ONE TREE, THREE ROLES: GOAL-STRUCTURED EXTRACTION, ADMISSION, AND REUSE OF PROCEDURAL MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents can improve by reusing procedures from past executions. However, long-horizon trajectories interleave multiple goals, failed attempts, and later corrections, making it difficult to identify coherent procedures and determine which execution evidence supports them. Reusable procedures also vary in scope, from local tool-use steps to multi-step workflows. We introduce GPM (Goal-structured Procedural Memory), a training-free framework for procedural memory that reconstructs the goal hierarchy reflected in a completed task trajectory. Higher-level goals are accomplished through lower-level subgoals, which may themselves be further decomposed. We reconstruct these relations from the finished ReAct trajectory to form a Goal Tree, while linking each goal to the actions and observations for it. The same tree is used across the memory pipeline: a goal or subtree defines the scope of a procedure, its linked execution records provide the evidence used to decide whether that procedure should be stored, and procedures can later be retrieved at different levels of granularity. Because the tree is reconstructed only after execution, the agent needs no hierarchical planner while acting. Experiments on AppWorld show that GPM improves Task Goal Completion by 3.8 and 7.4 percentage points across two executor backbones. A controlled admission study further shows that goal-scoped verification reduces invalid admission by 7.6 percentage points while largely preserving valid memories.

1 INTRODUCTION

Recent advances in large language models have enabled LLM agents to handle increasingly complex tasks that require multi-step reasoning, tool use, and interaction with external environments. Although individual tasks may differ, they often involve similar solution procedures, making past executions a useful source of procedural memory. Reusing these procedures can reduce repeated exploration and reasoning in future tasks (Zhao et al., 2024; Wang et al., 2025; Ouyang et al., 2026; Cao et al., 2026). However, extracting them from long execution trajectories is difficult: a trajectory may contain multiple objectives, failed attempts, retries, and later corrections. More fundamentally, a trajectory records an entire task, while a reusable procedure may be extracted from a single part of the execution or from several related parts together. Treating the whole trajectory as one memory can mix unrelated behavior, while reducing it to isolated actions can fragment multi-step procedures. Constructing procedural memory therefore requires identifying reusable units at an appropriate granularity (Fang et al., 2026; Fan et al., 2025).

Another challenge is that abstracting procedures from trajectories may introduce hallucinated or inaccurate content, such as steps, conditions, or effects that were never observed in the execution. Such errors motivate verification and admission mechanisms before extracted knowledge is retained in persistent memory (Cao et al., 2026; Mi et al., 2026; Zhang et al., 2026). Verifying an extracted procedure requires execution evidence, raising a further question: what execution records should be used for verification? If the verifier evaluates the procedure using the entire trajectory as evidence, irrelevant execution records may be mistaken for support, causing unsupported procedures to be admitted. To reduce this risk, the verifier should focus on the execution records relevant to the procedure.

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.

Our key insight is that both problems arise because a flat trajectory hides the structure of execution. A complex task is carried out through objectives at different levels: Higher-level objectives can be expanded into lower-level objectives, and this decomposition can repeat across levels, forming a hierarchy from the overall task to local operations, each associated with the actions and observations involved in completing it. We reconstruct this structure from the completed trajectory as a Goal Tree. A procedure can be defined over a single goal or a group of related goals, while its associated records provide the evidence for verification, enabling variable-granularity extraction and goal-scoped admission.

Building on this insight, we introduce GPM, a training-free memory framework that reconstructs a Goal Tree from each completed ReAct trajectory (Yao et al., 2022) and uses it for procedure extraction, admission, and reuse. The reconstructed goal structure supports procedure extraction at different granularities. Each extracted procedure is verified using its associated execution records. Only supported procedures are admitted to memory. During future tasks, procedures at different levels of granularity can be retrieved and reused as needed. Because the Goal Tree is reconstructed after execution, GPM requires neither a hierarchical planner nor parameter updates during task execution.

We evaluated GPM in AppWorld (Trivedi et al., 2024) and find that it improves the execution of downstream tasks over baselines. We also conduct a controlled admission study to isolate the effect of goal-scoped verification. Restricting verification to the relevant execution scope reduces unsupported memory admissions while retaining most supported procedures. These results support both the effectiveness of GPM and the value of the Goal Tree for memory admission.

Our contributions are threefold. First, we reconstruct a Goal Tree from completed ReAct trajectories, capturing the hierarchical structure of task execution without requiring hierarchical planning during execution. Second, we use the same structure to define both the unit for procedure extraction and the execution scope for memory admission. Third, we use the Goal Tree to support procedural memory at multiple granularities for later retrieval and reuse, and validate the resulting framework on AppWorld together with a controlled study of goal-scoped admission.

