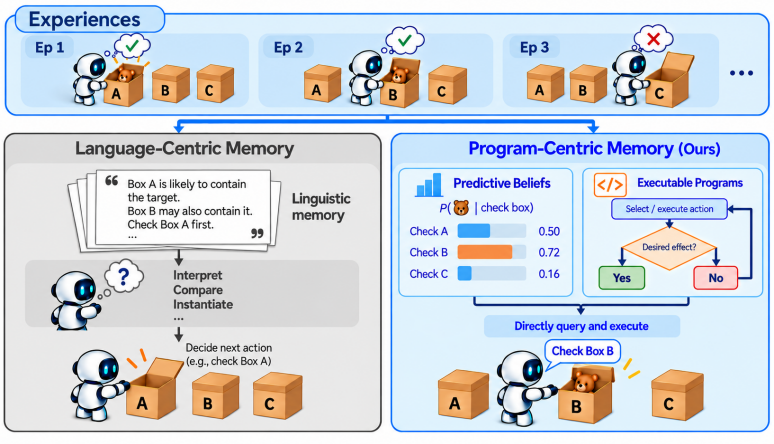


Figure 1: Conceptual comparison between Language-Centric Memory and Program-Centric Memory. Program-Centric Memory organizes experience as Predictive Beliefs and Executable Programs for action-outcome relationships and action patterns.

ships between actions and outcomes, trajectories provide repeated observations of what follows from an action under different states. By accumulating these observations, the agent can maintain explicit estimates of such relationships, which we call **Predictive Beliefs**. For reusable execution patterns, trajectories reveal recurring action structures that succeeded in similar situations. Instead of storing these patterns only as textual guidance, we represent them as **Executable Programs** that can be instantiated, executed, and refined. We argue that these two forms better align with their computational functions at reuse time: Predictive Beliefs support prediction through explicit, queryable estimates, while Executable Programs support execution through directly reusable control structures.

Based on this motivation, we propose **Program-Centric Memory (PCM)**, which compiles accumulated trajectories into two complementary forms: **Predictive Beliefs** and **Executable Programs**. Predictive Beliefs provide explicit, queryable estimates of action-outcome relationships, while Executable Programs encode reusable action patterns as stateful control structures. PCM is *program-centric* because Predictive Beliefs provide the information needed for decision-making, while Executable Programs determine how that information is used to produce actions. At runtime, PCM matches an applicable program, queries Predictive Beliefs when needed, and executes the resulting action; uncovered situations fall back to the LLM. In this way, past experience is represented in forms aligned to their computational functions at reuse time—Predictive Beliefs for prediction and Executable Programs for execution.

Our work makes three main contributions. (1) We introduce a memory-form perspective on experience reuse, which we summarize as *Representation for Function*: memory representations should align with their functions at reuse time, making memory form a key design dimension for agent memory. (2) We propose **Program-Centric Memory (PCM)**, which operationalizes this perspective through two complementary forms: Predictive Beliefs for explicit and queryable action–outcome estimation, and Executable Programs for directly reusable execution patterns. (3) Controlled experiments show that, with the underlying experience held fixed, using representations better aligned with their functions at reuse time substantially improves performance.

2 Related Work

Experience memory for language-model agents. Early language-model agents such as ReAct primarily rely on online reasoning over observations and interaction history. Memory-augmented agents extend this paradigm by retaining past interaction experience outside model parameters. Reflexion stores verbal feedback from previous trials, while ExPeL distills trajectories into reusable insights. Subsequent approaches further improve memory abstraction and organization through concise notes, structured representations, and graph-based memory, such as A-MEM and G-Memory (Yao et al., 2023; Shinn et al., 2023; Zhao et al., 2024; Xu et al., 2025; Zhang et al.,

2025). Together, these works substantially expand the design space of what experience is retained, how it is organized, and how relevant experience is retrieved for future decisions.

Procedural memory and runtime reuse. A complementary line of work represents reusable experience as workflows, skills, procedures, or executable code. AWM induces reusable task workflows, while Memp, MACLA, and Skill-Pro develop increasingly structured forms of procedural memory; code-based agents such as Voyager further demonstrate that reusable behavior can be retained in executable form (Wang et al., 2025; Fang et al., 2025; Forouzandeh et al., 2026; Mi et al., 2026; Wang et al., 2023). These approaches span different realizations of procedural knowledge, from language-mediated guidance to executable skills, motivating a broader question: in what form should action-relevant experience be represented to serve its computational function at reuse time? Unlike prior work that primarily develops particular forms of procedural memory, PCM takes a memory-form perspective, aligning different types of action-relevant information with their distinct computational functions at reuse time. It represents action–outcome relationships as explicit, queryable Predictive Beliefs, and reusable action patterns as Executable Programs that encode stateful control structures. This **Representation for Function** perspective aligns memory form with prediction and execution, establishing memory form as an important design dimension of experience reuse.

3 Program-Centric Memory

3.1 Problem Formulation

Consider an LLM agent interacting with an environment over a sequence of tasks. A trajectory is $\tau_i = (S_0, A_0, S_1, \ldots, A_{T_i-1}, S_{T_i})$, and the accumulated past experience is $\mathcal{D} = \{\tau_i\}_{i=1}^{N}$. The agent constructs an external memory $\mathcal{M} = \mathcal{F}(\mathcal{D})$, which persists across tasks and is reused to guide future actions. The goal of memory is not merely to retain past trajectories, but to preserve action-relevant information in forms suited to its functions in future actions.

We focus on two recurring kinds of action-relevant information in past interactions, which serve different computational functions at reuse time. *Action–outcome relationships* capture what observable outcomes tend to follow actions under different conditions, supporting prediction and comparison among alternatives. *Reusable execution patterns* capture how behavior should proceed across states through conditions, entity bindings, branching, and termination, supporting control and execution.

In language-mediated memory, stored experience is reused as textual context and interpreted by the LLM at reuse time. The computational function of the stored information is not explicit in the representation itself: action–outcome relationships must be reconstructed for prediction and comparison, while execution patterns must be interpreted and instantiated for the current state. This computation is repeated whenever the memory is reused, even when the relevant information has already been stored. We instead follow the principle of **Representation for Function**, encoding repeatedly reconstructed computation into reusable memory structures aligned with their computational functions.

$$\underbrace{\mathcal{M}_L \rightarrow \text{LLM} \rightarrow \{\text{prediction}, \text{execution}, ...\}}_{\text{Language-mediated reuse}} \quad \text{vs.} \quad \underbrace{\begin{matrix} B \rightarrow \text{prediction} \\ \Pi \rightarrow \text{execution} \end{matrix}}_{\text{Representation for Function}}.$$

To instantiate this principle, we propose **Program-Centric Memory** (PCM) as:

$$\mathcal{M}_{\text{PCM}} = (B, \Pi), \tag{1}$$

where B denotes **Predictive Beliefs** that represent action–outcome relationships as explicit and queryable estimates, and $\Pi = \{\pi_1, \ldots, \pi_K\}$ denotes a library of **Executable Programs** that represent reusable execution patterns as stateful control structures. Together, they represent predictive and procedural information in forms aligned with their computational functions at reuse time. Figure 2 summarizes the PCM lifecycle, from memory update to runtime reuse.

