PROSEVO: ROLE-AND-CARRIER ATTRIBUTION FOR CLOSED-LINEAGE AGENT SELF-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agents can retain interaction experience through external memory or parameter updates, but capability provenance becomes difficult to isolate when additional generative models participate in the learning loop. We study *closed-lineage self-evolution*, where experience collection, offline compilation, parameter updating, and subsequent agent generation remain within a single locally hosted generative backbone lineage, without introducing a stronger external generator. This setting creates a self-confirmation risk: unsupported self-generated behavior may be repeatedly interpreted and reinforced across rounds. We introduce ProSEvo, an evidence-grounded framework for closed-lineage self-evolution that addresses this risk through role-and-carrier attribution. A frozen same-backbone compiler labels recorded events and proposes procedural summaries, while deterministic validation constrains structure and source provenance. Admitted evidence is persisted through two capability carriers: typed procedural memory is routed to its consuming role, whereas original positive outputs provide memory-free supervision to their producing role. Successive students generate experience for their successors, jointly evolving external memory and role-specific parameters. Across OfficeBench, AppWorld, and τ-bench Retail under a unified same-backbone protocol, ProSEvo increases macro task success from 17.39% to 29.12%, compared with 22.88% for memory-only and 23.65% for parameter-only variants. Fixed-checkpoint interventions and round-wise controls further support the contributions of compiled memory, parametric internalization, and cumulative experience reuse within the evaluated closed-lineage setting.

1 INTRODUCTION

Large language models (LLMs) increasingly act as agents that reason, invoke tools, and interact with external environments. ReAct and Toolformer established foundations for coupling model reasoning with external actions (Yao et al., 2023; Schick et al., 2023), while agent harnesses add task decomposition, outcome verification, recovery, and fallback execution. Beyond improving an individual rollout, such harnesses elicit structured behavior that could become experience for later agents. Turning that interaction into persistent improvement remains difficult: OfficeBench, AppWorld, and τ-bench expose continuing failures in long-horizon planning, grounded execution, cross-application coordination, and policy adherence (Wang et al., 2024a; Trivedi et al., 2024; Yao et al., 2024).

Studying agent self-evolution requires specifying both how experience is retained and which generative models contribute to the learning loop. Memory-based agents preserve feedback and procedures for retrieval (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024; Han et al., 2026), whereas self-training and distillation update model parameters from selected behavior (Hinton et al., 2015; Zelikman et al., 2022; Gulcehre et al., 2023). We examine these two persistence mechanisms under a closed generative lineage: the collector, compiler, and deployed agent roles are instantiated from one base checkpoint, with role-specific student adapters updated across rounds. This controls the introduction of a stronger external generator. It does not remove the contributions of pretrained knowledge, environment feedback, hand-written harnesses, tool schemas, or the local retrieval encoder. Our focus is how recorded training interactions are admitted and reused under this constraint.

Using the same backbone for generation and interpretation leaves a risk of self-confirmation: unsupported interpretations may be retained and reinforced in later rounds (Zhu et al., 2026). ProSEvo

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.

addresses this risk through evidence-based admission, separating the environment's terminal success signal from the compiler's event-level labels. Terminal success makes a trajectory eligible for positive training data; it does not make every intermediate step correct. Source references, execution status, and schema constraints further restrict which artifacts are retained. Localized failures provide correction guidance through memory rather than making an unsuccessful rollout a positive SFT target. These checks constrain unsupported artifacts, but they do not independently certify every natural-language interpretation produced by the compiler.

Admission also leaves a reuse decision: which component should consume the retained evidence, and in what form? We use role attribution to denote the assignment of a memory record to its intended consumer, and carrier assignment to distinguish external retrieval from parameter supervision. These are explicit routing and construction rules, not estimates of numerical causal credit. A source action can support a controller-facing procedure while remaining an executor target for SFT; the consumer of a derived memory need not be the producer of its source event. We therefore ask how evidence-grounded memory construction and role-specific self-distillation can support cumulative improvement within a fixed controller–executor interface and a single generative backbone lineage.

ProSEvo begins with interactions collected by a frozen, harnessed copy of the base model. Later rounds use the preceding student to collect additional experience. A frozen same-backbone compiler annotates completed trajectories and proposes Structure and Failure Memory fields. After validation, deterministic builders materialize typed memory and extract original positive role outputs for memory-free SFT. The updated adapters and memory bank form the next agent state. We use this pipeline to examine external retrieval, parameter updating, and cumulative reuse of student-generated experience under a common interaction interface.