2 Related Work

2.1 Experience reuse and procedural memory

Agent memory has evolved from replaying past experience toward extracting increasingly reusable knowledge. Synapse retrieves complete trajectories as in-context exemplars for computer control (Zheng et al., 2024), while ExpeL extracts natural-language insights from accumulated experience and recalls both insights and past experience at inference time (Zhao et al., 2024). Agent Workflow Memory (AWM) goes further by inducing reusable workflows from offline demonstrations or online interaction (Wang et al., 2025). Together, these works span a progression from trajectory replay to increasingly abstract forms of experience reuse.

Recent work develops this direction into more explicit procedural memory. ReasoningBank distills reusable reasoning strategies from successful and failed experiences (Ouyang et al., 2026). MemP extracts both step-level instructions and higher-level script-like abstractions from past trajectories (Fang et al., 2026). ReMe builds procedural memory through multi-faceted distillation, context-adaptive reuse, and utility-based refinement (Cao et al., 2026). Skill-Pro represents experience as executable skills with activation, execution, and termination conditions, and uses a verification gate before retention (Mi et al., 2026).

These methods study how experience can be transformed into reusable knowledge and maintained across tasks. GPM focuses on a complementary question: the source execution is not only material for abstraction, but also the evidence used to determine whether an extracted procedure is supported. We therefore study which part of an execution should be used to verify each extracted procedure.

2.2 Structured and multi-granularity memory

Recent work also explores structured representations of agent experience. H-EPM constructs a tool graph from accumulated trajectories, representing recurring tool dependencies as procedural routines while retaining episodic context (Li et al., 2026). MUSE maintains Strategic, Procedural, and

Tool Memories at different levels of abstraction, and distills successful subtask trajectories into Procedural Memory (Yang et al., 2026). MetaFlowLLM organizes historically completed tasks into a hierarchical experience tree of reusable MetaFlows, supporting experience generalization and retrieval across levels of abstraction (Fan et al., 2025). Hierarchical Memory Tree (HMT) constructs an Intent–Stage–Action hierarchy from raw trajectories, separating task-level logic from action-level details (Tan et al., 2026). SkillX similarly distills trajectories into strategic plans, functional skills, and atomic skills (Wang et al., 2026). MemP represents procedural knowledge at both step and script levels (Fang et al., 2026). MACLA maintains a hierarchical procedural memory of reusable procedures and combines Bayesian reliability estimates with contrastive refinement across successful and failed experience (Forouzandeh et al., 2025).

These works show the value of structure and multiple levels of abstraction for organizing, generalizing, and reusing agent experience. In GPM, the reconstructed goal structure serves an additional role: it provides a shared structural boundary for procedure abstraction and verification evidence. A procedure is extracted from a goal or goal subtree, and the execution records associated with that same region define the evidence used to verify it. The resulting structure also supports later reuse at different granularities.

2.3 Memory verification and admission

Persistent memory also requires mechanisms for controlling memory quality. ReasoningBank uses successful and failed experience when distilling new strategies (Ouyang et al., 2026). ReMe maintains its memory pool through validation and utility-based refinement (Cao et al., 2026). Skill-Pro uses a PPO Gate to verify candidate skills before they enter the skill pool (Mi et al., 2026). Recuris turns execution into structured evidence for failure localization and applies localized, validation-gated updates to Skill Memory (Yu et al., 2026). CONTRAMEM uses outcome differences across same-task multi-model trajectories to construct Function Cards and Skill Cards, which are maintained through localized curation (Deng et al., 2026). More generally, A-MAC formulates long-term memory admission as a structured decision based on properties such as future utility, factual confidence, novelty, and recency (Zhang et al., 2026).

GPM addresses a different aspect of memory admission: the execution-evidence scope used to establish support for a procedure extracted from a trajectory. Rather than deciding admission primarily from properties of the candidate memory itself, GPM associates each extracted procedure with its corresponding region of the Goal Tree and restricts verification to the execution records from that region. This prevents unrelated parts of the trajectory from being mistaken for evidence for the extracted procedure.

3 Method

3.1 Problem Setup and Framework Overview

We consider an LLM agent that solves a task x through ReAct-style interaction with an external environment. An execution produces a trajectory

$$\tau = \big(x, (r_1, a_1, o_1), \ldots, (r_T, a_T, o_T)\big)$$

where r_t denotes the agent's intermediate reasoning, a_t denotes the executed action, including its arguments or executable content, and o_t denotes the resulting observation. Although the trajectory contains the full interaction history, it remains a flat sequence and does not explicitly represent how individual steps contribute to objectives at different levels.

