FROM MEMORY TO ACTION: EXECUTABLE AND SELF-EVOLVING CONTRACT GRAPHS FOR TOOL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using agents can improve future decisions by extracting reusable knowledge from past interactions and retrieving it when needed. Existing approaches often distill such interactions into textual experience that is retrieved as guidance for future tasks. However, this form of experience reuse leaves unclear how retrieved experience should be executed, composed with other experience, and updated from feedback. We introduce ContractFlow, a framework for transforming accumulated experience into executable, self-evolving knowledge. ContractFlow first compiles experience into executable contracts specifying goals, prerequisites, stable knowledge, runtime variables, tool actions, and completion criteria. It then organizes these contracts into a dependency graph that coordinates their execution according to the current task state. Within each episode, the graph guides observation gathering, binds variables to current evidence, and permits actions only when their prerequisites and execution conditions are satisfied, while tracking completion. Across episodes, successful executions, failures, and local repairs drive updates to contract nodes and relations, allowing subsequent tasks to reuse a refined execution structure. This process preserves reusable action rules across episodes, while keeping runtime-specific variables and completion states local to each episode, making the same graph both an execution structure and an object of continual improvement. Across three tool-use benchmarks, ContractFlow improves performance over strong baselines, with the largest gains on state-dependent action selection. These results support a shift from treating experience as textual advice to representing it as executable knowledge that is organized, tested, and refined through action.

1 INTRODUCTION

Tool-using agents increasingly improve not only through better pretrained models, but also by learning from their own interaction histories. Recent experience-guided agents distill successful and failed trajectories into reusable lessons, strategies, or workflows that can be recalled on later tasks. ExpeL extracts natural-language insights from prior experience, Agent Workflow Memory abstracts recurring action sequences, and ReasoningBank continually distills reasoning strategies from both successful and failed trajectories (Zhao et al., 2023; Wang et al., 2025; Ouyang et al., 2025). Together, these methods establish a powerful paradigm: an agent need not solve every task from scratch when relevant experience can be accumulated and reused.

Yet retrieving useful experience is not the same as being able to execute it. A recalled lesson may describe what worked before, and a retrieved workflow may suggest a promising sequence of operations, but neither by itself establishes that the same action is valid in the current state. The agent must still determine whether the experience applies, reacquire state-dependent information, bind abstract variables to current entities, resolve dependencies among multiple pieces of experience, and decide when an intended effect has actually been achieved. In most experience-guided systems, these execution semantics are reconstructed implicitly by the language model at inference time. We refer to this mismatch as the *experience–execution gap*: existing methods have made experience increasingly effective to store, retrieve, and refine, while leaving the transition from remembered knowledge to

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.

state-grounded action largely implicit. Our central question is therefore: *how can experience be represented as an object that supports direct execution, instead of serving only as advice?*

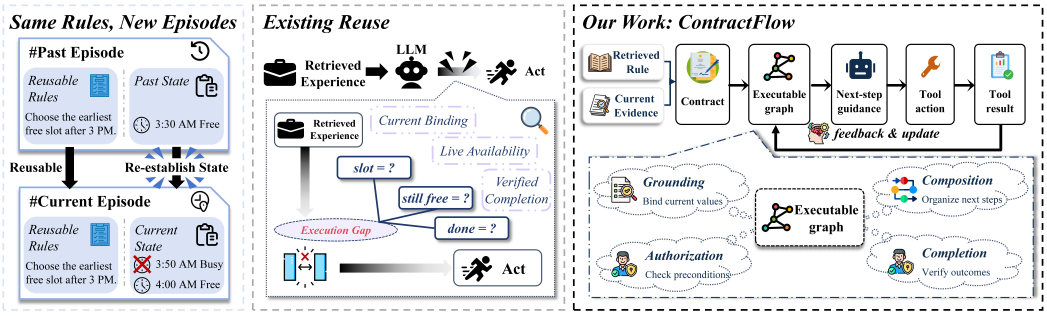

Figure 1: From retrieved advice to executable experience in CONTRACTFLOW. Retrieved advice leaves current applicability, bindings, composition, and completion implicit; ContractFlow makes these execution conditions explicit.

We introduce CONTRACTFLOW, a framework that turns accumulated agent experience into *executable, state-grounded knowledge*. Its basic unit is an executable *Contract*: a compact rule that records when an experience applies, what evidence it needs, which variables must be grounded, what action it can produce, and how completion is verified. A Contract separates reusable goals, constraints, and action rules from episode-specific entities, observations, bindings, and completion states. It therefore preserves the structure of successful behavior while requiring the current task to supply its own execution state.

Executable experience also requires composition. Long-horizon tool use rarely follows from one isolated rule: one operation may establish evidence required by another, an observation may resolve variables for several downstream actions, and completion of one subgoal may enable a different branch of execution. CONTRACTFLOW therefore organizes Contracts into an *Executable Control Graph*. Its nodes represent executable units of experience, while typed relations encode prerequisites, evidence flow, continuation, completion, and recovery structure. The graph is evaluated against the live execution state instead of being treated as a static collection of retrieved memories. At each step, it determines which Contracts are currently executable and, equally importantly, what missing observation, binding, or prerequisite prevents the others from executing. This distinction separates CONTRACTFLOW from graph-based memory systems whose primary role is to organize and retrieve related experience (Xu et al., 2025; Wang et al., 2026). A memory graph structures what knowledge is connected; an Executable Control Graph additionally defines how that knowledge participates in action. The graph is therefore not merely retrieved as context—its state is advanced through execution. For example, when a later traveler asks to reuse an earlier traveler's hotel, the system must first check whether the earlier plan provides a valid current binding before applying the dependency.

Making execution explicit creates a second opportunity: execution itself becomes structured supervision for improving memory. Conventional trajectory-level feedback identifies whether an episode succeeded or failed, but often leaves unclear which reusable knowledge should be credited, revised, or extended. In CONTRACTFLOW, actions are produced through identifiable Contracts and control relations, allowing outcomes to be attributed to the parts of the graph that actually participated in execution. Successful paths can provide evidence for reusable workflows, failed continuations can expose incorrect or insufficient dependencies, and successful local recoveries can be abstracted into reusable repair structure. These updates are localized to the affected Contracts and relations without requiring the agent to rewrite its memory as an undifferentiated whole. At the same time, episode-local bindings and completion states are discarded instead of being mistaken for persistent knowledge.

The same representation operates at two timescales. Within an episode, the graph tracks evidence, bindings, and completion; across episodes, it accumulates execution evidence and reusable repairs. CONTRACTFLOW thus connects experience reuse, runtime control, and feedback-driven improvement in one persistent representation.

Our contributions are fourfold:

- We identify the *experience–execution gap* in experience-guided agents: retrieving useful past knowledge does not by itself specify how that knowledge should be grounded, composed, executed, and completed in the current environment. We formulate executable experience as a distinct layer between agent memory and tool action.
- We introduce executable Contracts that separate persistent execution knowledge from episode-local state, and compose them into an Executable Control Graph whose typed relations encode prerequisites, evidence flow, continuation, completion, and recovery. This representation makes the applicability and execution state of reusable experience explicit.
- We develop an execution-grounded evolution mechanism that attributes runtime outcomes to the Contracts and relations that participated in execution. Successful paths, failures, and verified local repairs can thereby refine the reusable graph without globally rewriting unrelated memory.
- We evaluate CONTRACTFLOW on APOLLO, CROSSEP-TOOL, and MemoryArena, covering multi-domain tool calling, cross-episode tool use, and stateful group-travel modification. CONTRACTFLOW achieves 77.50% exact accuracy on APOLLO, 71.50% success on CROSSEP-TOOL, and 16.51% changed-slot accuracy on MemoryArena, while execution-structure comparisons analyze how dependency organization, recovery, and online evolution support these gains.