3.2 Predictive Beliefs

Predictive Beliefs represent recurring action–outcome relationships in a form aligned with their computational function of prediction and comparison. For each executed transition (S_t, A_t, S_{t+1}),

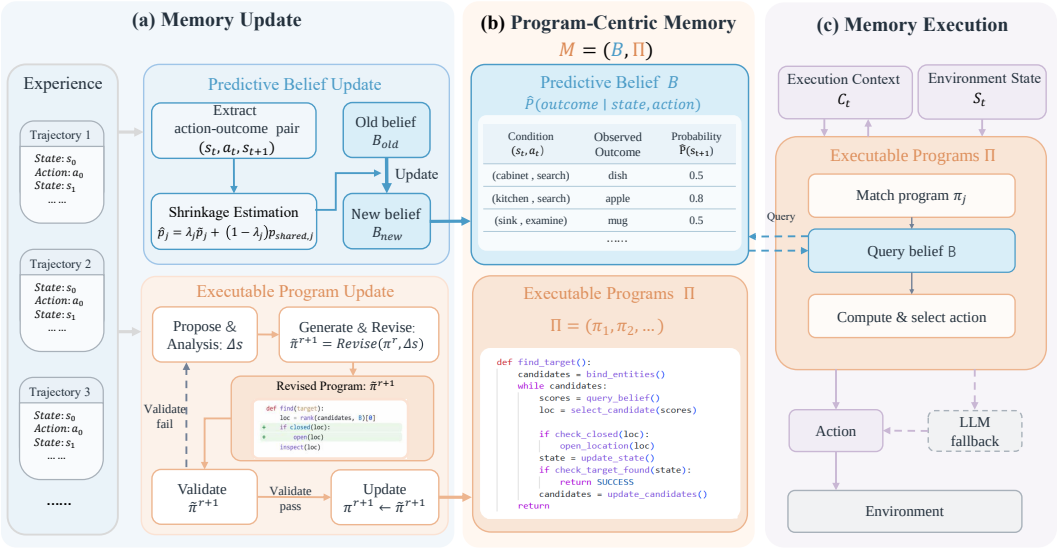


Figure 2: Overview of the Program-Centric Memory framework. **(a) Memory Update:** executed transitions provide action–outcome evidence for updating Predictive Beliefs, while trajectories and execution feedback support Executable Program construction and revision. **(b) Program-Centric Memory:** Predictive Beliefs B and Executable Programs Π. **(c) Memory Execution:** given current state S_t and execution context C_t, the agent matches an applicable program, queries B when needed, and computes the next action; states outside program coverage use LLM fallback.

we extract an observable context $X_t = \psi(S_t, A_t)$ and a local outcome $Y_{t+1} = \phi(S_t, A_t, S_{t+1})$, where both are computed from information available to the agent. For an outcome space $\mathcal{Y} = \{y_1, \ldots, y_J\}$, a Predictive Belief represents the conditional outcome distribution as

$$B(x) = \widehat{\mathbf{p}}(x), \qquad \widehat{\mathbf{p}}(x) = \big(\widehat{p}_1(x), \ldots, \widehat{p}_J(x)\big) \in \Delta^{J-1}, \tag{2}$$

where $\widehat{p}_r(x)$ denotes the estimated probability of outcome y_r under context x. Thus, instead of describing an outcome as merely "likely" or "unlikely", the Predictive Beliefs expose explicit estimates that can be used to compare candidate actions. They model observable local outcomes rather than final task success or a complete environment model, and do not imply causal intervention effects.

Predictive Belief Update. From past trajectories, we collect context–outcome pairs (X_t, Y_{t+1}) from executed transitions. For a context cell j, let $\mathbf{c}_j = (c_{j1}, \ldots, c_{jJ}), n_j = \sum_{r=1}^{J} c_{jr}$, where c_{jr} is the number of times outcome y_r is observed under context j. A purely local empirical estimate can be unreliable when n_j is small. We therefore shrink the local evidence toward a broader shared distribution. Specifically, a predefined grouping rule $g(\cdot)$ associates each context cell with a set of related contexts. Let $\mathbf{p}_{\text{shared},j}$ denote the empirical outcome distribution estimated from historical observations in this broader group. We adopt a Dirichlet–multinomial model with prior concentration $\kappa > 0$ centered at $\mathbf{p}_{\text{shared},j}$. Its posterior mean is $\widehat{\mathbf{p}}_j = \frac{\mathbf{c}_j + \kappa \mathbf{p}_{\text{shared},j}}{n_j + \kappa}$. Equivalently, when $n_j > 0$,

$$\widehat{\mathbf{p}}_j = \lambda_j \widetilde{\mathbf{p}}_j + (1 - \lambda_j)\mathbf{p}_{\text{shared},j}, \qquad \lambda_j = \frac{n_j}{n_j + \kappa}, \tag{3}$$

where $\widetilde{\mathbf{p}}_j = \mathbf{c}_j / n_j$ is the local empirical estimate. Consequently, sparse contexts rely more on shared experience, while increasing local evidence gradually shifts the belief toward the context-specific estimate. When $n_j = 0$, the belief reduces to $\mathbf{p}_{\text{shared},j}$. The derivation is provided in Appendix A.4.

3.3 Executable Programs

The Executable Program library $\Pi = \{\pi_1, \ldots, \pi_K\}$ represents reusable action patterns as stateful control structures aligned with their computational functions. Each program is defined as:

$\pi_j = (m_j, q_j, d_j, u_j)$, where m_j determines program applicability and binds entities, q_j specifies when and how to query Predictive Beliefs, d_j selects the next environment action, and u_j updates the execution context. Together, these components preserve reusable execution logic rather than replaying a fixed sequence of actions from a past trajectory.

Let S_t denote the current observable state and C_t the execution context, which maintains bindings and execution progress. Given (S_t, C_t), m_j produces the required bindings $z_t = m_j(S_t, C_t)$. When outcome comparison is needed, q_j constructs a belief query $Q_t = q_j(S_t, C_t, z_t)$, and the decision rule selects the next action as $A_t = d_j\left(S_t, C_t, z_t, Q_t(B)\right)$. After observing S_{t+1}, the execution context is updated as follows: $C_{t+1} = u_j(C_t, S_t, A_t, S_{t+1})$, allowing the program to maintain bindings and execution progress across multiple steps. In this way, Executable Programs realize reusable control and execution, while Predictive Beliefs B provide explicit outcome information when required by that logic.

Executable Program Update. Executable Programs are constructed from past trajectories and progressively refined using new trajectories and execution feedback. The LLM analyzes real trajectories generated through interactions between the agent and the environment. If an improved behavioral pattern is associated with an existing program π_j, the system produces a revision proposal Δs_j, specifying how the existing program should be modified. Likewise, if a reusable behavioral pattern is not covered by any existing program, the system produces a generation proposal. Based on these proposals, the LLM can either revise or generate a new candidate program $\tilde{\pi}_j$.

$$\tilde{\pi}_j^{(r+1)} = \mathrm{Revise}\left(\pi_j^{(r)}, \Delta s_j\right). \tag{4}$$

Program generation can be viewed as a special case of revision in which the current program $\pi_j^{(r)}$ is empty. Before a candidate program is committed to the program library, it is validated through interface and behavioral tests. If validation succeeds, the candidate replaces the current program. Otherwise, the LLM uses the failure diagnostics to further revise the candidate within a bounded update budget. If no candidate passes validation, the existing program is retained. A detailed case study and analysis are provided in Appendix B.3.