Our goal is to construct reusable procedural memory from completed trajectories without requiring hierarchical planning during execution. Given a completed trajectory τ, GPM first reconstructs a *Goal Tree* G_τ that organizes the execution around goals at different levels. Each goal is associated with the actions and observations involved in pursuing it, while parent–child relations represent goal decomposition, where a higher-level goal is carried out through its lower-level goals. The reconstructed structure is then used for three purposes: identifying procedure units for extraction, defining the execution evidence used for memory admission, and supporting later reuse at different granularities.

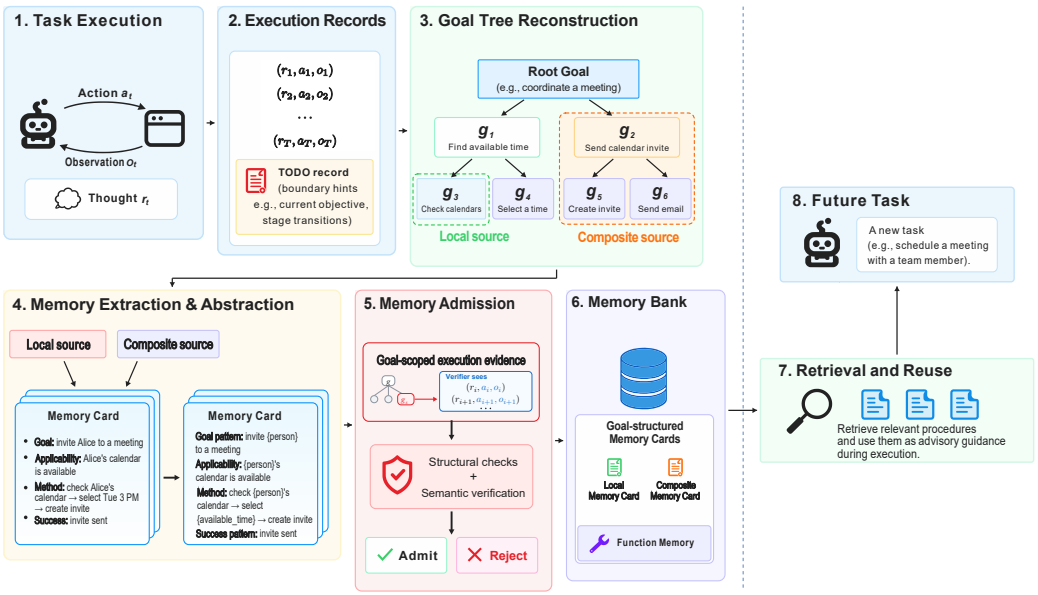


Figure 1: Overview of GPM. A completed ReAct trajectory is reconstructed into a Goal Tree, which supports procedure extraction, goal-scoped admission, and later retrieval and reuse.

Figure 1 gives an overview of the framework. After task execution, GPM reconstructs the Goal Tree from the completed trajectory and extracts candidate procedures from selected regions of the tree. Each candidate is checked against the execution evidence associated with its source goal before it is admitted to persistent memory. For a future task, the agent retrieves procedures that are relevant to its current execution need and uses them as advisory guidance. All memory construction is performed after execution, and GPM requires no parameter updates.

3.2 Post-hoc Goal Tree Reconstruction

Given a completed trajectory τ, GPM reconstructs a rooted Goal Tree

$$G_\tau = (V, E)$$

where each node $g \in V$ represents a goal pursued during the execution, and the root node represents the overall task. Leaf nodes represent goals with no child goals, while internal nodes represent goals with one or more child goals. Each edge $(g_i, g_j) \in E$ represents a parent–child relation between two goals, where g_j is a lower-level goal under g_i. The reconstruction uses the reasoning, actions, and observations in the completed trajectory to infer the hierarchical structure of the execution. At the top level, GPM groups the trajectory into coarse execution phases corresponding to higher-level objectives. TODO markers are used only as boundary hints at this stage. Within each phase, actions are assigned to the goals they directly serve. Composite goals are then recursively decomposed into lower-level goals, yielding a hierarchy from the overall task to local operations.

3.3 Goal-Structured Procedure Extraction

The Goal Tree defines procedure sources at different levels of granularity. A leaf node can provide a local source, while an internal node together with its descendants can provide a composite source. Because internal nodes occur at different levels of the tree, composite sources can cover different amounts of the execution.