Our contributions are as follows.

- **Closed-lineage self-evolution.** We formulate cumulative agent self-evolution within one locally hosted generative backbone lineage, without introducing a stronger external generator, with explicit boundaries across collection, compilation, updating, and deployment.
- **Role-and-carrier attribution.** We introduce evidence-grounded attribution that routes procedural memory to its consuming role and original positive outputs to their producing role for supervision, thereby separating memory consumption from parameter supervision.
- **Cumulative dual-carrier evolution.** ProSEvo jointly evolves external memory and role-specific parameters as successive students generate new experience. Across three benchmarks, Macro increases from 17.39% to 29.12%, with fixed-checkpoint and round-wise controls supporting the contributions of both carriers and cumulative experience.

2 Related Work

Experiential agents and procedural memory. Experience-based agents retain feedback, rules, and workflows for later decisions (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024; Wang et al., 2024b). ReasoningBank distills reusable strategies from self-judged successful and failed experience (Ouyang et al., 2025). Contextual Experience Replay provides training-free improvement by retrieving accumulated experience (Liu et al., 2025). ReMe manages procedural memory through experience distillation, context-adaptive reuse, and utility-based refinement (Cao et al., 2026). These approaches motivate treating memory construction and reuse as substantive parts of agent learning. ProSEvo studies typed retrieval alongside cumulative role-specific parameter updates, with terminal environment verification and explicit source references. This combination defines our study setting; it does not imply superiority to memory systems that are not evaluated under the same protocol.

Verified self-improvement and agent distillation. STaR and ReST convert generations selected by correctness or reward into training data (Zelikman et al., 2022; Gulcehre et al., 2023). Subsequent methods derive reflection and recovery-oriented supervision from unsuccessful behavior (Dou et al., 2024; Yuan et al., 2025; Zhao et al., 2026), while Skill-SD, HINT-SD, HERO, and SEED distill reusable skills, hindsight signals, or environment-grounded feedback from agent trajectories (Wang et al., 2026; Yeo et al., 2026; Liu et al., 2026; Wu et al., 2026). ProSEvo combines terminal verification with event-level admission: positive original role outputs from successful trajectories provide

SFT targets, while supported negative events provide retrievable correction guidance. Its supervised targets are copied from admitted source events rather than generated as new executable actions by the compiler.

Self-evolving experience and memory systems. Self-evolving agents progressively refine how interaction experience is distilled, retained, and reused (Gao et al., 2026). EvolveR couples experience distillation with online interaction and policy evolution, while EDV separates execution, distillation, and verification to construct reliable experience (Wu et al., 2025; Zhu et al., 2026). MemSkill learns and evolves memory operations, MUSE accumulates hierarchical knowledge through post-execution critique, and DecentMem maintains decentralized exploitation and exploration memories for individual agents (Zhang et al., 2026; Yang et al., 2026; Hao et al., 2026). ProSEvo builds on these directions by separating full-trace interpretation from the deterministic construction of typed memory and original-output supervision. Our analysis concerns the reuse of cumulative training experience through these two persistence mechanisms within a shared generative backbone lineage.

Agent architecture and coordination. OneFlow establishes single-agent execution of multi-agent workflows as a strong comparison point (Xu et al., 2026). Research on scaling agent systems further shows that coordination outcomes depend on task and system conditions (Kim et al., 2025). ProSEvo uses separate controller and executor adapters as its study interface. Neither different adapter parameters nor role-swapped retrieval tests establish that this interface is preferable to a matched unified agent or shared-adapter alternative. Our present claims therefore concern experience reuse within the chosen interface, rather than a general advantage of multi-agent decomposition.

3 Problem Formulation

We study how environment-verified interaction can be converted into persistent capability under a closed generative lineage. The learned agent state contains two persistent capability carriers: role-specific parameters and typed procedural memory.