2 RELATED WORK

Prior work has developed several ways for agents to learn from and reuse past interactions. ExpeL and Reflexion extract natural-language lessons or feedback from prior attempts, while Agent Workflow Memory and ReasoningBank reuse recurring action patterns and reasoning strategies (Zhao et al., 2023; Shinn et al., 2023; Wang et al., 2025; Ouyang et al., 2025). A complementary line of work organizes experience with structured memories and graphs: A-MEM dynamically links memory notes, Associa and GAM support associative retrieval and event progression, and SAGE combines graph-based retrieval with feedback-driven memory evolution (Xu et al., 2025; Zhang et al., 2025; Wu et al., 2026; Wang et al., 2026). These approaches improve how agents extract, organize, retrieve, and refine experience.

Related graph-based approaches organize and update reusable behavior, whereas CONTRACTFLOW focuses on making retrieved experience executable through current-state grounding, eligibility, completion, and recovery.

Our focus is the interface between persistent experience and state-grounded tool execution. Retrieved lessons and workflows provide useful guidance, while runtime agent frameworks such as ReAct and Agent-C regulate interaction within the current task (Yao et al., 2023; Kamath et al., 2025). They do not by themselves specify how reusable experience should be checked against current evidence, connected through execution dependencies, or marked complete. CONTRACTFLOW addresses this interface with executable Contracts and an Executable Control Graph: Contracts expose the conditions, bindings, actions, and completion criteria of reusable experience, while typed relations organize dependencies, evidence flow, recovery, and feedback-driven updates.

3 CONTRACTFLOW

CONTRACTFLOW represents reusable experience as a persistent execution structure that can be instantiated for a new task and revised from subsequent execution outcomes. The framework proceeds in three stages. First, reusable experience is encoded as executable Contracts with explicit execution semantics. Second, Contracts are composed into an Executable Control Graph that coordinates cross-Contract dependencies, evidence flow, and runtime eligibility. Third, execution outcomes are attributed to the Contracts and relations that participated in the trajectory, providing localized evidence for updating the persistent graph used in future episodes.

The same representation operates at two timescales. Let $\mathcal{G}_n^K$ denote the persistent graph available before episode n, and let $\mathcal{G}_{n,k}^I$ denote its episode-specific execution instance at runtime step k. The persistent graph stores reusable Contracts, control relations, and accumulated execution evidence,

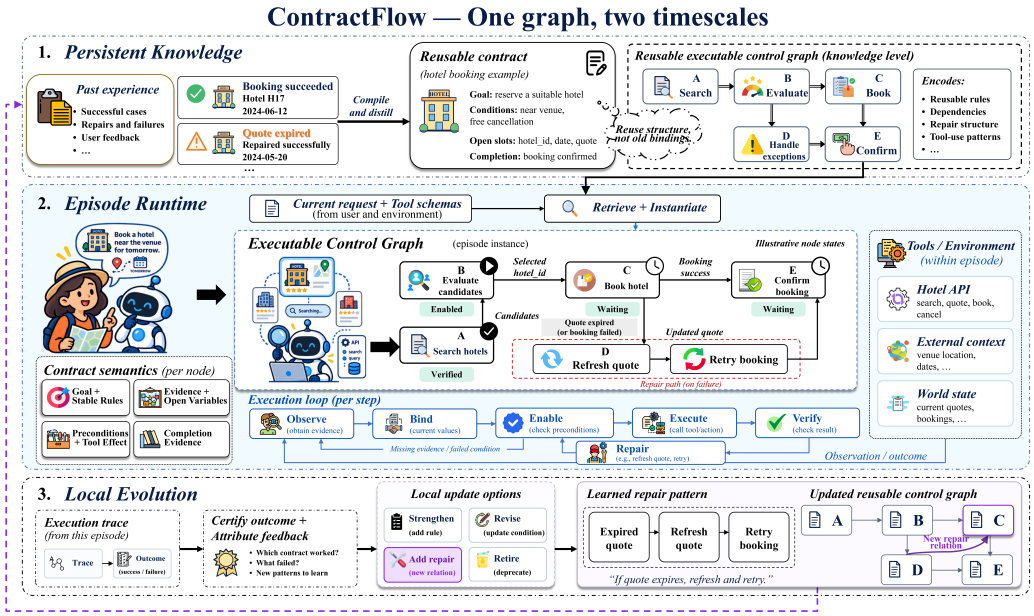

Figure 2: ContractFlow: one graph operating across persistent knowledge, episode runtime, and local evolution.

while its runtime instance maintains the current request and episode state. For request q_n, the execution interface is:

$$\mathcal{G}^I_{n,0} = \text{Instantiate}\big(\text{Retrieve}(\mathcal{G}^K_n, q_n), q_n\big)\,, \tag{1}$$

$$(y_n, \tau_n) = \text{Execute}(\mathcal{G}^I_{n,0}, q_n, \mathcal{T}), \tag{2}$$

$$\mathcal{G}^K_{n+1} = \text{Evolve}\big(\mathcal{G}^K_n, \text{Certify}(\tau_n)\big)\,. \tag{3}$$

Here, y_n is the task outcome and τ_n records the corresponding tool interactions, observations, and runtime decisions. Within an episode, $\mathcal{G}^I_{n,k}$ evolves only through execution state; across episodes, $\mathcal{G}^K_n$ evolves the reusable knowledge from which future execution instances are constructed.

Runtime realization. ContractFlow defines a shared interface for compiling prior experience into Contracts, grounding them in episode-local state, controlling graph-level eligibility, and attributing execution outcomes. Environment adapters expose tool schemas, observations, and validity checks through this shared interface; the Contract and control semantics remain common across benchmarks.

3.1 Executable Contracts

Natural-language experience can describe what worked previously, but does not by itself specify when that experience applies, which values must be reacquired from the current environment, or what evidence is required before the associated action may execute. ContractFlow addresses this gap by compiling reusable experience into persistent executable Contracts. A Contract is defined as

$$c^K_i = \langle g_i, D_i, S_i, V_i, \phi_i, e_i, \psi_i \rangle\,, \tag{4}$$

where g_i denotes a local goal; D_i specifies Contract-local evidence requirements and execution prerequisites; S_i contains knowledge that is stable across episodes; and V_i contains volatile variables whose values must be grounded in the current episode. The predicate ϕ_i specifies the applicability conditions of the Contract, e_i is a parameterized tool-effect template, and ψ_i defines the evidence required to establish completion. Together, these fields encode a reusable execution rule instead of a previously instantiated tool call.

Compilation preserves reusable goals, preferences, constraints, and procedures while replacing episode-specific arguments with open variables. For example, a historical trajectory may reveal a reusable procedure for selecting and operating on an eligible object. The procedure itself may persist, but the identifier of the historical object and its observed availability do not. Instead, the resulting

Contract records which variables must be instantiated, which sources may provide them, and which conditions those values must satisfy. CONTRACTFLOW therefore preserves the structure of successful behavior without replaying the incidental state in which that behavior was originally observed.

Instantiating a persistent Contract creates an episode-local execution state with binding map $b_{i,n,k}$. A volatile variable may be bound only from admissible evidence in the current request or current tool observations:

$$b_{i,n,k}(v) = x \quad \Longrightarrow \quad \exists s \in \{q_n, o_{n,1}, \ldots, o_{n,k}\} : \mathrm{Supports}(s, v, x), \qquad v \in V_i. \tag{5}$$

The support relation verifies the expected source and type of the binding instead of accepting the mere occurrence of a matching value in the context. Variables for which no admissible support is available remain unresolved. Historical bindings are therefore never treated as current execution state. A prior plan or binding is not copied into a new action by default; it must first be judged against the current request, observations, and relevant dependency. Contracts thus make execution constraints explicit: the same reusable rule can be applied across episodes while stale or unsupported values are prevented from being carried forward.

We summarize the Contract-local execution conditions with a gate $G_{n,k}(c_i)$. Let $\mathrm{Sat}(d; q_n, o_{n,1:k}, b_{i,n,k})$ indicate that local requirement $d \in D_i$ is satisfied by the current request, observations, and bindings. Then

$$\begin{aligned} G_{n,k}(c_i) \Longleftrightarrow \phi_i(q_n, o_{n,1:k}, b_{i,n,k}, S_i) \\ \wedge \bigwedge_{d \in D_i} \mathrm{Sat}(d; q_n, o_{n,1:k}, b_{i,n,k}). \end{aligned} \tag{6}$$

This gate captures only the requirements local to the Contract. Dependencies on other Contracts are introduced separately by the Executable Control Graph in section 3.2.