The Selector selects sources that contain reusable experience and identifies the execution records that support each selected source. For a node g, let $A_{\text{local}}(g)$ denote the actions directly assigned to

g, and let

$$A_{\text{composite}}(g) = \bigcup_{v \in \text{subtree}(g)} A_{\text{local}}(v)$$

denote the actions in the subtree rooted at g. A local source is restricted to $A_{\text{local}}(g)$, whereas a composite source may use $A_{\text{composite}}(g)$. The corresponding candidate source regions are

$$\mathcal{S}(G_\tau) = \big\{(\{g\}, A_{\text{local}}(g)) \mid g \text{ is a leaf}\big\} \cup \big\{(\text{subtree}(g), A_{\text{composite}}(g)) \mid g \text{ is internal}\big\}.$$

The first component identifies the goal region from which a procedure is extracted, the second bounds the execution records that may support that procedure.

The Abstractor constructs a Memory Card from each selected source and its supporting execution records. Depending on the validated outcome and retry information, the Memory Card records positive experience, a correction, or a guard. Positive and correction Memory Cards contain a goal pattern, applicability conditions, a method, and a success condition. A correction additionally records the observed defect and its validated repair. A guard records what should be avoided without introducing an unobserved repair. The goal pattern expresses the source goal in a reusable, parameterized form. The applicability conditions, method, and success condition describe when the procedure applies, how the goal is carried out, and what state indicates completion. The original task and the selected source goal define the operation and constraints that belong to this source. Source-owned execution records ground claims about performed steps, observed outcomes, and failures. The resulting Memory Card must preserve the source goal without extending its claims beyond the available evidence.

The source granularity determines the structure of positive and correction Memory Cards. A local source produces a *Local Memory Card*, a single-node memory corresponding to the source goal. A composite source produces a *Composite Memory Card*, a multi-node memory that preserves the parent–child decomposition of the Goal Tree. The root represents the overall procedure, and its child nodes record the procedural details of the corresponding lower-level goals.

Function-specific repairs are handled separately through Function Memory. A validated failure–repair pair for the same function is stored as a call-level repair pattern for later reuse.

3.4 Goal-Scoped Memory Admission

The Abstractor may produce claims that are not fully supported by the recorded execution. GPM therefore treats each extracted Memory Card as a candidate until it passes an admission check. The verification scope of each candidate is determined by its source in the Goal Tree. For a selected source s, let $A = A_{\text{local}}(g)$ when s is local and $A = A_{\text{composite}}(g)$ when s is composite. Its source-scoped execution records are

$$\tau_s = \{(r_t, a_t, o_t) \in \tau \mid a_t \in A\}.$$

The Verifier checks the candidate against the full scope τ_s. Records in τ_s outside the selected support can expose omissions or contradictions, but cannot justify new claims.

Admission first checks that every cited execution record can be resolved and belongs to τ_s. It also checks structural constraints such as action ownership, valid failure–repair ordering, and branch coverage for composite sources. Candidates that fail these checks are rejected before semantic verification by the LLM Verifier.

The remaining candidates are evaluated against their source-scoped execution evidence. For positive Memory Cards, the Verifier checks whether the recorded operations support the method and whether the observed results support the success condition. For correction Memory Cards, it checks that the stated defect and repair match a validated failure–repair sequence. For guard Memory Cards, it checks that the recorded execution supports the stated failure condition or behavior to avoid. The Verifier also checks that the abstraction remains consistent with the source goal and does not introduce unsupported requirements.

Composite Memory Cards receive additional checks over their child structure. Each relevant child goal must be represented by supported execution, and the child-level outcomes must collectively support the parent-level procedure and completion condition. A candidate is admitted to persistent memory only when all required checks pass.

3.5 Procedure Retrieval and Reuse

GPM retrieves admitted Memory Cards according to the agent's current execution need. For positive and correction cards, semantic similarity produces an initial shortlist. For guard cards, goal-pattern similarity produces one shortlist, while an LLM examines the goals, applicability conditions, and guard statements of the remaining cards to nominate additional candidates. The two guard candidate sets are deduplicated before reranking. These recall steps favor coverage; they do not decide whether a candidate should guide the current task.

A structured LLM reranker makes that decision. For positive and correction cards, it compares the candidate's goal and expected outcome with the current execution need. It rejects explicit conflicts in goal ownership, object roles, operation direction, required state transitions, or fixed task constraints. A separate guard reranker checks whether the guard's goal and negative constraint apply under the same task constraints. Conditions that depend on live environment state remain unresolved at this stage and are passed to the Executor for observation-based validation.