3.4 Memory Execution

At runtime, PCM coordinates Predictive Beliefs B and the Executable Program library Π to produce actions. Given the current observable state S_t, the execution context C_t, and the applicability conditions of all programs, denoted by $\mathrm{COND}_\Pi = (m_1, m_2, \ldots)$, the system first matches an applicable program: $(\pi_j, z_t) = \mathrm{Match}_\Pi\left(S_t, C_t, \mathrm{COND}_\Pi\right)$, where π_j denotes the selected program and z_t contains the current bindings.

If a program π_j is matched, its decision rule d_j produces the next action, using Predictive Beliefs when required; otherwise, action selection falls back to the LLM:

$$A_t = \begin{cases} d_j\left(S_t, C_t, z_t, Q_t(B)\right), & \text{covered}, \\ \pi_{\mathrm{LLM}}\left(S_t, C_t\right), & \text{otherwise}. \end{cases} \tag{5}$$

After observing S_{t+1}, the execution context is updated as $C_{t+1} = u_j\left(C_t, S_t, A_t, S_{t+1}\right)$, allowing the program to maintain task-specific bindings and execution progress across steps. During evaluation, B and Π remain fixed; only the transient execution context C_t is updated. Hence, compiled experience supports prediction and execution through the corresponding memory forms, while the LLM is invoked only when the current situation is not covered by existing programs.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate PCM on ALFWorld (Shridhar et al., 2021), WebShop (Yao et al., 2022), and TravelPlanner (Xie et al., 2024), covering embodied interaction, online shopping, and multi-constraint planning. ALFWorld contains 140 valid-seen and 134 valid-unseen tasks and serves as the primary benchmark for both overall evaluation and controlled studies of memory form, memory

Table 1: Comparison of methods on ALFWorld and WebShop. Success rates are percentages; Mean Reward is on its original scale. * denotes results reproduced by our re-implementation.

		ALFWorld ↑		WebShop ↑	
Method	Models	Valid_seen	Valid_unseen	Exact Success	Mean Reward
ReAct* (Yao et al., 2023)	DS-V4-flash	62.1%	65.7%	45.4%	0.656
ExPeL* (Zhao et al., 2024)	DS-V4-flash	65.0%	69.4%	44.6%	0.656
AWM* (Wang et al., 2025)	DS-V4-flash	70.0%	76.1%	45.4%	0.665
ReflAct* (Kim et al., 2025)	DS-V4-flash	79.3%	86.6%	43.3%	0.646
MACLA (Forouzandeh et al., 2026)	Llama-2-7B	87.2%	90.3%	–	0.702
PCM	DS-V4-flash	**97.9%**	**97.8%**	**47.1%**	**0.713**

Table 2: Interaction efficiency and LLM usage across evaluated environments. Values are means per task, rounded to three significant figures. * denotes results reproduced by our re-implementation.

		ALFWorld ↓			WebShop ↓		
Method	Models	Steps	LLM Calls	Tokens	Steps	LLM Calls	Tokens
ReAct* (Yao et al., 2023)	DS-V4-flash	27.1	28.6	18.7k	8.78	6.50	9.45k
ExPeL* (Zhao et al., 2024)	DS-V4-flash	25.8	27.0	19.2k	8.40	6.29	9.60k
AWM* (Wang et al., 2025)	DS-V4-flash	22.2	22.9	22.0k	8.38	**6.11**	10.1k
ReflAct* (Kim et al., 2025)	DS-V4-flash	23.4	23.4	38.7k	**8.34**	6.18	**8.31k**
PCM	DS-V4-flash	**10.9**	**3.94**	**2.49k**	8.49	6.42	9.22k

updates, and individual mechanisms. WebShop includes 240 evaluation tasks in a distinct interactive shopping environment, testing the generality of PCM across environments. TravelPlanner contains 180 validation queries in a structurally different planning setting, further evaluating the generality of PCM beyond step-by-step interactive tasks.

Baselines. We compare PCM with representative baselines including ReAct (Yao et al., 2023), ReflAct (Kim et al., 2025), ExPeL (Zhao et al., 2024), AWM (Wang et al., 2025), and MACLA (Forouzandeh et al., 2026), and report benchmark-specific comparisons below.

Evaluation Protocol and Implementation. We strictly separate memory construction from evaluation. Predictive Beliefs B and Executable Programs Π are fully constructed and frozen before evaluation; trajectories, action outcomes, and success signals generated during evaluation are not used to update persistent memory. During evaluation, only the transient execution context C_t is updated through interaction, while B and Π remain fixed. We evaluate PCM using DeepSeek V4 Flash (DS-V4-flash) with thinking disabled through its official API and fixed memory configurations. The ALFWorld memory-construction data and evaluation splits are detailed in Appendix A.1. TravelPlanner implementation details are in Appendix A.2.

Evaluation Metrics. We evaluate all methods along three dimensions: task performance, interaction efficiency, and LLM usage. For task performance, we report the benchmark-native metrics: task success on ALFWorld, exact success and mean reward on WebShop, and Final Pass Rate (FPR) on TravelPlanner. Interaction efficiency is measured by the number of environment steps when applicable. LLM usage is measured by the number of LLM calls and total online tokens consumed during execution.

4.2 Overall Comparison

We compare PCM with representative baselines on ALFWorld and WebShop in terms of task performance (Table 1) and online efficiency (Table 2), and further evaluate its generality on TravelPlanner (Table 3). Overall, PCM achieves strong task performance while maintaining efficient online execution across all three environments. On ALFWorld, PCM reaches success rates of 97.9% and 97.8% on Valid Seen and Valid Unseen, respectively, with only 10.9 interaction steps, 3.94 LLM calls, and 2.49k tokens per task. On WebShop, PCM achieves an exact success rate of 47.1% and a mean reward of 0.713, with 8.49 steps, 6.42 LLM calls, and 9.22k tokens per task. On TravelPlanner, PCM obtains 98.9 HC-Macro, 100 CS-Macro, and 98.9 FPR while using 1.73k tokens per query. These

Table 3: TravelPlanner validation results ($n = 180$). Tokens are recorded totals divided by 180 queries, rounded to three significant figures; k denotes 1,000 tokens. * denotes our re-implementation.

Type	Method	Backbone	HC-Macro↑	CS-Macro↑	FPR↑	Tokens↓
General	ReAct* (Yao et al., 2023)	DS-V4-flash	91.7	19.0	16.7	46.8k
General	ReflAct* (Kim et al., 2025)	DS-V4-flash	89.4	17.8	16.7	44.7k
General	ExPeL* (Zhao et al., 2024)	DS-V4-flash	85.7	16.8	13.9	51.6k
General	AWM* (Wang et al., 2025)	DS-V4-flash	92.9	22.8	20.6	54.9k
General	MACLA† (Forouzandeh et al., 2026)	Llama-2-7B	–	83.3	–	–
TP-specific‡	TriFlow (Chen et al., 2025)	GPT-4o	96.1	95.0	91.1	–
TP-specific‡	Behavior Forest (Yuan et al., 2026)	–	94.4	96.1	94.4	–
Solver-based	Formal Verification (Hao et al., 2025)	GPT-4	**98.9**	95.0	93.3	–
General	**PCM (Ours)**	DS-V4-flash	**98.9**	**100**	**98.9**	**1.73k**

† MACLA reports CS=83.3 (aggregation unspecified). ‡ TP-specific denotes TravelPlanner-specific methods.

results show that PCM provides a favorable balance between task performance and online efficiency across diverse agent environments.