After the effect of c_i has been executed, completion is established only from episode-local evidence satisfying its completion criterion. We use a monotone completion latch within the same goal instance:

$$\begin{aligned} &\mathrm{complete}(c_i, k) = \mathrm{complete}(c_i, k-1) \vee \psi_i(q_n, o_{n,1:k}, b_{i,n,k}), \\ &\mathrm{complete}(c_i, 0) = \mathrm{false}. \end{aligned} \tag{7}$$

Thus, once completion has been established for the current goal instance, the Contract remains complete for the rest of that episode. The latch is runtime state and is not inherited by future episodes.

3.2 EXECUTABLE CONTROL GRAPH

A Contract may satisfy its local conditions yet remain ineligible for execution because another Contract must first establish a required subgoal. Long-horizon tool use therefore requires more than retrieving a set of individually relevant rules: their execution must be coordinated according to dependencies and information flow. CONTRACTFLOW represents this structure as a persistent Executable Control Graph

$$\mathcal{G}_n^K = (\mathcal{C}_n, \mathcal{E}_n, \mathcal{H}_n), \tag{8}$$

where $\mathcal{C}_n$ is the set of persistent Contracts, $\mathcal{E}_n$ is the set of directed control relations, and $\mathcal{H}_n$ stores execution evidence associated with Contracts, relations, and reusable workflows.

Relations in $\mathcal{E}_n$ carry operational instead of merely semantic meaning. Dependency relations specify which upstream goals must be completed before a downstream Contract may execute. Evidence-flow relations specify how observations or Contract outcomes can provide variables required downstream. Continuation relations encode conditional progress, completion-dependent transitions, and recovery paths. These explicit control relations are kept distinct from empirical path evidence in $\mathcal{H}_n$: observing two actions consecutively does not by itself make their ordering a mandatory dependency. The graph therefore represents not only which pieces of experience are related, but which execution states permit one piece of experience to follow another.

For a new request, retrieval selects a relevant subgraph from the persistent representation, but retrieval itself does not satisfy any execution requirement. Each Contract must still be grounded against the current episode. Extending the local gate in eq. (6), Contract c_i is enabled at runtime step k only if it

is active, all prerequisite Contracts are complete, its Contract-local gate holds, and it has not already completed:

$$\begin{aligned}\mathrm{Enabled}_{n,k}(c_i) \iff & \mathrm{Active}_{n,k}(c_i) \\ & \wedge \bigwedge_{c_j \in \mathrm{Pred}(c_i)} \mathrm{complete}(c_j, k) \\ & \wedge G_{n,k}(c_i) \wedge \neg\, \mathrm{complete}(c_i, k).\end{aligned} \tag{9}$$

Here, $\mathrm{Active}_{n,k}(c_i)$ indicates activation in the current control branch; ϕ_i tests Contract-local semantic applicability. $\mathrm{Pred}(c_i)$ denotes the Contracts connected to c_i by strict prerequisite relations. This definition separates local validity from compositional executability: $G_{n,k}$ evaluates the requirements internal to c_i, whereas $\mathrm{Pred}(c_i)$ captures dependencies induced by the graph.

A disabled Contract is not treated as an opaque failure. Its unresolved requirements provide explicit targets for subsequent execution. Missing evidence induces an observation step; an unfinished dependency redirects execution to an upstream Contract; and a condition known to be false activates an alternative eligible continuation when one is available. The control graph therefore does more than suppress unsupported actions: it exposes the reason an action is not yet executable and identifies the state transition required to make progress.

Once a Contract becomes enabled, its effect template e_i is instantiated from persistent knowledge S_i and current bindings $b_{i,n,k}$ and may then be dispatched to the corresponding tool. Execution does not imply completion. The Contract is marked complete only when subsequent evidence satisfies ψ_i according to eq. (7). The resulting completion latch prevents repeated effects within the same goal instance while remaining strictly episode-local. If execution fails, CONTRACTFLOW preserves the valid executed prefix and revises only the affected pending continuation. The resulting failure and recovery trace is then passed to the evolution mechanism as structured execution evidence.

3.3 EXECUTION-GROUNDED GRAPH EVOLUTION

Explicit execution structure makes feedback attributable. Instead of treating an episode as a single undifferentiated success or failure, CONTRACTFLOW associates observations and outcomes with the Contracts, relations, paths, and recoveries that actually participated in the trajectory. We denote the resulting structured evidence by

$$\xi_n = \mathrm{Certify}(\tau_n), \qquad \mathcal{G}^K_{n+1} = \mathcal{U}(\mathcal{G}^K_n, \xi_n), \tag{10}$$

where $\mathrm{Certify}$ converts the execution trace into structured evidence while preserving the provenance of observed tool outcomes, and $\mathcal{U}$ applies the corresponding updates to the persistent graph. Here, $\mathcal{U}$ denotes the Evolve operator in eq. (3).

Evolution operates at three levels. First, execution records update the usage and outcome evidence associated with the Contracts and relations that participated in the trajectory. Successful execution adds support to the control structure that produced it, whereas failures identify conditions, dependencies, or continuations whose support is insufficient or whose structure may require revision. Importantly, episode-level outcomes are not propagated indiscriminately to every retrieved graph element; updates are restricted to components that actually participated in execution.

Second, reusable successful paths may be consolidated into workflows. A workflow preserves recurring dependency and action structure across successful executions while leaving episode-specific arguments open for future grounding. Consolidation therefore enables later episodes to reuse recurring multi-step structure without turning a single historical trajectory into a fixed sequence of instantiated actions.

Third, a successful local recovery may be abstracted into a repair Contract or recovery relation conditioned on the state that triggered it. CONTRACTFLOW retains the reusable recovery structure and its supporting execution evidence, while discarding temporary identifiers, observations, bindings, and completion states from the episode in which the recovery occurred. Conversely, repeatedly unsupported knowledge may be revised or retired without rewriting unrelated regions of the graph.

The evolved persistent graph changes what can be instantiated in later episodes. Updated execution evidence informs reuse, while consolidated workflows and recovery structures provide new continuations that can be grounded under the state of a future task. Execution therefore serves two roles

in CONTRACTFLOW: within an episode, it advances the runtime state of the control graph; across episodes, it provides the evidence used to revise the persistent graph. The representation that governs action is thus also the representation that accumulates evidence for future improvement.

4 EXPERIMENTS

We evaluate whether organizing experience into executable, updateable structures improves tool-use agents. Our analysis follows three connected questions: does CONTRACTFLOW improve task outcomes, does its advantage persist as tasks progress, and which aspects of experience organization contribute to the observed gains?

4.1 EXPERIMENTAL SETUP

We evaluate on APOLLO (1,760 cases), CROSSEP-TOOL (200 episodes), and the MemoryArena group-travel setting (240 groups; 1,662 target people), using Qwen3.5-9B as the main experimental model. The three benchmarks probe complementary aspects of experience reuse. APOLLO measures exact accuracy and domain-level performance for multi-step tool behavior. CROSSEP-TOOL measures task success and partial progress as experience is reused across episodes. MemoryArena evaluates incremental planning for multiple travelers whose requests may refer to earlier travelers' plans. Its metrics separate requested modifications (Changed), whole-person correctness (Person), preservation of unmentioned slots (Preserve), and overall slot accuracy (All-slot). A common subset of 144 groups and 994 people supports the participant-position analysis. Comparisons cover no-memory agents, retrieval and history baselines, and learned experience memories. The appendix summarizes execution settings, structured interfaces, evaluation boundaries, and environment-wise CROSSEP-TOOL results; the main results table reports token counts for the CROSSEP-TOOL methods. The main comparisons evaluate the configured agent systems, including their experience representations and execution policies. GPT-5-mini provides an additional model reference.