The Goal Tree permits reuse at the granularity represented in memory. A Local Memory Card is retrieved as one procedure. For a Composite Memory Card, the reranker may select the full card or the individual nodes that match the current execution need. Each selected node must independently match that need. Retrieval can select an existing subprocedure, but it cannot combine unrelated nodes into a new procedure that is absent from the card. The Executor later verifies any applicability condition that depends on the live environment.

Retrieved cards are supplied to the Executor as procedural guidance. They do not establish current environment facts and do not indicate that any part of the current task has already been completed. The Executor binds procedure variables to the current task, observes the relevant state, and checks the applicability and success conditions during execution. If only selected nodes of a Composite Memory Card are retrieved, the remaining nodes provide no guidance for that execution.

Positive and correction cards provide procedures to follow, while guard cards provide conditions or behaviors to avoid. Function Memory follows a separate failure-triggered path. After a function call fails, GPM retrieves validated repairs for the same function and exposes a repair only when it is compatible with the observed failure.

4 Experiment

4.1 Experimental setup

Benchmark and protocol. We evaluate on AppWorld, an interactive benchmark in which agents complete natural-language tasks by operating multiple simulated applications through APIs. We use 90 tasks from the official training split to construct a GPM memory bank with MiMo-V2.5. The resulting memory bank is frozen before evaluation and used for both backbone evaluations, and evaluation trajectories are never written back into memory. We use the complete official `test_normal` split of 168 tasks for the main downstream evaluation and the official development set of 57 tasks for controlled ablations.

Models and agent configuration. For the main downstream evaluation, we use two executor backbone models, MiMo-V2.5 and Qwen3.7-Flash. We disable model reasoning and use temperature $T = 0$ and seed 0 throughout. The executor follows a ReAct-style interaction loop, and all compared configurations under the same backbone share the same executor and base prompt. Unless otherwise specified, all controlled mechanism experiments, including the admission study and memory-granularity ablations, use MiMo-V2.5. We keep these experiments on a single backbone to isolate the corresponding design factors rather than introducing additional variation from the underlying model.

Baselines. We compare three configurations under each backbone in the main downstream evaluation. **ReAct** is the memoryless base executor. **ReAct+TODO** augments the same executor with the lightweight execution-state mechanism used by GPM but provides no persistent cross-task memory. **GPM** further equips ReAct+TODO with our complete memory pipeline, including post-hoc goal-

Table 1: Main results on the complete AppWorld `test_normal` split (168 tasks) with two backbone models. Results are averaged over three repeated runs. TGC is Task Goal Completion; higher is better.

Backbone	Method	TGC Avg@3 (%) ↑
MiMo-V2.5	ReAct	81.2
	ReAct+TODO	81.7
	GPM	**85.5**
Qwen3.7-Flash	ReAct	73.6
	ReAct+TODO	73.0
	GPM	**81.0**

Table 2: Controlled memory-admission study. IA is Invalid Admission (lower is better); VR is Valid Retention (higher is better).

Admission policy	IA (%) ↓	VR (%) ↑
No verification	42.9	**100.0**
Full-trajectory scope	39.5	92.9
Goal-tree scope	**31.9**	91.7

tree reconstruction, memory abstraction and admission, retrieval, and runtime reuse. Within each backbone, all configurations use the same model, base prompt, and evaluation protocol.

Evaluation metrics. For downstream agent performance, we report **Task Goal Completion (TGC)** under the official AppWorld evaluator, defined as the fraction of tasks for which all required evaluator conditions are satisfied. Downstream results are averaged over three repeated runs (**Avg@3**). For the controlled admission study, we use a manually adjudicated set of valid and invalid candidate memories and report **Invalid Admission (IA)**, the proportion of admitted Memory Cards judged invalid, measuring contamination among stored memories, and **Valid Retention (VR)**, the fraction of valid candidates that remain admitted. Lower IA and higher VR are better.

4.2 Does GPM improve agent execution?

We first evaluate the end-to-end effect of GPM on the complete AppWorld `test_normal` split using two different backbone models. ReAct+TODO controls for the lightweight execution-state mechanism used by our executor, while GPM additionally enables cross-task memory construction, retrieval, and reuse. Evaluating the same three configurations with both MiMo-V2.5 and Qwen3.7-Flash tests whether the downstream effect of GPM is specific to a single executor backbone.