On ALFWorld, PCM increases success rates from 87.2% to 97.9% on Valid Seen and from 90.3% to 97.8% on Valid Unseen. This setting requires multi-step household tasks with roughly 30 candidate actions per state under a 50-step limit (Appendix B.4), making both effective action comparison and execution important. Predictive Beliefs support action comparison through explicit action–outcome estimates, while Executable Programs encode reusable multi-step control structures. The comparable gains on Seen and Unseen tasks further suggest that PCM captures transferable action–outcome regularities and execution structures rather than task-specific solutions.

4.3 Does Memory Form Matter?

A central hypothesis of PCM is that memory effectiveness depends not only on *what* information is retained, but also on whether its representation is aligned with its computational function at reuse time. We test this with a controlled 3×3 Program Form $\times$ Belief Form design. Textual (P_T) and Executable Programs (P_E) are built from the same execution patterns but represent them as linguistic instructions versus executable structures; Descriptive (B_D) and Quantitative Beliefs (B_Q) use the same predictive evidence but represent it as textual descriptions versus explicit probabilities. Implementation details are provided in Appendix A.3. Together with no-Program (P_0) and no-Belief (B_0) conditions, this design isolates the effect of memory form while holding the underlying information fixed across the same 274 ALFWorld tasks.

Table 4: Program Form $\times$ Belief Form on ALFWorld. Rows vary Program Form and columns vary Belief Form. Darker shading indicates better performance within each panel. The four corner cells correspond to No Memory, Belief Only, Program Only, and Full PCM.

	B_0	B_D	B_Q
P_0	69.7	76.3	79.2
P_T	69.3	77.7	79.9
P_E	91.2	92.7	**97.8**

(a) Success (%) ↑

	B_0	B_D	B_Q
P_0	25.5	22.9	22.5
P_T	24.1	22.7	20.8
P_E	20.6	20.1	**10.9**

(b) Steps / Task ↓

	B_0	B_D	B_Q
P_0	17.1	40.1	100.6
P_T	22.7	44.9	98.4
P_E	**2.2**	5.1	2.5

(c) Online Tokens / Task (k) ↓

Memory form matters. Table 4 shows a clear advantage of function-aligned memory representations. Across all belief conditions, Executable Programs substantially outperform Textual Programs. Under B_Q, replacing P_T with P_E increases success from 79.9% to 97.8%, while reducing steps from 20.8 to 10.9 and online tokens from 98.4k to 2.5k. Since both forms are derived from the same underlying action patterns, this provides controlled evidence that representing the same information in forms better aligned with its computational function can substantially improve runtime effectiveness.

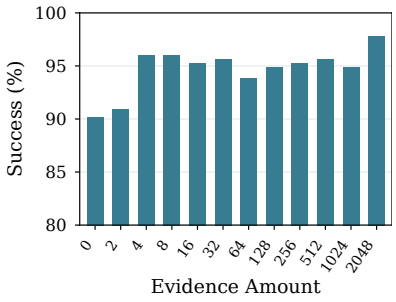
Beliefs and Programs are complementary. Quantitative Beliefs consistently outperform Descriptive Beliefs, with the largest benefit when paired with Executable Programs. Within P_E, replacing B_D with B_Q raises success from 92.7% to 97.8% and reduces steps from 20.1 to 10.9. The

component ablation shows the same pattern: Program Only reaches 91.2% success, while combining it with Quantitative Beliefs yields the best overall result of 97.8%. These results support PCM as a coupled design in which Predictive Beliefs support prediction through explicit outcome estimates, while Executable Programs support execution through reusable control structures. Details are provided in Table 7.

4.4 Analysis of Predictive Beliefs

In this section, we examine whether Predictive Beliefs are reliably estimated, actively used in runtime decision making, and progressively improved through accumulated interaction evidence.

Quality of Estimation. We evaluate whether Predictive Beliefs provide reliable estimates of local action outcomes. Coverage measures whether an estimate is available, while Brier score and NLL evaluate the quality of the predicted probabilities. All estimates are evaluated out-of-sample using statistics constructed from task-disjoint training folds. As shown in the right panel of Figure 3, Local MLE covers only 76.3% of cases, while pooled and prior-based estimators achieve full coverage at the cost of weaker predictive quality. In contrast, the proposed Shrinkage Estimation achieves 100% coverage together with the lowest Brier score (0.0716) and NLL (0.270). These results show that shrinkage effectively balances sparse local evidence with shared statistics, providing more reliable and broadly available action-outcome estimates for runtime decision making.



Estimation Method	Coverage ↑	Brier ↓	NLL ↓
Local MLE	76.3%	0.0858	0.979
Pooled MLE	100%	0.0873	0.318
Global-Prior Estimation	100%	0.0798	0.284
Shrinkage Estimation (Ours)	**100%**	**0.0716**	**0.270**

Figure 3: Success vs. evidence amount (left) and Predictive Belief estimation quality (right).

Belief-to-Action Influence. We examine whether Predictive Beliefs affect runtime action selection, rather than merely being stored as auxiliary statistics. To isolate this effect, we compare 301 matched decision points before the first action divergence between executions with and without belief use, thereby controlling for decision context. Belief use changes the executed action in 264 cases, corresponding to an action-change rate of 87.7%. This shows that Predictive Beliefs actively participate in runtime decision making rather than serving as passive statistics. Together with the task-level performance gains, these results suggest that belief-driven action changes contribute to more effective behavior.

Evidence Accumulation. We examine whether Predictive Beliefs become more effective as interaction evidence accumulates, while keeping the Executable Programs fixed. We vary the amount of evidence from 0 to 2048, where 0 denotes a neutral-belief reference with no evidence, and evaluate all settings on the same tasks. As shown in Figure 3, introducing predictive evidence quickly improves task success over the no-evidence reference, with the best performance of 97.8% achieved at 2048 samples. Although the trend is not strictly monotonic, performance remains consistently high across larger evidence budgets. These results show that Predictive Beliefs can extract useful action–outcome regularities from accumulated experience, while larger evidence budgets alone do not guarantee further performance gains.

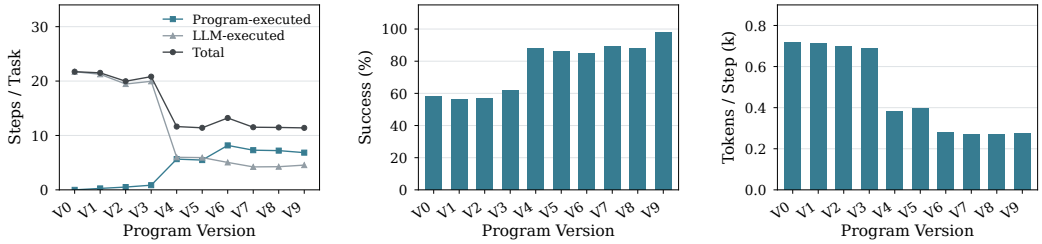
4.5 Analysis of Executable Programs

Decision-Level Execution Effectiveness. We test whether reusable action patterns are more effective when represented as executable control structures rather than textual programs. As shown in Table 5, on 400 fixed decision states, Executable Programs cover 86.0% of cases and achieve 95.9% correctness, compared with 62.8% for language-mediated reuse, while requiring no online LLM tokens. With LLM fallback on uncovered states, PCM reaches 94.5% overall correctness at only 0.456k tokens per decision. These results show that representing reusable action patterns

as executable control structures improves both action correctness and online efficiency relative to language-mediated reuse.

Table 5: Decision-level correctness and online token cost. PCM includes an executable Program and LLM fallback.