4.2 FROM EXPERIENCE TO EXECUTABLE BEHAVIOR

Table 1: Main results across two tool-use settings. APOLLO scores are percentages. CROSSEP-TOOL reports success (%), progress, and recorded tokens per episode. Higher is better except for tokens.

APOLLO (1,760 cases)						**CROSSEP-TOOL (200 episodes)**			
Method	*Exact* ↑	*Comm.* ↑	*Travel* ↑	*Food* ↑	*Ent.* ↑	*Method*	*Success* ↑	*Progress* ↑	*Tokens* ↓
A0 FUNCTION CALLING	24.66	26.25	32.32	15.83	10.83	NO MEMORY	61.00	0.7942	76,366
REMINDER	27.05	34.31	26.61	20.00	13.33	RAW HISTORY	65.00	0.8137	324,030
AWM	27.27	24.17	16.96	50.00	37.92	BM25	59.50	0.7821	182,963
RAG	28.30	36.67	27.50	25.00	8.33	FLAT SUMMARY	61.50	0.7813	86,397
BASE	28.64	35.56	27.50	25.42	13.75	AWM	59.00	0.7830	82,552
FLAT CONTRACT MEMORY	32.78	33.47	43.39	6.67	32.08	ACE	48.50	0.6546	155,845
QUERY-SELECTIVE FLAT	33.24	33.75	45.18	5.83	31.25	REASONINGBANK	69.00	0.8394	92,458
REASONINGBANK	39.32	42.36	43.04	22.50	38.33	GPT-5-MINI	54.00	0.7521	**34,913**
LIFECYCLE CONTRACT SET	50.00	34.72	71.79	50.00	45.00	FROZEN GRAPH	67.00	0.8268	77,102
CONTRACTFLOW *(Ours)*	**77.50**	**85.28**	**81.79**	**60.00**	**61.67**	CONTRACTFLOW *(Ours)*	**71.50**	**0.8434**	78,912

4.2.1 APOLLO: ORGANIZING STATE-DEPENDENT BEHAVIOR

Table 1 shows that CONTRACTFLOW reaches 77.50% Exact accuracy on APOLLO, compared with 39.32% for REASONINGBANK, and achieves the highest score in each domain. Relative to the Lifecycle Contract Set, the margins are 50.56, 10.00, 10.00, and 16.67 percentage points for Commute, Travel, Food, and Entertainment, respectively. The uneven gains are consistent with the view that reusable experience must be organized around the current state and the dependencies among actions, instead of being retrieved as undifferentiated advice. This system-level comparison evaluates the combined representation and execution policy in the transition from retrieved experience to state-conditioned action, which is the central capability targeted by ContractFlow.

4.2.2 CROSSEP-TOOL: REUSING EXPERIENCE ACROSS EPISODES

On CROSSEP-TOOL, CONTRACTFLOW achieves 71.50% success and 0.8434 progress, compared with 69.00% and 0.8394 for REASONINGBANK and 61.00% and 0.7942 for NO MEMORY. The margin is smaller than on APOLLO, but the result shows that CONTRACTFLOW remains effective when experience must be reused across a sequence of episodes. Compared with raw-history reuse, this organization also avoids repeatedly passing the entire accumulated history to every decision; Table 1 reports token counts, and the appendix provides environment-wise results and execution examples. The result captures two aspects of experience reuse: the agent retains useful structure across episodes while avoiding the unbounded replay of raw history.

4.2.3 MEMORYARENA: EXECUTING CROSS-PERSON DEPENDENCIES

These results motivate a complementary question: can executable experience coordinate dependencies across successive requests while preserving valid earlier state? We study this question in MemoryArena, where later travelers may inherit or modify plans established by earlier travelers. This setting tests whether ContractFlow can carry forward useful structure while deciding which parts of the previous state remain applicable to the current request.

ContractFlow improves targeted changes while preserving unrelated state. MemoryArena evaluates a sequence of travelers whose requests can inherit or impose relative constraints on earlier plans. Relative to NO MEMORY, CONTRACTFLOW improves Changed accuracy from 10.83% to 16.51% and Preserve from 87.04% to 93.86%. Person accuracy also increases from 2.71% to 4.93%, while All-slot accuracy increases from 75.13% to 81.77%. These results show that ContractFlow improves cross-person dependency resolution while preserving unmentioned parts of the current plan. The joint improvement indicates that ContractFlow can adapt a shared plan to new traveler-specific constraints without treating every earlier decision as either immutable or disposable.

Table 2: MemoryArena results (240 groups; 1,662 target people).

Method	*Changed* ↑	*Person* ↑	*Preserve* ↑	*All-slot* ↑
NO MEMORY	10.83	2.71	87.04	75.13
LONG CONTEXT	11.81	1.87	91.59	79.13
BM25	11.56	2.65	92.41	79.78
FLAT SUMMARY	11.42	2.53	93.38	80.57
AWM	11.81	2.35	91.02	78.64
ACE	8.48	2.65	86.01	73.90
REASONINGBANK	12.30	1.99	88.84	76.88
CONTRACT SET	2.56	0.48	78.75	66.84
CONTRACTFLOW *(Ours)*	**16.51**	**4.93**	**93.86**	**81.77**

4.3 SEQUENTIAL EXPERIENCE REUSE

ContractFlow continues to benefit from experience as execution proceeds. Figure 3 breaks down Changed accuracy as the group-travel task progresses. CONTRACTFLOW achieves 17.41%, 18.38%, 17.03%, and 17.77% across the four position bins, exceeding all six listed baselines in each bin. At positions ≥ 5, it exceeds the strongest baseline in that bin, FLAT SUMMARY, by 5.72 percentage points. Its first-to-last-bin change is +0.36 points, while the corresponding changes are -5.07 for NO MEMORY and -5.93 for REASONINGBANK.

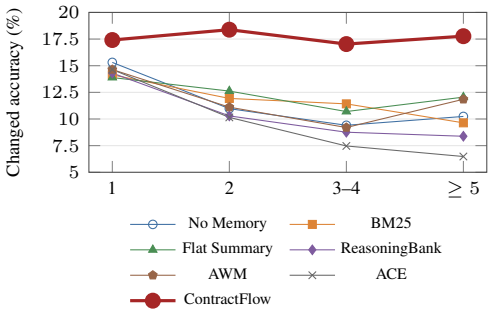

Figure 3: Changed accuracy by participant position on the common 144-group, 994-person subset (categorical bins; no error bars available).

This sustained performance indicates that CONTRACTFLOW can continue to exploit experience accumulated earlier in the group instead of relying only on the initial context. Together with the cross-episode result on CROSSEP-TOOL, the position analysis shows that execution structure remains useful for later decisions as within-group state accumulates. The position bins use a common 144-group subset, while the aggregate MemoryArena results above use all 240 groups. Within each group, later travelers are not independent requests: they inherit information from earlier plans or impose new relations on them. ContractFlow must therefore carry forward valid shared state while deciding which parts of that state remain applicable to the new request. The stable position-wise

performance is consistent with this ability to preserve useful execution structure while adapting it to later constraints. Together with the cross-episode update evidence from CROSSEP-TOOL, this supports the view that experience remains useful as execution proceeds. The agent is not merely replaying earlier plans: it carries forward reusable structure while rechecking its applicability for each new request.

4.4 WHY THE EXECUTABLE STRUCTURE WORKS

Execution structure and feedback. Table 3 connects the main execution and update components to the success metric reported in the main results. The full method achieves 71.50% success, compared with 61.00% for NO MEMORY and 65.00% for RAW HISTORY.

Table 3: Execution-structure comparison on CROSSEP-TOOL. Success is reported as a percentage; Δ is the difference in percentage points relative to the full method.