As shown in Table 1, GPM consistently outperforms the strongest non-GPM baseline under both backbones. With MiMo-V2.5, GPM improves TGC from 81.7 to 85.5, an absolute gain of 3.8 percentage points over ReAct+TODO. With Qwen3.7-Flash, GPM improves TGC from 73.6 to 81.0, an absolute gain of 7.4 points over ReAct. These results show that ***GPM provides consistent downstream gains across two backbone families***.

4.3 Does goal-tree-scoped evidence improve memory admission?

The downstream result measures the utility of GPM as a complete system, but does not directly measure whether an admitted Memory Card is supported by its source execution. We therefore evaluate admission quality independently on a fixed, manually adjudicated set of 156 candidate Memory Cards.

We compare three conditions. **No verification** applies the shared admission pipeline while omitting semantic verification. **Full-trajectory scope** verifies each candidate using the complete source execution context. **Goal-tree** scope uses the same verifier but restricts the available evidence to the goal-tree region assigned to the candidate Memory Card.

Table 3: Memory-granularity ablation on the AppWorld development set. All variants use the same GPM pipeline and retrieval and memory-injection budgets; only the available Memory Card granularities differ.

Available Memory Cards	TGC Avg@3 (%) ↑
Local only	73.7
Composite only	74.3
Local + Composite	**81.9**

Table 2 shows that semantic verification alone provides only a modest reduction in invalid admission when applied to the full trajectory. Restricting verification to the executed goal tree further reduces IA from 39.5 to 31.9, a 7.6-point improvement, while VR changes only from 92.9 to 91.7. The gain supports the intended mechanism of goal-scoped verification: excluding execution records that are irrelevant to the candidate procedure ***reduces the chance that unrelated evidence is mistaken as support***.

4.4 Does variable-granularity reuse improve execution?

A recovered goal tree exposes reusable experience at different structural scales. As defined in Section 3, a **Local Memory Card** captures a procedure grounded within an individual goal, whereas a **Composite Memory Card** captures reusable structure that spans multiple related goals. GPM retains both types and allows retrieval to select the granularity that best matches the current request.

We test whether this variable-granularity design is useful, or whether a single fixed reuse unit is sufficient. The ablation is conducted on the official AppWorld development set of 57 tasks. **Local-only** removes all Composite Memory Cards, so retrieval can reuse only goal-local procedures. **Composite-only** removes all Local Memory Cards, so retrieval can reuse only multi-goal procedures. **Variable granularity** retains both card types and lets the same retrieval pipeline choose among them. All three configurations use the same GPM construction, admission, retrieval, reranking, and execution pipeline, as well as the same retrieval and memory-injection budget, only the available Memory Card granularity changes.

Restricting GPM to either Memory Card type substantially reduces downstream performance. Allowing both Local and Composite Memory Cards improves TGC by 8.2 points over Local-only reuse and by 7.6 points over Composite-only reuse. These results indicate that a single fixed reuse unit is insufficient: goal-local procedures and procedures spanning multiple goals provide useful reusable structure at different scales, and ***GPM benefits from preserving both as retrieval candidates***.

5 Conclusion

We introduced GPM, a training-free framework that reconstructs a Goal Tree from completed ReAct trajectories in which goals, failed attempts, and corrections are interleaved. The tree defines local and composite sources for procedure extraction, bounds the execution evidence used for memory admission, and supports reuse at different granularities without requiring hierarchical planning during execution. On AppWorld, GPM improves Task Goal Completion by 3.8 and 7.4 percentage points over the strongest non-GPM baseline under each of two executor backbones. Goal-scoped verification reduces invalid admission by 7.6 points relative to full-trajectory verification while largely preserving valid memories. The granularity ablation further shows that retaining both Local and Composite Memory Cards outperforms retaining either type alone. Together, these findings support using the same recovered goal structure to define what can be extracted, which execution records justify storing it, and how it can be reused.

AI use statement

We used generative AI tools to identify potentially relevant related work, provide feedback on research methodology and experimental design, assist with implementing the proposed method in code, help analyze experimental results, and improve the wording and clarity of the manuscript. All

AI-assisted outputs were reviewed and verified by the authors. The authors take responsibility for the final manuscript, implementation, claims, and reported results.

Reproducibility statement

We provide the benchmark splits, model and agent configurations, evaluation protocol, and randomization settings in Section 4.1. Detailed implementation specifications for Goal Tree reconstruction, Memory Card representation, memory admission, and retrieval and executor integration are provided in Appendices B–D.

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 Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.829. URL https://aclanthology.org/2026.findings-acl.829/.