Method	Covered ↑	Uncovered ↑	Overall ↑	Tokens / Decision ↓
LLM + Text program	62.8%	85.7%	66.0%	3.15k
Program	**95.9%**	–	–	**0**
LLM Fallback	–	85.7%	–	3.26k
PCM	**95.9%**	**85.7%**	**94.5%**	**0.456k**



(a) Execution Composition (b) Task Success (c) Tokens / Step

Figure 4: Program update with fixed Belief condition. (a) Mean Program-executed, LLM-executed, and total steps per task, (b) task success, and (c) online tokens per executed step.

Program Updates. With the Belief condition fixed, we track Program versions V_0–V_9 in Figure 4. Across updates, execution progressively shifts from LLM reasoning to Program execution: the share of Program-executed steps rises from 0% to 60.1%, while LLM-executed steps decrease and trajectories become shorter. Meanwhile, task success improves overall and the average token cost per step declines. These trends indicate that Program updates progressively convert recurring execution patterns into reusable control structures rather than merely accumulating additional experience. As more behavior is captured by these structures, execution relies less on online LLM reasoning while task performance improves. Together, these results show that successive Program updates increase Program coverage, improve task success, and reduce online reasoning cost.

5 Limitations

Representation for Function is a general design principle rather than a claim that PCM is the only possible realization. PCM instantiates this principle for prediction and execution, but other memory forms may better serve different functions or environments. Exploring alternative instantiations and how to select representations for different functions remains future work.

6 Conclusion

LLM agents can improve future behavior by reusing experience, but language-stored experience may not represent action-relevant information in forms aligned with its computational functions at reuse time. We therefore introduce *Representation for Function*, the principle that memory should represent information in forms aligned with how it is used at reuse time. Following this principle, Program-Centric Memory (PCM) represents action–outcome relationships as Predictive Beliefs and reusable action patterns as Executable Programs. Across diverse environments, PCM achieves strong performance, while controlled experiments holding the underlying information fixed show that memory form substantially affects reuse effectiveness, establishing representation form as a key dimension of experience reuse in LLM agents.

AI USE STATEMENT

In this work, we used generative AI tools to formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, implement methods, and assist with translation. We did not use generative AI tools to generate synthetic datasets, develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, clean or reformat datasets, support qualitative or thematic data analysis, or interpret results; other required-disclosure tasks were not applicable to this work.

Additionally, generative AI tools were used for auxiliary tasks including drafting and revising portions of the manuscript, improving clarity and organization, preparing LaTeX, tables, figures, and references, and assisting with software coding and debugging.

The authors reviewed all AI-assisted work. AI-assisted mathematical claims and proofs were independently checked by the authors; method implementations were verified through code inspection, execution, and testing; translations were reviewed for correctness; citations were checked against source records; and reported numerical results were validated against the underlying experimental outputs. All research decisions, interpretation of results, and final scientific claims were made by the authors. The authors take full responsibility for the final content of this work, including any text, claims, code, or artifacts produced with the aid of generative AI.

REFERENCES

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.829.

Yuxing Chen, Basem Suleiman, and Qifan Chen. TriFlow: A progressive multi-agent framework for intelligent trip planning. *arXiv preprint arXiv:2512.11271*, 2025.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *ECAI 2025 – 28th European Conference on Artificial Intelligence*, pp. 2993–3000. IOS Press, 2025. doi: 10.3233/FAIA251160.

Runze Fang, Yuxuan Liang, Xinyi Wang, Jipeng Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025. URL https://arxiv.org/abs/2508.06433.

Saman Forouzandeh, Wei Peng, Parham Moradi, Xinghuo Yu, and Mahdi Jalili. Learning hierarchical procedural memory for LLM agents through Bayesian selection and contrastive refinement. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2026. URL https://arxiv.org/abs/2512.18950.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. Large language models can solve real-world planning rigorously with formal verification tools. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3434–3483. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.176.

Jeonghye Kim, Sojeong Rhee, Minbeom Kim, Dohyung Kim, Sangmook Lee, Youngchul Sung, and Kyomin Jung. ReflAct: World-grounded decision making in LLM agents via goal-state reflection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33433–33465. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1697.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-Pro: Learning reusable skills from experience via non-parametric PPO for LLM agents. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. URL https://arxiv.org/abs/2602.01869.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://arxiv.org/abs/2303.11366.

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*, 2021.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://arxiv.org/abs/2409.07429.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. TravelPlanner: A benchmark for real-world planning with language agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 54590–54613. PMLR, 2024.

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*, volume 38, 2025. URL https://arxiv.org/abs/2502.12110.

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*, volume 35, 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*, 2023. URL https://arxiv.org/abs/2210.03629.

Duanyang Yuan, Sihang Zhou, Yanning Hou, Xiaoshu Chen, Haoyuan Chen, Ke Liang, Jiyuan Liu, Chuan Ma, Xinwang Liu, and Jian Huang. Decoupled travel planning with Behavior Forest. *arXiv preprint arXiv:2604.21354*, 2026.

Guibin Zhang, Miao Fu, Guancheng Wan, Mengyue Yu, Kaixin Wang, and Shirui Yan. G-Memory: Tracing hierarchical memory for multi-agent systems. *arXiv preprint arXiv:2506.07398*, 2025. URL https://arxiv.org/abs/2506.07398.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExPeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://arxiv.org/abs/2308.10144.

CONTENTS

1 Introduction **1**

2 Related Work **2**

3 Program-Centric Memory **3**

3.1 Problem Formulation . . . 3

3.2 Predictive Beliefs . . . 3

3.3 Executable Programs . . . 4

3.4 Memory Execution . . . 5

4 Experiments **5**

4.1 Experimental Setup . . . 5

4.2 Overall Comparison . . . 6

4.3 Does Memory Form Matter? . . . 7

4.4 Analysis of Predictive Beliefs . . . 8

4.5 Analysis of Executable Programs . . . 8

5 Limitations **9**

6 Conclusion **9**

A Implementation Details **13**

A.1 Memory Construction Data . . . 13

A.2 TravelPlanner Instantiation . . . 13

A.3 Program Form $\times$ Belief Form Implementation . . . 13

A.4 Derivation of the Predictive-Belief Shrinkage Estimator . . . 14

A.5 Predictive-Belief Evaluation Metrics . . . 15

A.6 Compared Predictive-Belief Estimators . . . 16

B Additional Results **16**

B.1 Fixed-Program Experience Curve . . . 16

B.2 PCM Component Ablation . . . 17

B.3 Program Revision Case Study and Analysis . . . 17

B.4 ALFWorld Action-Space Statistics . . . 18

C Prompts **18**

C.1 Runtime Actor Prompt . . . 18

C.2 Optional Memory Context . . . 19

C.3 Recent-History Serialization . . . 19

C.4 Frozen Text Program Representation (P_T) . . . 20

C.5 Prompt Instantiation Across Conditions . . . 20

A Implementation Details

A.1 Memory Construction Data

All persistent memory components used in our ALFWorld experiments, including Executable Programs and Predictive Beliefs, are constructed exclusively from the designated Construction Set. The Construction Set is a subset of the ALFWorld training pool and is disjoint from the Evaluation Pool at the exact task-ID level. No evaluation task is used to construct or revise persistent memory, and no evaluation trajectory, action outcome, execution feedback, or success signal is incorporated into memory. During evaluation, the constructed memory is frozen and only the transient execution context is updated.