Configuration	*Success* ↑	Δ
NO MEMORY	61.00	−10.50
RAW HISTORY	65.00	−6.50
w/o Path Relations	67.50	−4.00
w/o Turn-local Retrieval	66.50	−5.00
w/o Recovery Contracts	65.50	−6.00
FROZEN GRAPH	67.00	−4.50
CONTRACTFLOW *(Ours)*	**71.50**	—

Removing recovery contracts yields the largest observed reduction among the execution-structure comparisons, decreasing success by 6.00 percentage points. Disabling turn-local retrieval reduces success by 5.00 points, removing the online update reduces it by 4.50 points, and removing path relations reduces it by 4.00 points. The pattern is consistent with the ContractFlow execution loop: relevant experience must be selected for the current turn, related actions must be coordinated, local failures must be recoverable, and feedback must remain available to later episodes. The comparison also clarifies how these stages work together. Retrieval identifies experience relevant to the current turn, relations determine how related actions should be coordinated, recovery preserves progress after local failures, and online updating makes the resulting execution evidence available to later episodes. The full method therefore benefits from the interaction of selection, dependency-aware execution, recovery, and feedback, consistent with the component comparisons.

5 CONCLUSION

We introduced CONTRACTFLOW, which turns reusable agent experience into executable Contracts organized by an Executable Control Graph. Contracts specify applicability, grounding, action, and completion; the graph coordinates their execution while keeping observations, bindings, and completion state local to each task. Attributed successes, failures, and local recoveries inform updates to the participating graph components without rewriting unrelated memory.

Across APOLLO, CROSSEP-TOOL, and MemoryArena, CONTRACTFLOW improves outcomes over no-memory, retrieval, history, and learned experience-memory baselines. Gains are largest on APOLLO; MemoryArena performance remains stable at later participant positions, and CROSSEP-TOOL component comparisons are consistent with the role of execution structure. These results show how stored experience can become grounded, executable, and revisable through interaction.

Limitations. The current evaluation leaves several aspects of ContractFlow untested. The three benchmark settings do not cover the full range of observation spaces, tool APIs, failure modes, or task horizons that may arise in deployment. Moreover, the component studies do not fully disentangle the effects of the representation, prompting strategy, and controller, so the contribution of each factor cannot be determined independently from the present experiments. Evidence for sequential reuse comes from within-sequence experience reuse and position-wise performance; the experiments do not cover a long-horizon deployment and do not yet quantify memory growth, update stability, or degradation from accumulated errors. Finally, execution-grounded updates rely on the quality of tool outcomes and completion checks, so incorrect or ambiguous verification can propagate unsupported Contracts. Broader environments, calibrated completion signals, and longer-running studies are needed to test robustness and memory evolution under these conditions.

AI USE STATEMENT

AI-assisted tools were used to support manuscript organization, language editing, LaTeX formatting, and the preparation of table and visualization code. The authors reviewed and revised all suggested text and code, verified the reported experimental measurements, and remain fully responsible for the paper's claims and contents. No AI-generated output was treated as empirical evidence without author verification.

ETHICS STATEMENT

This work studies memory and execution mechanisms for tool-using agents using existing evaluation settings and does not introduce a human-subject study or a real-world deployment. Persistent agent memory may retain sensitive, stale, or incorrect information, and tool execution may amplify the consequences of such errors. CONTRACTFLOW is designed to reduce these risks through current-state grounding, explicit execution conditions, and attributable updates, but these mechanisms do not replace application-specific access control, privacy protection, human oversight, or safety evaluation.

REPRODUCIBILITY STATEMENT

The Contract representation, runtime eligibility conditions, graph execution, and execution-grounded update interface are defined in section 3. The benchmark sizes, metrics, baselines, principal results, and component ablations are reported in section 4. The appendix describes structured interfaces, runtime instructions, benchmark execution settings, and evaluation boundaries, with worked examples of grounding, dependency resolution, and group-level recovery.

REFERENCES

Lei Chen, Guilin Zhang, Kai Zhao, Dalmo Cirne, Andy Olsen, Xu Chu, Zeke Miller, Alet Blanken, Amine Anoun, and Jerry Ting. Deployment-time memorization in foundation-model agents, 2026. URL https://arxiv.org/abs/2606.10062. The Second Workshop on the Impact of Memorization on Trustworthy Foundation Models at ICML 2026.

Yingjie Gu, Bo Xiong, Yijuan Guo, Chao Li, Xiaojing Zhang, Liqiang Wang, Pengcheng Ren, Qi Sun, Jingyao Ma, and Shidang Shi. FSFM: A biologically-inspired framework for selective forgetting of agent memory, 2026. URL https://arxiv.org/abs/2604.20300.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *The Fourteenth International Conference on Learning Representations*. OpenReview.net, 2026. URL https://openreview.net/forum?id=DT7JyQC3MR.

Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Enforcing temporal constraints for LLM agents, 2025. URL https://arxiv.org/abs/2512.23738.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory, 2025. URL https://arxiv.org/abs/2509.25140.

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, pages 8634–8652, 2023. URL https://arxiv.org/abs/2303.11366.

Juntong Wang, Haoyue Zhao, Guanghui Pan, Xiyuan Wang, Yanbo Wang, Qiyan Deng, and Muhan Zhang. SAGE: A self-evolving agentic graph-memory engine for structure-aware associative memory, 2026. URL https://arxiv.org/abs/2605.12061.

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*, pages 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/.

Zhaofen Wu, Hanrong Zhang, Fulin Lin, Wujiang Xu, Xinran Xu, Yankai Chen, Henry Peng Zou, Shaowen Chen, Weizhi Zhang, Xue Liu, Philip S. Yu, and Hongwei Wang. GAM: Hierarchical graph-based agentic memory for LLM agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 34647–34664. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1600. URL https://aclanthology.org/2026.acl-long.1600/.

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, pages 20004–20031, 2025. doi: 10.52202/085713-0593. URL https://pro.nips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. Memory-R1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12805–12825. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.583. URL https://aclanthology.org/2026.acl-long.583/.

Chao Yao, Yangbo Wei, Zhen Huang, Junhong Qian, Chenle Chen, Shaoqiang Lu, Chen Wu, and Lei He. Forgetting without restarting: Execution-state unlearning for stateful LLM agents, 2026. URL https://arxiv.org/abs/2609.04875.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Yujie Zhang, Weikang Yuan, and Zhuoren Jiang. Bridging intuitive associations and deliberate recall: Empowering LLM personal assistant with graph-structured long-term memory. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17533–17547. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.901. URL https://aclanthology.org/2025.findings-acl.901/.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners, 2023. URL https://arxiv.org/abs/2308.10144.

A PURPOSE AND SCOPE

This appendix expands the execution semantics and case-level evidence behind CONTRACTFLOW. The main paper presents the central representation and aggregate results. The appendix records how experience is compiled, grounded, executed, repaired, and reused, and presents case-level evidence, protocol details, and environment-wise results.

The appendix distinguishes three kinds of evidence. A *verified case* is backed by a frozen result or verification artifact. An *implementation-grounded example* follows the actual adapter and runner semantics. An *illustrative example* explains the intended behavior and is explicitly marked as such.

B CONTRACT COMPILATION AND RUNTIME EXECUTION

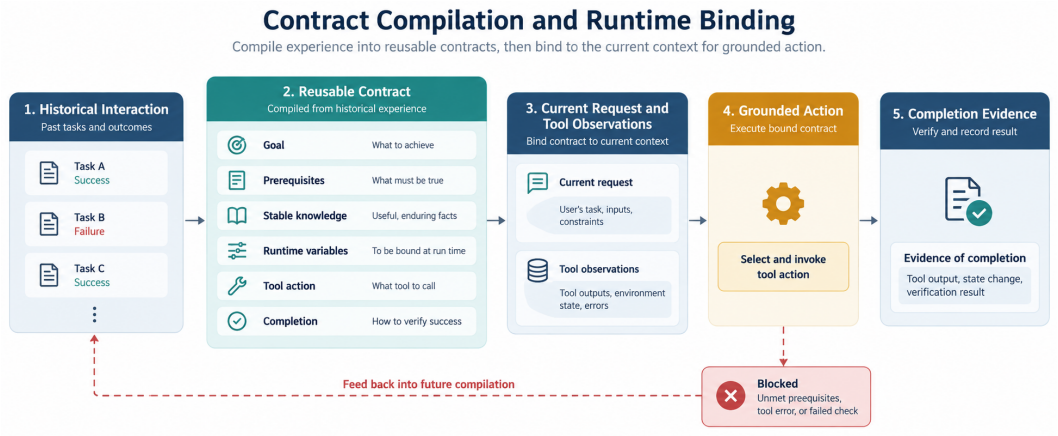

Figure 4: Contract compilation and runtime binding. Historical interactions are compiled into reusable Contract fields, which are grounded against the current request and tool observations before action dispatch.