Interaction and verified evidence. Let $\mathcal{X}_{\mathrm{tr}}$ and $\mathcal{X}_{\mathrm{ev}}$ denote disjoint training and evaluation task sets in an interactive environment $\mathcal{E}$. For task x, the agent produces a trajectory

$$\tau = (x, e_1, \ldots, e_T, y), \qquad e_t = (g_t, u_t, o_t), \qquad g_t \in \mathcal{G} = \{\mathrm{pc}, \mathrm{ex}\}, \tag{1}$$

where g_t is the role producing output u_t, and o_t records the associated observation or execution evidence. The planner–controller produces planning, delegation, completion, or customer-facing decisions, whereas the executor produces environment-grounded actions.

The decision-time context h_t contains only information available before u_t is produced, including the current delegation when the executor acts. It excludes the resulting observation o_t and all later events. The same causal boundary is imposed when constructing parameter supervision.

The environment verifier evaluates terminal task completion:

$$V_{\mathcal{E}}(\tau) \in \{0, 1\}. \tag{2}$$

A successful trajectory is eligible to provide positive learning evidence, but terminal success does not certify every intermediate event.

Two persistent capability carriers. After update round r, the persistent agent state is

$$\mathcal{A}_r = (\theta_r, \mathcal{M}_r), \qquad \theta_r = (\theta_r^{\mathrm{pc}}, \theta_r^{\mathrm{ex}}), \tag{3}$$

where θ_r contains role-specific parameters and $\mathcal{M}_r$ contains typed procedural memory. At decision time,

$$\widehat{\mathcal{M}}_{r,t}^{g_t} = \rho_{g_t}(h_t, \mathcal{M}_r), \qquad u_t \sim \pi_{\theta_r^{g_t}}\left(\cdot \mid h_t, \widehat{\mathcal{M}}_{r,t}^{g_t}\right). \tag{4}$$

Thus, parameters provide internalized role behavior, whereas retrieval supplies state-dependent procedural experience. Reusable capability may therefore persist through parameters, external memory, or both.

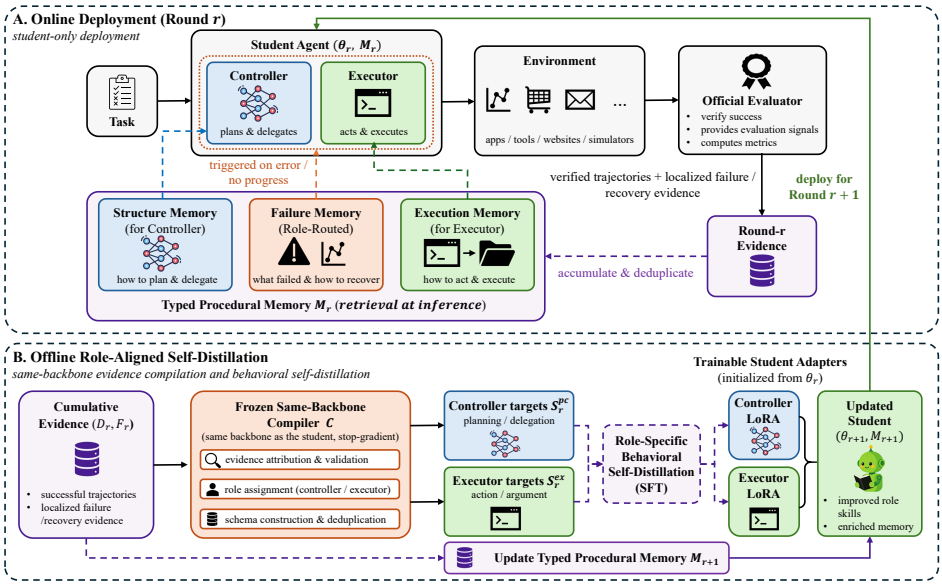


Figure 1: Overview of ProSEvo. Training experience is compiled and validated into typed memory and original-output role supervision, supporting updates from $(\theta_r, \mathcal{M}_r)$ to $(\theta_{r+1}, \mathcal{M}_{r+1})$. Evidence and supervision indexed by r here correspond to update $r+1$ in the main text. The frozen compiler operates only offline.