Table 6: ALFWorld memory construction and evaluation data by task family.

	Train Pool		Evaluation Pool		
Task Family	Construction Set	Full Train	Valid Seen	Valid Unseen	Evaluation Total
Clean and place	375	650	27	31	58
Cool and place	307	533	25	21	46
Heat and place	265	459	16	23	39
Look in light	177	308	13	18	31
Pick and place	455	790	35	24	59
Pick two and place	469	813	24	17	41
Total	**2,048**	**3,553**	**140**	**134**	**274**

A.2 TravelPlanner Instantiation

We instantiate PCM on TravelPlanner as a program-centric framework that separates semantic understanding from deterministic execution. For each query, the LLM is invoked to extract evidence-grounded constraints, including budget, party size, accommodation preferences, transportation restrictions, and cuisine requirements. The LLM does not directly generate the final itinerary or access the reference information. The execution logic is automatically generated and progressively revised by PCM through the Program Update mechanism, based on accumulated execution trajectories and feedback.

The deterministic program then constructs candidate sets from the public reference information and selects transportation, accommodations, restaurants, and attractions under these constraints to produce a complete multi-day itinerary. Before submission, the generated itinerary is further checked for route consistency, budget feasibility, accommodation requirements, transportation restrictions, and output-format compliance. We do not rely on a manually designed TravelPlanner-specific planner; instead, execution strategies and constraint-handling logic are automatically formed from experience by PCM and progressively updated into directly executable programs, while the runtime LLM primarily serves as a natural-language constraint parser.

A.3 Program Form $\times$ Belief Form Implementation

To isolate the effect of Program form, the Textual Program (P_T) and Executable Program (P_E) conditions are derived from the same source procedure. Both preserve the same reusable procedural content, including applicability conditions, entity bindings, decision rules, progress updates, recovery behavior, and termination criteria. Their difference lies in representation: P_T exposes this procedure as frozen linguistic guidance to the runtime LLM, whereas P_E realizes the same procedure as executable control structure.

The Belief-form comparison follows the same principle. Descriptive Beliefs (B_D) and Quantitative Beliefs (B_Q) are constructed from the same action–outcome evidence and context definitions. B_D expresses the resulting predictive information descriptively, whereas B_Q exposes the corresponding estimates as explicit probabilities. No additional interaction evidence is introduced when changing the belief form.

Thus, the controlled comparisons vary how the same underlying information is represented and reused, rather than changing the source experience available to each condition.

A.4 Derivation of the Predictive-Belief Shrinkage Estimator

Consider a context cell j with a finite outcome space $\mathcal{Y} = \{y_1, \ldots, y_J\}$. Let

$$\mathbf{p}_j = (p_{j1}, \ldots, p_{jJ}) \in \Delta^{J-1}$$

denote its unknown outcome-probability vector, and let $\mathbf{c}_j = (c_{j1}, \ldots, c_{jJ})$ denote the observed outcome counts, with $n_j = \sum_{r=1}^{J} c_{jr}$.

Let $\mathbf{p}_{\mathrm{shared},j}$ be the shared outcome distribution estimated from historical observations under the predefined grouping rule. Conditioning on this shared distribution, we place the Dirichlet prior

$$\mathbf{p}_j \sim \mathrm{Dirichlet}\left(\kappa \mathbf{p}_{\mathrm{shared},j}\right),$$

where $\kappa > 0$ controls the concentration around the shared distribution. We assume $p_{\mathrm{shared},j,r} > 0$ for all r.

Given $\mathbf{p}_j$, the observed counts follow a multinomial likelihood,

$$\mathbf{c}_j \mid \mathbf{p}_j \sim \mathrm{Multinomial}(n_j, \mathbf{p}_j).$$

Posterior distribution. The Dirichlet prior has density

$$p(\mathbf{p}_j) \propto \prod_{r=1}^{J} p_{jr}^{\kappa p_{\mathrm{shared},j,r} - 1},$$

while the multinomial likelihood satisfies

$$p(\mathbf{c}_j \mid \mathbf{p}_j) \propto \prod_{r=1}^{J} p_{jr}^{c_{jr}}.$$

Therefore,

$$p(\mathbf{p}_j \mid \mathbf{c}_j) \propto p(\mathbf{c}_j \mid \mathbf{p}_j) p(\mathbf{p}_j) \propto \prod_{r=1}^{J} p_{jr}^{c_{jr} + \kappa p_{\mathrm{shared},j,r} - 1}.$$

By Dirichlet–multinomial conjugacy,

$$\mathbf{p}_j \mid \mathbf{c}_j \sim \mathrm{Dirichlet}\left(\mathbf{c}_j + \kappa \mathbf{p}_{\mathrm{shared},j}\right).$$

Posterior-mean estimator. The posterior mean of the r-th outcome probability is

$$\mathbb{E}[p_{jr} \mid \mathbf{c}_j] = \frac{c_{jr} + \kappa p_{\mathrm{shared},j,r}}{n_j + \kappa}.$$

Hence the posterior-mean estimate is

$$\widehat{\mathbf{p}}_j = \frac{\mathbf{c}_j + \kappa \mathbf{p}_{\mathrm{shared},j}}{n_j + \kappa}.$$

For $n_j > 0$, define the local maximum-likelihood estimate

$$\widetilde{\mathbf{p}}_j = \frac{\mathbf{c}_j}{n_j}.$$

Then

$$\widehat{\mathbf{p}}_j = \frac{n_j}{n_j + \kappa} \widetilde{\mathbf{p}}_j + \frac{\kappa}{n_j + \kappa} \mathbf{p}_{\mathrm{shared},j}.$$

Defining

$$\lambda_j = \frac{n_j}{n_j + \kappa},$$

we obtain

$$\widehat{\mathbf{p}}_j = \lambda_j \widetilde{\mathbf{p}}_j + (1 - \lambda_j) \mathbf{p}_{\mathrm{shared},j}.$$

Thus, the posterior mean is a convex combination of the local maximum-likelihood estimate and the shared distribution.

Shrinkage property. For $n_j > 0$, subtracting $\mathbf{p}_{\text{shared},j}$ from both sides gives

$$\widehat{\mathbf{p}}_j - \mathbf{p}_{\text{shared},j} = \lambda_j \left(\widetilde{\mathbf{p}}_j - \mathbf{p}_{\text{shared},j}\right).$$

Since $0 \le \lambda_j < 1$ for finite n_j and $\kappa > 0$, for any norm $\|\cdot\|$,

$$\|\widehat{\mathbf{p}}_j - \mathbf{p}_{\text{shared},j}\| = \lambda_j \|\widetilde{\mathbf{p}}_j - \mathbf{p}_{\text{shared},j}\|.$$

Therefore, the posterior estimate is explicitly shrunk toward the shared distribution. The amount of shrinkage decreases with local support:

$$\lambda_j = \frac{n_j}{n_j + \kappa} \to 1 \qquad \text{as} \qquad n_j \to \infty.$$

When $n_j = 0$, the posterior mean reduces to

$$\widehat{\mathbf{p}}_j = \mathbf{p}_{\text{shared},j}.$$

A.5 Predictive-Belief Evaluation Metrics

Predictive Beliefs estimate the probability that a local grounded consequence will succeed under a given action context, rather than the probability of complete-episode success. For each evaluation record, the estimator outputs a probability p_i for the corresponding consequence. We evaluate these estimates using coverage, Brier score, and negative log-likelihood (NLL).