B.1 PERSISTENT AND EPISODE-LOCAL INFORMATION

A historical interaction contains reusable structure and accidental state. Reusable structure may include a goal, preference, procedure, dependency, or recovery pattern. Accidental state includes identifiers, observations, runtime bindings, and completion states that belong only to the episode in which the interaction was observed. CONTRACTFLOW compiles the former into persistent Contracts and reconstructs the latter for each new episode.

A Contract is represented as

$$c_i = \langle g_i, D_i, S_i, V_i, \phi_i, e_i, \psi_i \rangle,$$

where g_i is a local goal, D_i contains evidence and prerequisite requirements, S_i stores stable knowledge, V_i contains volatile variables, ϕ_i describes applicability, e_i is a parameterized tool effect, and ψ_i specifies completion evidence. This representation makes execution constraints explicit without freezing the incidental state of a previous task.

B.2 CURRENT-STATE BINDING

A volatile variable is bound only when the current request or a current tool observation supports the binding. A prior plan or historical binding is not copied into a new action by default; it must first be judged against the current request, observations, and relevant dependency. Thus, the same reusable rule can be applied across episodes while stale or unsupported values are prevented from being carried forward.

B.3 GRAPH-LEVEL EXECUTION

Retrieval selects a relevant subgraph, but retrieval alone does not satisfy an execution requirement. A Contract is enabled only when its control branch is active, its local conditions hold, its required

variables are grounded, all strict prerequisite Contracts are complete, and the Contract has not already completed for the current goal instance. Missing evidence becomes an observation target; an unfinished prerequisite redirects execution upstream; and a failed continuation can activate a recovery relation when one is available.

Completion is established only from episode-local evidence satisfying the Contract's completion predicate. The completion latch prevents repeated execution for the same goal instance while remaining local to the episode. The persistent graph stores reusable evidence and relations, not the temporary bindings that happened to produce one successful action.

C Runtime Algorithm

The following execution loop makes the relation between the formal definitions and the benchmark adapters explicit. Persistent knowledge is instantiated for the current request, while bindings and completion state remain episode-local.

```
Input: request q, persistent graph G^K, tool set T
1. Retrieve a request-relevant subgraph from G^K.
2. Instantiate episode state: observations, bindings, and completion
    latches.
3. For each active Contract, bind variables supported by current
    evidence.
4. Resolve strict prerequisite edges before downstream actions.
5. If evidence is missing, gather the required observation and revisit
     binding.
6. Check local conditions and dispatch the action only if enabled.
7. Verify the tool result against the completion criterion.
8. On failure, preserve the valid prefix and activate a recovery
    continuation.
9. Record attributed execution evidence for participating Contracts
    and edges.
10. Update persistent reusable structure after the episode.
Output: task result y and updated graph G^K_{next}
```

The loop separates retrieval from execution. Retrieval proposes a relevant subgraph, but only current evidence, local conditions, and completed prerequisites can make an action eligible. The final update operates on attributed execution evidence instead of rewriting every retrieved memory.

D Protocol and Metric Summary

Table 4 summarizes the benchmark protocols used in the main paper. The table records the task boundaries and metrics needed to interpret the reported results; representative prompt interfaces and run checks are given below.

For MemoryArena, Changed measures requested modifications, Preserve measures unmentioned slots that remain correct, Person requires whole-person correctness, and All-slot aggregates slot-level correctness. These metrics capture different failure modes and are reported separately.

D.1 APOLLO development and evaluation sets

The released APOLLO task file contains 2,080 cases across four domains: Commute, Entertainment, Food, and Travel. For method development and component diagnostics, we define a fixed 320-case development subset by stratified sampling over domain, history type, and query type:

$$4 \text{ domains} \times 2 \text{ history types} \times 2 \text{ query types} \times 20 \text{ cases per stratum} = 320.$$

The two history types are explicit and implicit, and the two query types are reactive and proactive. Sampling uses random seed 17. The seed controls only case selection; these cases are not used for

Table 4: Benchmark protocol and metric summary.

Benchmark	Task structure	Main scale	Primary metrics	State boundary
APOLLO	Multi-domain, multi-step tool calling	1,760-case evaluation stream	Exact and domain accuracy	Fresh case instance; current observations ground actions
CROSSEP-TOOL	Ordered cross-episode tool use	200 episodes	Success and Progress	Episode order is preserved; evolving variants update later reuse
MemoryArena	Sequential multi-traveler planning	240 groups / 1,662 people	Changed, Person, Preserve, All-slot	Group order is preserved; labels are analyzer-only

model-weight fine-tuning or cross-case memory initialization. They are used for method development, debugging, and component diagnostics.

The 1,760 evaluation cases are the disjoint complement of the 320 development cases within the official 2,080-case task collection. The 320-case subset is used for development and component diagnostics, while the 1,760-case stream is used for the reported large-scale evaluation. All APOLLO cases are executed independently with fresh runtime state.

E Structured Interfaces

Contracts are compiled from prior interaction records using the persistent schema defined in the main paper. The runtime grounds them in the current request and tool observations, while the adapter maps the environment's tool and observation interface to the shared ContractFlow representation. The following schema and runtime instructions specify the interface between Contract compilation and execution in the MemoryArena adapter.

E.1 Contract compiler interface

The MemoryArena adapter uses a structured compiler interface:

```
{
  "person": "name",
  "preserve_unmentioned": true,
  "reference_person": "base person name",
  "contracts": [
    {
      "id": "unique short id",
      "day": 1,
      "slot": "accommodation",
      "operation": "inherit_person|computed_relation|replace",
      "constraints": {"...": "..."},
      "depends_on": [{"person": "name", "day": 1,
                      "slot": "slot"}],
      "tool": "AccommodationSearch|null",
      "success_test": "how to verify the chosen value"
    }
  ]
}
```

The explicit operation type separates inheritance from a computed relation. These fields instantiate the benchmark's task schema; the ContractFlow runtime uses the same binding, dependency, completion, and update interfaces across environments.

E.2 Graph runtime interface

The graph runtime follows the following protocol:

```
1. Start from the reference plan and preserve every unmentioned slot.
2. Resolve dependency edges first.
3. Call tools only for unresolved volatile Contracts.
4. For computed relations, read the referenced value first.
5. Do not modify a locked slot or repeat a satisfied lookup.
6. Check dependencies, evidence, and constraints before answering.
```

The excerpt maps the prompt interface to the formal Contract and Graph definitions.

E.3 Execution-interface responsibilities

The formal operators in the main paper define interfaces between persistent memory and runtime execution. $\mathrm{Retrieve}$ selects a request-relevant subgraph; $\mathrm{Supports}$ checks current evidence and bindings; Sat evaluates Contract requirements; and Active checks activation in the current control branch. Their inputs are mapped from the environment interface, while the state and control semantics are shared.

After an episode, $\mathrm{Certify}$ converts the execution trace into outcome records with Contract and relation provenance. The update operator $\mathcal{U}$ applies those records to participating graph elements: successful support can be strengthened, an unsupported condition can be revised, a verified local recovery can add a repair relation, and structure that continues to fail support checks can be retired. These operations use the same state separation and attribution rules throughout the evaluation.