Closed-lineage self-evolution. Let $\mathcal{B}$ denote the locally hosted base checkpoint, and let $\mathcal{G}_0$ denote a frozen, harnessed copy used exclusively for bootstrap collection. The offline compiler is another frozen copy of the same backbone, while the planner–controller and executor use role-specific adapters initialized from $\mathcal{B}$. Experience collection follows

$$\begin{aligned} \mathcal{R}_0 &= \mathrm{Rollout}(\mathcal{G}_0; \mathcal{X}_{\mathrm{tr}}, \mathcal{E}), \\ \mathcal{R}_r &= \mathrm{Rollout}(\mathcal{A}_{r-1}; \mathcal{X}_{\mathrm{tr}}, \mathcal{E}), \qquad r \in \{1, \ldots, R\}. \end{aligned} \tag{5}$$

After bootstrap collection, each updated student therefore becomes the experience generator for its successor. This closed generative lineage constrains the source of generative capability; environment feedback, hand-written interfaces, and the local retrieval encoder remain additional non-generative inputs.

Role-and-carrier attribution. Verified interaction does not imply that every admitted piece of evidence should be consumed by the same role or persisted through the same mechanism. ProSEvo therefore separates role and carrier attribution. For the memory carrier, an admitted record is assigned to the role that should consume the retrieved procedure. For the parameter carrier, supervision remains attached to the role that produced the admitted original output. The same source evidence may support either carrier or both, so memory consumption and parameter-target production need not coincide.

This yields a carrier-specific evidence attribution problem: which reusable information should become externally retrievable procedural memory, which original behavior should be internalized into role-specific parameters, and which role should consume each resulting artifact.

The resulting learning problem is to construct a sequence $\mathcal{A}_0, \ldots, \mathcal{A}_R$ in which successive students accumulate, reuse, and extend verified training experience while evaluation remains strictly held out. Full definitions of cumulative evidence pools, memory partitions, role-specific supervision, optimization, and held-out evaluation are given in Appendix B.1.

4 METHOD

Figure 1 summarizes the evolution loop. Completed training interactions provide source records for evidence-grounded compilation, and the resulting typed memory and original-output supervision update the student that collects the next round of experience.

4.1 VERIFIED EVIDENCE ACCUMULATION

A harnessed, frozen copy of the initial student bootstraps experience through task decomposition, outcome checks, and bounded replanning. Successive students collect additional experience through the closed-lineage interface in Equation (5). For round r, let

$$\mathcal{V}_r = \{\tau \in \mathcal{R}_r \mid V_{\mathcal{E}}(\tau) = 1\}$$

denote the terminally verified trajectories. ProSEvo incrementally maintains

$$\begin{aligned} \mathcal{D}_r &= \mathrm{TraceDedup}\big(\mathcal{D}_{r-1} \cup \mathcal{V}_r\big), \\ \mathcal{F}_r &= \mathrm{EvidenceDedup}\big(\mathcal{F}_{r-1} \cup \mathrm{LocalFail}(\mathcal{R}_r)\big), \end{aligned} \tag{6}$$

where $\mathcal{D}_{-1} = \mathcal{F}_{-1} = \varnothing$. The verified pool retains successful training trajectories, whereas $\mathcal{F}_r$ retains source-linked localized failures, including failures within ultimately successful trajectories. Pool membership alone does not imply admission of a memory or SFT artifact; admission additionally requires the compilation and validation rules below. Detailed cumulative-pool definitions and deduplication are given in Appendix B.1.

4.2 EVIDENCE-GROUNDED COMPILATION

The compiler operates on a deterministically projected completed training trajectory rather than on an isolated action. For source trajectory τ, let $\mathrm{Proj}(\tau)$ contain the task, available interface or policy, ordered source events, recorded observations, and official outcome. With fixed schema Σ, the frozen compiler produces

$$\begin{aligned} o_\tau &= C(\mathrm{Proj}(\tau); \Sigma)\,, \\ a_\tau &= A_{\mathrm{sch}}(o_\tau) \wedge A_{\mathrm{src}}(o_\tau, \tau). \end{aligned} \tag{7}$$

The compiler output contains event labels, an optional Structure summary, and Failure entries; it does not contain newly generated executable actions. Schema and source checks validate output structure, source identifiers, recorded execution or acceptance status, support-label consistency, and recovery ordering. Rejected outputs produce neither memory nor SFT examples.

For an accepted output, deterministic builders materialize persistent artifacts. Execution Memory copies actions and arguments from original positive executor events; Structure Memory retains a supported procedural summary; and Failure Memory retains source-grounded error and correction guidance. Routing assigns each memory artifact a permitted consumer. Parameter supervision is constructed separately by pairing admitted original role outputs with their recorded memory-free decision contexts.