Coverage. Coverage measures the fraction of evaluation records for which the estimator can produce a valid probability estimate:

$$\text{Coverage} = \frac{\#\{\text{records with an available estimate}\}}{\#\{\text{evaluation records}\}}.$$

It captures whether an estimate is available at all, which is particularly important when experience is sparse and some action contexts have little or no direct support.

Brier score. Brier score measures the squared error between the predicted probability and the observed consequence:

$$\text{Brier} = \frac{1}{M} \sum_{i=1}^{M} \left(p_i - \mathbb{I}[\text{consequence succeeds}]\right)^2,$$

where M is the number of scorable records. Lower values indicate better probability estimates. Because the loss is bounded, each prediction contributes a limited amount of error.

Negative log-likelihood. NLL measures the probability assigned to the outcome that is actually observed:

$$\text{NLL} = -\frac{1}{M} \sum_{i=1}^{M} \left[\mathbb{I}[\text{success}] \log \widetilde{p}_i + \mathbb{I}[\text{failure}] \log(1 - \widetilde{p}_i)\right],$$

where $\widetilde{p}_i$ is clipped to $[10^{-6}, 1 - 10^{-6}]$ only when computing NLL for numerical stability. Lower values are better. In contrast to the bounded Brier score, NLL places a substantially larger penalty on highly confident incorrect estimates.

Why these metrics? Predictive Beliefs are intended to preserve the strength of an action–outcome relationship, rather than only whether the most likely outcome is predicted correctly. Classification accuracy discards this probability information: for example, probabilities of 0.51 and 0.99 may imply the same prediction while representing substantially different beliefs. We therefore use Brier score and NLL as the primary measures of estimation quality, while coverage measures whether such an estimate is available in the first place.

Interpretation. The three metrics characterize complementary properties of Predictive Beliefs. Coverage indicates whether retained experience supports an estimate for the current context; Brier score measures how closely the estimated probability agrees with observed consequences; and NLL reveals whether the estimator makes strongly overconfident errors. Together, they assess whether Predictive Beliefs provide broadly available and reliable probabilistic estimates of local action consequences.

A.6 Compared Predictive-Belief Estimators

All estimators predict the same quantity: the probability that the current grounded action/context produces a successful local consequence. For each evaluation record, statistics are constructed only from the other folds of the task-disjoint split. Let s_{local} and n_{local} denote the number of successful and total observations for the exact local context. Let S_{all} and N_{all} denote the corresponding counts pooled over the available training folds.

Local MLE. Local MLE uses only observations from the exact matching context:

$$p_{\text{local}} = \frac{s_{\text{local}}}{n_{\text{local}}}.$$

It preserves context-specific differences, but is unavailable when the local context has no support and can become unstable when support is sparse.

Pooled MLE. Pooled MLE ignores context distinctions and estimates a single probability from all training records:

$$p_{\text{pooled}} = \frac{S_{\text{all}}}{N_{\text{all}}}.$$

It provides a stable full-coverage estimate, but largely reflects the global base rate and does not capture variation across local contexts.

Global-Prior Estimation. Global-Prior Estimation combines local evidence with a smoothed global parent probability:

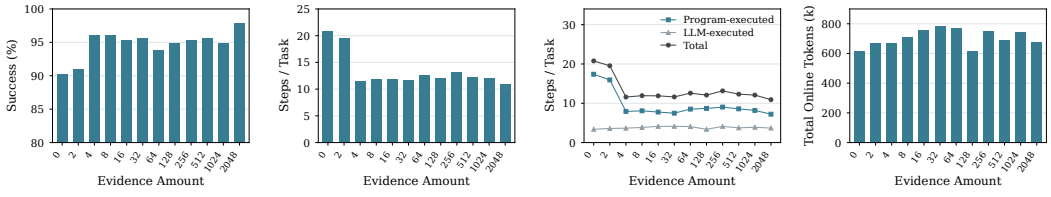
$$p_{\text{global-prior}} = \frac{s_{\text{local}} + \kappa p_{\text{parent}}}{n_{\text{local}} + \kappa},$$

where p_{parent} is estimated from the pooled training evidence and κ controls the strength of shrinkage toward the global parent. When local support is small, the estimate relies more strongly on the parent; as local support increases, it increasingly reflects context-specific evidence. With no local support, the estimate falls back to the parent distribution.

These baselines represent three natural strategies for sparse predictive estimation: exact-context estimation, global pooling, and local estimation regularized by a global prior.

B Additional Results

B.1 Fixed-Program Experience Curve



(a) Task Success (b) Trajectory Length (c) Execution Composition (d) Model Usage

Figure 5: ALFWorld fixed-program evidence-budget analysis on 274 tasks per point. 0 is the neutral-belief reference with no evidence. Panels show absolute task success, mean trajectory length, approximate LLM/program step composition derived from rounded mean steps and action shares, and measured total online tokens.

Overall, increasing evidence tends to improve success and reduce trajectory length, although the trend is not strictly monotonic.

B.2 PCM Component Ablation

Table 7 reports the P_0B_0, P_0B_Q, P_EB_0, and P_EB_Q configurations from the current nine-cell experiment in Section 4.3. Each configuration contains 274 tasks; online tokens and steps are task means. Values are rounded to three significant figures; k denotes 1,000 tokens.

Table 7: Component ablation of PCM. We compare the no-memory baseline with Program-only, Belief-only, and full PCM configurations in terms of task success, interaction steps, and online token usage.

Configuration	Program	Belief	Success (%)	Mean Steps	Tokens / Task
No Memory	–	–	69.7	25.5	17.1k
Belief Only	–	Quantitative	79.2	22.5	101k
Program Only	Executable	–	91.2	20.6	**2.19k**
Full PCM	Executable	Quantitative	**97.8**	**10.9**	2.49k

B.3 Program Revision Case Study and Analysis

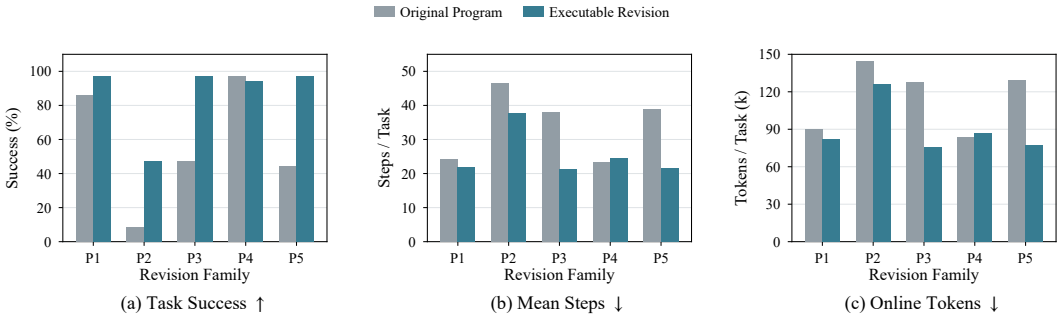


Figure 6: **Program revision case study.** Outcomes across five revision families comparing the original Program with its executable revision. The three panels report task success, mean environment steps, and online tokens per task.

We further selected five concrete cases to examine the practical effectiveness of Program revision within the broader Program update framework. For each case, the model first analyzed the problem in the current Program and proposed a corresponding revision proposal. It then automatically generated validation tests based on that proposal to check whether the updated Program correctly implemented the intended modification. The five cases cover different types of program-logic issues, including target search, handling closed containers, typed-role search, heating-device search, and fallback behavior after execution failure.