Zheyuan Deng, Binghang Lu, Hanqi Feng, Shirley Huang, Dianzhuo Wang, Yuanda Xu, Zhiwei Zhang, Yige Sun, Changhong Mou, Runyu Zhang, Yuexing Hao, Barnabas Poczos, and Xiaomin Li. Contramem: Learning self-evolving procedural memory from contrasting multi-model trajectories, 2026. URL https://arxiv.org/abs/2608.22533.

Shengda Fan, Xin Cong, Zhong Zhang, Yuepeng Fu, Yesai Wu, Hao Wang, Xinyu Zhang, Enrui Hu, and Yankai Lin. Generalizing experience for language agents with hierarchical metaflows. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 64103–64132. Curran Associates, Inc., 2025. doi: 10.52202/085713-2149. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5c882988ce5fac487974ee4f415b96a9-Paper-Conference.pdf.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.866. URL https://aclanthology.org/2026.findings-acl.866/.

Saman Forouzandeh, Wei Peng, Parham Moradi, Xinghuo Yu, and Mahdi Jalili. Learning hierarchical procedural memory for llm agents through bayesian selection and contrastive refinement, 2025. URL https://arxiv.org/abs/2512.18950.

Sijia Li, Yuchen Huang, Zifan LIU, Zijian Li, Jingjing Fu, Lei Song, Jiang Bian, Jun Zhang, and Rui Wang. Experience-evolving multi-turn tool-use agent with hybrid episodic–procedural memory. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=PJ0GpmFYrR.

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 (ICML 2026)*, 2026. URL https://arxiv.org/abs/2602.01869. Spotlight.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long 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. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 94327–94354,

2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/980ea04d23d1f6908964eba2a74afe45-Paper-Conference.pdf.

Yunteng Tan, Zhi Gao, and Xinxiao Wu. Enhancing web agents with a hierarchical memory tree, 2026. URL https://arxiv.org/abs/2603.07024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, and Shumin Deng. Skillx: Automatically constructing skill knowledge bases for agents, 2026. URL https://arxiv.org/abs/2604.04804.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Cheng Yang, Xuemeng Yang, Licheng Wen, Daocheng Fu, Jianbiao Mei, Rong Wu, Pinlong Cai, Yufan Shen, Nianchen Deng, Jia Xu, Botian Shi, Yu Qiao, and Haifeng Li. Towards self-evolving agents: Enabling autonomy through interactive experience refinement. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30424–30451, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1522. URL https://aclanthology.org/2026.findings-acl.1522/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Zhaochen Yu, Yingcheng Wu, Zhenfei Yin, Kaiyuan Chen, Zhe Zhao, Mengdi Wang, Shuicheng Yan, and Ling Yang. Recursive experiential-working memory evolution for long-horizon agent harnesses, 2026. URL https://arxiv.org/abs/2608.24876.

Guilin Zhang, Wei Jiang, Xiejiashan Wang, Aisha Behr, Kai Zhao, Jeffrey Friedman, Xu Chu, and Amine Anoun. Adaptive memory admission control for LLM agents. In *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, 2026. URL https://openreview.net/forum?id=mmdqUrEY24.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pp. 19632–19642. AAAI Press, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 19036–19066, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/52f050499cf82fa8efb588e263f6f3a7-Paper-Conference.pdf.

Table 4: Primary error categories among invalid admissions under Full-trajectory and Goal-tree scoped verification.

Error type	Full-trajectory scope	Goal-tree scoped
Declaration-only support	13	10
Unsupported generalization	19	18
Temporal leakage	12	2
Capability–state confusion	7	6
Total invalid admissions	51	36

A Additional Experiments and Analysis

A.1 Admission Error Analysis

To understand where evidence scope affects admission decisions, we categorize invalid candidates admitted by the Full-trajectory scope verifier and compare their outcomes under Goal-tree scoped verification. We focus on errors that can arise when evidence outside the candidate's supported execution region influences the admission decision, including declaration-only support, unsupported generalization, temporal leakage, and capability–state confusion. Table 4 summarizes the resulting error counts under Full-trajectory and Goal-tree scoped verification.

This analysis complements the aggregate IA and VR results by identifying which admission errors are affected by restricting the verifier's evidence domain.

B Implementation details

B.1 Goal-tree records

The Goal Tree stores goal structure and action ownership. Executed actions and their observations remain in a separate action ledger and are referenced by action ID. The fields used by downstream memory construction are:

```
{
  goal_id: string,
  parent_id: string | null,
  description: string,
  success_condition: string,
  local_action_ids: [string],
  outputs: [{key, value, source_action_ids: [string]}],
  execution_grounded: bool
}
```

The tree is stored as a list of nodes. Parent–child edges are encoded by `parent_id`: the children of g are the nodes whose `parent_id` equals g's `goal_id`. The subtree rooted at g is obtained by recursively following these edges. The `local_action_ids` field contains only actions directly assigned to the node; descendant actions are obtained through the tree structure. Each output retains the action IDs supporting it. The `execution_grounded` flag is derived from whether the node or a descendant owns an executed action. It does not, by itself, establish that the goal succeeded.