F Verified Case I: Evidence-Grounded Action Selection

F.1 Same-task paired outcome

The APOLLO evaluation artifact contains a task for which the recorded baseline accuracy is 0 while the ContractFlow branch receives accuracy 1. The ContractFlow trace records a current search observation, a proposed premature finish, an evidence-complete action trigger, a grounded booking action, and a successful tool observation.

The execution pattern is:

1. Search the current environment and obtain candidate entities.
2. Check whether the current Contract's required evidence has been consumed by the proposed action.
3. Reject the premature finish when the action arguments remain ungrounded or unsupported.
4. Bind the action argument from the current observation and dispatch the tool call.
5. Verify the resulting success observation before latching completion.

The artifact records `trigger_reason=evidence_complete_action` and a successful booking observation. The baseline and ContractFlow rows share the same input task. The trace illustrates how execution checks affect action dispatch through current-state grounding.

Provenance. The case is extracted from the stored APOLLO evaluation results. The baseline and ContractFlow outcomes are taken from the same input task and the corresponding execution trace.

Table 5: Anonymous verification record for the paired APOLLO case.

Recorded field	Verified value
Shared input task	yes
Baseline exact accuracy	0
ContractFlow exact accuracy	1
Current search observation	present
Premature-finish rejection	present
Grounded booking action	present
Successful tool observation	present
Completion after success evidence	present

G Implementation-Grounded Case II: Cross-Person Dependencies

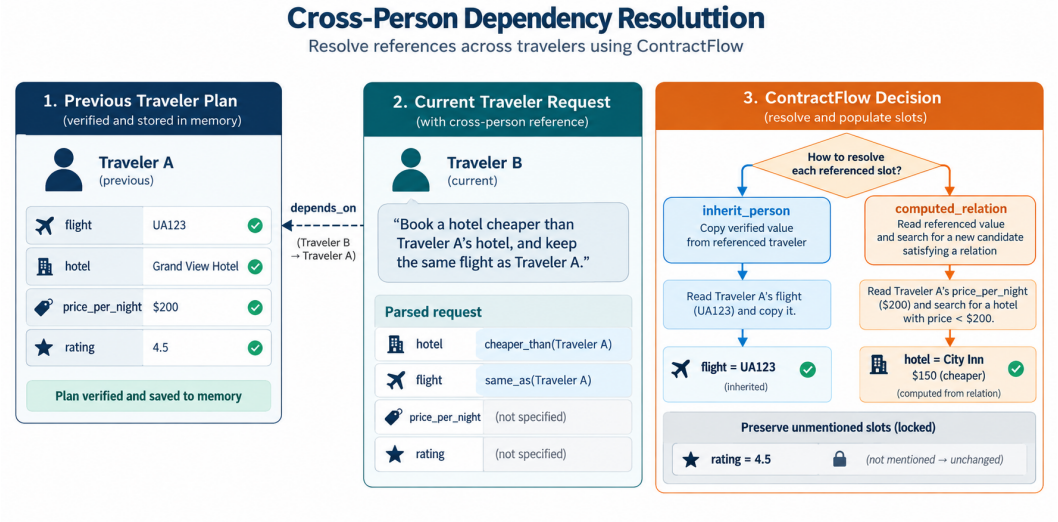

Figure 5: Cross-person dependency resolution. Inheritance and computed relations use different execution paths while unmentioned slots remain locked.

G.1 Inheritance and Relative Constraints

MemoryArena processes a base traveler followed by later travelers whose requests may refer to earlier plans. The runner defines two distinct operations. For an inheritance request, it emits an `inherit_person` Contract with a `depends_on` address. For a relative request, it emits a `computed_relation` Contract that reads the referenced value before searching for a candidate satisfying the new relation.

For example, the following requests invoke the two operations:

1. "Use the same hotel as the previous traveler."
2. "Choose a cheaper hotel than the previous traveler."

The first request can copy a verified referenced slot after dependency resolution. The second must inspect the referenced slot and evaluate a new candidate. In both cases, unmentioned slots remain locked to the reference plan.

The operation types, dependency schema, and runtime rules in this example follow the benchmark adapter implementation.

G.2 Paired Held-Out Group

On held-out group 151, the same input group and traveler order were evaluated with the Long Context and Contract Graph variants. The evaluator identifies 18 *requested modification slots* for this group.

These are slot-level obligations induced by the travelers' requests; they are not 18 travelers and do not include unmentioned slots that are scored only for preservation. Of these 18 requested modification slots, Long Context satisfies 5 and Contract Graph satisfies 16.

Table 6: Anonymous slot-level verification record for MemoryArena group 151.

Variant	Requested modification slots	Satisfied
Long Context	18	5
Contract Graph	18	16

This paired comparison is used only as a case-level illustration of traveler-specific dependency resolution inside a shared planning context; the aggregate MemoryArena results are reported separately in the main paper.

Provenance. The counts are obtained from the slot-level analyzer applied to the same held-out group under both variants.

H Verified Case III: Group-Level Recovery and State Preservation

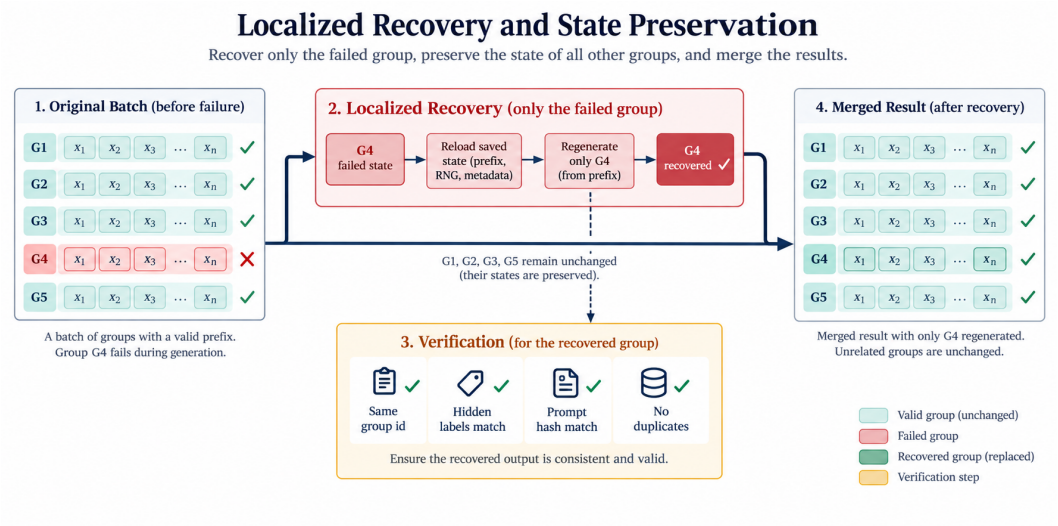

Figure 6: Group-level result recovery and state preservation. Only the failed group is regenerated and verified before it is merged with unchanged groups.

H.1 Group-29 repair

The MemoryArena development run contains an original Contract Graph row for group 29 with a fatal error and a separate recovery row. The recovery verifier checks that exactly one recovery row exists, that it is group 29, and that the recovery row is valid. It also checks that the repaired row equals the recovery row and that all other original group rows remain unchanged.

The verification record additionally checks exact expected group IDs, hidden labels, no duplicate groups, no fatal errors after repair, and matching prompt hash bundles across variants. The repair therefore has a clear group-level scope:

1. retain the results of unaffected groups;
2. regenerate only the failed group;
3. verify identity, format, and label isolation;
4. merge the repaired row only after all checks pass.

Table 7: Recovery-verifier checks for MemoryArena group 29.

Verification condition	Result
Exactly one recovery row is present	Pass
Recovery row corresponds to group 29	Pass
Recovery row passes validity checks	Pass
Merged group-29 row equals the verified recovery row	Pass
All non-29 group rows are unchanged	Pass
Expected group IDs are complete and non-duplicated	Pass
No fatal error remains after repair	Pass
Prompt-hash bundles match the compared variants	Pass
Hidden labels remain analyzer-only	Pass

This development case documents replacement of a failed group result while preserving the other group results, as checked by the recovery verifier.