Importantly, these validation tests are designed only to verify whether the revision proposal has been correctly implemented. They use smoke inputs to test program behavior, without interacting with the actual environment, and the test cases themselves are generated by the model from the revision proposal. Therefore, this stage evaluates implementation correctness within Program update, rather than the real task performance of the updated Program. In all five cases, the Program revisions passed all validation tests and correctly implemented their intended modifications.

To control for differences in model capability, all Program revisions were performed using the same DS-V4-flash model. We then evaluated the Programs before and after update on the same 36-task panel. Across the five cases, the average success rate increased from 56.7% to 86.7%, an improvement of 30.0 percentage points. The average number of environment steps decreased from 34.2 to 25.4, a reduction of 25.7%, while the average online token cost decreased from approximately 115.1k to 89.5k tokens per task, a reduction of 22.2%.

However, in one case, the revision correctly implemented the proposed change and passed all validation tests, yet the success rate in the real environment slightly decreased from 97.2% to 94.4%. This indicates that a Program revision can be locally correct without necessarily improving end-to-

end task performance. Newly introduced program logic may still interact with other control logic or environment states in ways that are not covered by simple validation tests.

Overall, these results show that Program update can effectively improve existing Programs, while also highlighting that lightweight offline validation cannot replace real environment interaction. Continued Program improvement therefore requires a closed loop that combines offline revision with online execution feedback: the offline stage analyzes problems, revises the Program, and validates implementation correctness, while online execution data determines whether the revision actually improves task performance and provides new evidence for subsequent Program revisions.

B.4 ALFWorld Action-Space Statistics

To characterize the action-selection space in ALFWorld, we collect statistics from the complete 274-task ReflAct formal trajectories. The action-level statistics are computed over 6,402 decision states. Table 8 summarizes the resulting branching factors and scene-level search spaces.

Table 8: ALFWorld action-space statistics from the complete 274-task ReflAct formal trajectories. Action-level statistics are aggregated over 6,402 decision states. The 10–90% column reports the 10th and 90th percentiles.

Metric	Median	Mean	10–90%	Max
Admissible actions per step	30	30.44	18–43	68
`go to` candidates per step	24	24.28	12–38	54
Navigable receptacle instances per scene	25	23.25	13–36	54
Receptacle types per scene	10	9.42	7–11	12
Openable container instances per scene	11	11.51	4–24	42

In ALFWorld, *receptacle* is used in a broad sense and includes not only containers such as cabinets, drawers, refrigerators, microwaves, and safes, but also surfaces or locations such as countertops, dining tables, shelves, beds, and sofas, as well as functional devices such as sink basins, stove burners, and coffee machines. The environment defines 37 receptacle types in total, including seven movable receptacle types (e.g., bowls, boxes, cups, mugs, plates, pans, and pots). Across the 274 tasks considered here, 29 navigable receptacle types are observed.

Overall, a typical scene contains roughly 25 navigable search locations, including about 11 openable containers, while each decision state exposes around 30 admissible actions. These statistics provide additional context for the action-selection setting discussed in the main text.

C Prompts

This section reports the frozen runtime actor template and the textual Program representation used in the language-mediated conditions. Both arms receive the task goal, current public observation, admissible-action interface, bounded interaction history, and applicable source-matched memory context. The Language arm uses the runtime actor to select and emit the next admissible action. In the Hybrid arm, the same actor is invoked only when the Executable Program abstains or when the current state falls outside its public coverage contract.

The prompts do not reveal evaluation outcomes or private simulator state. Parsing and admission-repair attempts are logged rather than silently discarded.

C.1 Runtime Actor Prompt

All language-mediated conditions use the same frozen ReAct actor template. At each environment step, the actor receives the task goal, optional cross-task memory context, current public observation, the five most recent interaction-history entries, and the current list of admissible actions. The actor is required to emit exactly one admissible action.

The runtime actor template is:

```
You are controlling an ALFWorld text-only household agent.
```

```
Task goal:
{task_goal}

{optional_memory_block}

Current observation:
{observation}

Recent history:
{recent_history}

Valid actions:
- {action_1}
- {action_2}
- ...

Choose exactly one next text action for the environment.
Do not use markdown.
Keep the Thought to at most 256 words; do not write a plan or
explanation.
Copy the Action verbatim from the valid actions list.
Stop immediately after it.
Never emit a `think:` action.

Your response must contain exactly this format:

Thought: <brief reasoning about the next step>
Action: <one valid text action>
```

C.2 Optional Memory Context

The placeholder {optional_memory_block} is instantiated only when cross-task memory is available in the corresponding experimental condition. Depending on the condition, the block may contain the textual Program representation, a Descriptive Belief, or both. Components that are not present are omitted rather than replaced with empty fields.

When a textual Program is available, the memory block contains:

```
Cross-task historical evidence (not a current-scene fact):

{text_program_if_present}
```

When a Descriptive Belief is available, it is serialized as:

```
DESCRIPTIVE_B={descriptive_b_json}
```

When both memory components are present, they are inserted into the same cross-task memory block:

```
Cross-task historical evidence (not a current-scene fact):

{text_program_if_present}

DESCRIPTIVE_B={descriptive_b_json}
```

C.3 Recent-History Serialization

Immediately before each actor call, the five most recent interaction entries are serialized into the {recent_history} field. Each history entry follows the format below:

```
Step {i}:
Thought: {previous_thought}
```

```
Action: {previous_action}
Observation after action: {observation_after}
Reward: {reward}, Done: {done}
```

If fewer than five previous interaction steps are available, all available entries are included in chronological order.

C.4 Frozen Text Program Representation (P_T)

The Textual Program condition uses a frozen natural-language representation of the source execution pattern. The representation is descriptive only: it is provided to the runtime actor as language context and cannot directly execute control logic or issue environment actions.

The frozen Textual Program representation is:

```
PROGRAM TEXT

program_mode: descriptive_text_only
executable_program_object: forbidden

PRECONDITIONS:
Use only public task goal, observation, legal actions, and
confirmed feedback.

BINDING:
Bind an observed target instance before manipulation;
preserve object and destination identity.

CANDIDATES:
Enumerate legal public navigation and manipulation candidates;
exclude already searched locations.

RANKING:
Order candidates by information provided.

COST:
Prefer a single admissible action per step; do not invent
actions outside the public action list.

ABSTENTION:
When the fixed public procedure cannot establish an admissible
action, abstain to the common LLM fallback.

PROGRESS:
Update only from observed transitions, confirmed held/placed
state, and public action feedback.

RECOVERY:
Use the common parser/admission-repair contract once;
unresolved responses remain method failures.

TERMINATION:
Stop on public success, public terminal failure, or the frozen
50-step limit.

FALLBACK:
The text interface is descriptive only; the LLM submits every
exact action.
```

C.5 Prompt Instantiation Across Conditions

The runtime actor template is shared across all language-mediated conditions; only the optional memory context varies.

In the no-memory condition, the optional memory block is omitted. In the Textual Program condition, the block contains the frozen textual Program representation P_T. In the Descriptive Belief condition, it contains the serialized Descriptive Belief B_D. When both components are enabled, both P_T and B_D are included in the same cross-task memory block.

The Hybrid condition uses the same runtime actor only as a fallback. When an Executable Program applies to the current state and produces an admissible action, no runtime actor call is made for that step. The actor is invoked only when the Executable Program abstains or when the current state falls outside its public coverage contract.