B.2 Tree reconstruction

Reconstruction proceeds from ordered, non-overlapping execution phases to action-owning goals and then to their hierarchical decomposition. Every task-execution action is assigned to exactly one goal. Parent–child edges express that a child is a constituent of its parent; action order alone does not create such an edge. Failed attempts and repairs remain in the recovered execution rather than being replaced by an idealized procedure.

TODO records provide hints about phase boundaries and goal names. The trajectory and action ledger determine action ownership, while the observed execution determines the goal hierarchy. A TODO's wording or completion status is not treated as evidence that its stated goal was achieved.

B.3 Source and evidence bookkeeping

As defined in Section 3.3, a local source consists of a leaf goal and its directly assigned actions, while a composite source consists of an internal goal, its descendants, and their actions. For each selected source, the Selector identifies supporting execution records within the source's action scope. These selected records are used to construct the candidate Memory Card.

Admission uses the full source-scoped record set τ_s defined in Section 3.4. Records in τ_s outside the selected support can reveal an omitted operation or a contradiction. They do not become additional support for claims made by the candidate.

B.4 Memory Card representation

A Memory Card uses a parameterized goal pattern and records the conditions and operations relevant to that goal. Its node payload has the following schematic fields:

```
{
  source_node_id,
  goal_pattern,
  applicability_conditions,
  method: {summary, steps},
  success_condition_pattern,
  defect,
  repair,
  guards
}
```

Positive cards use the goal, applicability, method, and success fields. Correction cards additionally record the observed defect and its validated repair. Guard cards record a supported condition or behavior to avoid without supplying an unobserved repair. Method steps preserve the order and direction of operations that were executed; task-specific identifiers are replaced by parameters only where the source goal and evidence support that abstraction. Claims about executed steps and observed outcomes retain references to their supporting records.

A Local Memory Card contains one procedure node. A Composite Memory Card preserves the source's parent–child structure: its root represents the overall procedure, and its child nodes represent supported lower-level procedures. Function-level failure–repair cases are stored separately as Function Memory.

C Admission Protocol

A Memory Card remains a candidate until it passes admission. Admission begins with deterministic structural validation. This stage resolves every cited execution record and checks that it belongs to the source-scoped record set τ_s defined in Section 3.4. It also checks action ownership, the ordering of a validated failure–repair sequence where applicable, and branch coverage for composite sources. A candidate that fails these checks is rejected before semantic verification.

The LLM Verifier checks the candidate against the full source-scoped record set τ_s. Records outside the Selector's selected support can reveal omissions or contradictions, but cannot provide positive support for additional claims. An executed operation grounds a claim about what was performed; claims about returned content or persistent state additionally require corresponding observed results.

For a positive candidate, the Verifier checks method faithfulness, grounding of the claimed effect, scope safety, and abstraction quality. A composite positive candidate receives additional checks for child coverage, grounding of the decomposition, and support for the parent-level outcome. Cor-

rection and guard candidates receive checks for their respective defect, repair, failure-condition, or avoidance claims. Admission requires every applicable check to pass.

D Retrieval and Executor Integration

Positive and correction cards are shortlisted by semantic similarity. For guard cards, goal-pattern similarity produces one shortlist; an LLM examines the goals, applicability conditions, and guard statements of the remaining cards to nominate additional candidates. The guard candidate sets are deduplicated before reranking.

Separate structured rerankers assess positive or correction cards and guard cards. They check whether a candidate's goal and relevant constraints match the current execution need, rejecting explicit conflicts in object roles, operation direction, required state transitions, or fixed task constraints. Conditions that depend on live environment state are left for the Executor to validate from observations during execution.

A Local Memory Card supplies one procedure. For a Composite Memory Card, retrieval may select the complete procedure or individual nodes that independently match the current need. Separate nodes are not combined to infer an unstated procedure. Function Memory follows a separate path after a function-call failure and supplies a validated same-function repair only when it is compatible with the observed failure.

Retrieved cards provide procedural guidance. The Executor binds their variables to the current task and checks applicability and success conditions against observations obtained during execution. A retrieved card does not establish current environment state or prove that the current task is complete.