Provenance. The case is supported by the MemoryArena recovery verifier used in the reported run.

I CONTINUAL EXPERIENCE REUSE

CROSSEP-TOOL and MemoryArena expose two distinct forms of sequential reuse. MemoryArena tests whether the active planning representation remains useful as later travelers inherit or transform earlier state within the same group. Its participant-position analysis therefore measures robustness to accumulated within-group state and dependency depth; by itself, it does not identify the causal effect of persistent cross-episode memory updates. CROSSEP-TOOL provides the corresponding update intervention: after a shared bootstrap graph is constructed, the frozen branch disables episode-boundary updates while the evolving branch continues to update the same initial graph. We therefore use MemoryArena to analyze sequential state reuse and the CROSSEP frozen/evolving fork to analyze the contribution of post-bootstrap online evolution.

J VERIFIED CASE IV: CROSS-EPISODE ONLINE UPDATE

The Qwen3.5 CROSSEP telemetry distinguishes reuse events from persistent episode-boundary updates. Each environment first executes five bootstrap episodes (indices 0–4) in the official order. The bootstrap updater constructs the initial graph from those five agent-visible trajectories, after which the same graph snapshot is copied into a read-only branch and an evolving branch.

For the `gorilla_fs` evolving stream, the stored telemetry records 249 `utilize` events and a final `update_count` of 50. The graph has `update_count=5` at the branch point after bootstrap and advances to 50 because the evolving branch performs one episode-boundary update after each of episodes 5–49. The corresponding frozen branch loads the same bootstrap graph with `readonly=True`; its final `update_count` remains 5, confirming that no post-bootstrap persistent update is applied. A representative evolving update touches three skill entries together with a localized set of graph nodes and edges, including command nodes such as `cd`, `mkdir`, and `mv` and observed transitions such as `cd->mkdir` and `mkdir->mv`.

The telemetry is evidence that the evolving condition both reuses existing structure and performs localized persistent updates. Performance attribution is made at the run level through the paired frozen/evolving comparison described below; the event counts themselves are not treated as evidence of improved task accuracy.

Provenance. The event counts and update counters are taken from the stored telemetry and state summaries of the reported CROSSEP run.

K CROSSEP-TOOL ENVIRONMENT BREAKDOWN

The following table gives the environment-wise Success breakdown for all methods on the 200-episode CROSSEP-TOOL evaluation. The four environments contain 50 episodes each.

Table 8: CROSSEP-TOOL Success by environment over 200 episodes.

Method	Gorilla	Vehicle	Trading	Travel	Overall
Episodes	50	50	50	50	200
NO MEMORY	64.00	50.00	90.00	40.00	61.00
BM25	52.00	56.00	90.00	40.00	59.50
ACE	58.00	38.00	90.00	8.00	48.50
AWM	64.00	48.00	72.00	52.00	59.00
FROZEN GRAPH	66.00	56.00	90.00	56.00	67.00
REASONINGBANK	66.00	68.00	88.00	54.00	69.00
CONTRACTFLOW	**72.00**	66.00	**92.00**	56.00	**71.50**

Relative to Frozen Graph, the evolving CONTRACTFLOW configuration improves Success by 6 percentage points on Gorilla (66% $\rightarrow$ 72%), 10 points on Vehicle (56% $\rightarrow$ 66%), and 2 points on Trading (90% $\rightarrow$ 92%), while matching Frozen Graph on Travel (56% in both conditions). The overall gain is 4.5 percentage points (67.0% $\rightarrow$ 71.5%). The effect of post-bootstrap evolution is therefore environment-dependent rather than uniform: it is largest on Vehicle, moderate on Gorilla, small on Trading, and not measurable on Travel in this run.

L BENCHMARK EXECUTION SETTINGS

This section summarizes the execution settings for the three benchmarks reported in the main paper.

L.1 APOLLO

The official task file contains 2,080 cases organized into Commute, Entertainment, Food, and Travel. The project-specific 320-case development subset is sampled with seed 17 using 20 cases from each domain $\times$ history-type $\times$ query-type stratum, as described above. These cases are used for development and component diagnostics only; they are not used for model-weight fine-tuning or cross-case memory initialization.

The main-paper APOLLO results use the 1,760-case evaluation complement defined above. The stored execution settings use Qwen3.5-9B, temperature 0, reasoning effort `none`, a maximum of 8 steps, and 512 completion tokens. The ContractFlow execution configuration uses the dependency-edge form for Commute and Entertainment and the lifecycle Contract Set form for Travel and Food. These domain-specific execution configurations were selected using only the 320 development cases and fixed before evaluation on the 1,760 held-out cases; they were not adjusted based on evaluation results.

L.2 CROSSEP-TOOL

We use a 200-episode ordered evaluation protocol across Gorilla FS, Vehicle Control, Trading Bot, and Travel API, with 50 episodes per environment. The stored Qwen3.5-9B formal configuration uses an OpenAI-compatible service, a 20-step limit, a 1,200-second request timeout, and one worker per environment stream.

Each environment uses episodes 0–4 as bootstrap episodes in the official order. The bootstrap updater constructs a graph from the five agent-visible trajectories. That exact graph snapshot is then copied into the two evaluation branches, and the stored state check confirms identical graph hashes at the branch point. For episodes 5–49, the frozen branch loads the graph with `readonly=True` and skips episode-boundary updates, whereas the evolving branch continues to update after each completed episode. Both branches retain the same model, episode order, tool interface, request budget, and bootstrap graph; the manipulated factor is whether persistent post-bootstrap updating is enabled.

L.3 MEMORYARENA GROUP-TRAVEL

The group-travel release contains 270 groups, with 6 development-smoke groups, 24 confirmation groups, and 240 held-out groups containing 1,662 people. The stored Qwen3.5-9B configuration uses

temperature 0, 12 runtime steps, 2,048 output tokens, and a 6,000-character tool-result bound. Group order and person order are preserved, and labels are read only by the analyzer. The main metrics are Changed, Person, Preserve, and All-slot.

M Implementation and Evaluation Protocol

M.1 Implementation scope

Environment adapters invoke the same Contract compilation, current-state binding, dependency resolution, completion, recovery, and update interfaces; they expose environment schemas, observations, and task validity checks. The main comparisons evaluate each method with its configured representation and execution policy. GPT-5-mini in Table 1 provides an additional model reference. The paired Frozen Graph comparison isolates post-bootstrap updating under the shared configuration described below.

M.2 Run configuration

The main experimental protocols use Qwen3.5-9B on the APOLLO 1,760-case evaluation stream, CROSSEP-TOOL (200 ordered episodes), and MemoryArena (240 held-out groups containing 1,662 target people).

For CROSSEP-TOOL, ContractFlow denotes the evolving branch after the common five-episode bootstrap. Frozen Graph denotes the read-only ablation used in both the main-results and component-comparison tables. This branch starts from the same post-bootstrap graph as the evolving condition and disables all persistent episode-boundary updates during episodes 5–49. The two branches use the same Qwen3.5-9B model, official episode order, tool interfaces, step/request budgets, and bootstrap graph. Equality of the copied bootstrap graph is checked by its stored graph hash. The resulting final update counters are 5 for the frozen branch and 50 for the evolving branch.

M.3 Evaluation boundaries

The benchmark manifests and task order define the evaluation inputs. APOLLO cases are executed independently with fresh runtime state; the 320-case development subset is used for development and component diagnostics, while the 1,760-case evaluation stream is used for the reported large-scale evaluation.

In MemoryArena, labels are reserved for the analyzer, while the runner operates on requests, reference plans, and tool observations. Group-level recovery uses the same input identity and generation settings as the original task; the replacement must satisfy identity, format, label-isolation, and unchanged-row checks before it is merged with the unaffected results.

In CROSSEP-TOOL, the frozen/evolving comparison is paired at the post-bootstrap branch point. The bootstrap graph is identical across branches; only post-bootstrap persistent updating differs. This boundary supports interpreting the frozen/evolving gap as evidence about online evolution rather than differences in initialization, episode order, tools, or model budget.