Terminal verification, compiler-produced labels, and deterministic validation therefore serve different functions. These checks constrain structure and provenance rather than independently certifying every semantic interpretation. A recovery link is recorded only when supported by a later positive event; otherwise, Failure Memory does not claim an observed successful recovery.

4.3 TYPED AND ROLE-ALIGNED MEMORY

Each admitted memory record has a functional type and an explicit consuming role. Structure Memory is consumed by the planner–controller and stores supported high-level procedures and completion criteria. Execution Memory is consumed by the executor and preserves positive, actually executed or accepted actions together with their grounded constraints. Failure Memory is routed to the role responsible for the corresponding failure: decomposition and completion-check failures are controller-facing, whereas tool-selection and argument failures are executor-facing.

The detailed type–role partitions of $\mathcal{M}_r$, together with the exact materialization and routing rules, are specified in the appendix. In particular, a Failure record may contain a supported correction

without claiming that the correction was itself executed successfully. We use *Success Memory* as shorthand for $\mathcal{M}_r^{\mathrm{str,pc}} \cup \mathcal{M}_r^{\mathrm{exe,ex}}$.

4.4 Role-Specific Parameter Internalization

For recorded event e_t, let g_t denote its producing role and $\ell_\tau(e_t) \in \{+, -, 0\}$ its compiler label. Let $c_t^{g_t}$ be the stored role input before optional retrieved references are appended. The admitted role-specific supervision is

$$\mathcal{S}_r^g = \{(c_t^g, u_t) : \tau \in \mathcal{D}_r,\ a_\tau = 1,\ \ell_\tau(e_t) = +,\ g_t = g\}, \qquad g \in \{\mathrm{pc}, \mathrm{ex}\}. \tag{8}$$

The target u_t is the original role output rather than compiler-generated executable content. Controller targets preserve the admitted delegation, finish decision, or customer-facing response, whereas executor targets preserve the original application action, API program, or tool call. Parameter supervision therefore remains attached to the producing role, independently of the consumer of any memory derived from the same source evidence.

The compiler may inspect later observations when labeling a past event, but the stored SFT input must represent the corresponding decision-time context: it excludes retrieved-memory content, the observation caused by the target action, and all subsequent events. Detailed construction and temporal-validity rules are provided in the appendix.

4.5 Role-Aligned Procedural Retrieval

At decision time, each role queries memory using its currently visible task state. Candidate records are first filtered by memory type, consuming role, and application-, tool-, or stage-level compatibility, and are then ranked by cosine similarity using the local embedding encoder. The top-k compatible records are appended as non-authoritative procedural references.

Structure Memory is supplied to the planner–controller, Execution Memory to the executor, and compatible Failure Memory to the role responsible for the relevant error. Parameters therefore provide internalized role behavior, whereas retrieval provides state-dependent procedural experience. Exact queries, compatibility filters, retrieval keys, and serialization rules are specified in the appendix.

4.6 Cumulative Closed-Lineage Evolution

Round 0 is collected by the frozen harnessed base model and produces the evidence used to train the first student. For $r \geq 1$, the preceding student $\mathcal{A}_{r-1}$ collects $\mathcal{R}_r$. Cumulative admitted evidence then updates both $\mathcal{M}_r$ and the role-specific supervision $\mathcal{S}_r^g$, while each adapter is warm-started from θ_{r-1}^g. The resulting state $\mathcal{A}_r = (\theta_r, \mathcal{M}_r)$ generates the next collection round.

At held-out evaluation, the learned adapters and training-derived memory are fixed. The student backbone, role adapters, local retrieval encoder, environment interface, and decision-time state remain active, whereas the bootstrap collector and offline compiler are removed. Detailed deployment boundaries are specified in the appendix.

5 Experiments

5.1 Experimental Setup

Benchmarks and metrics. We evaluate ProSEvo on 152 OfficeBench tasks, 168 AppWorld episodes, and 40 tasks from the Retail subset of τ-bench, covering multi-application workflows, state-changing API operations, and policy-constrained customer service (Wang et al., 2024a; Trivedi et al., 2024; Yao et al., 2024). Our primary metric is environment-verified task success rate (SR). Train SR measures performance on the training manifests, whereas Test SR measures performance on disjoint evaluation manifests reserved for final testing. Macro is the unweighted mean Test SR across benchmarks, and Avg. Rank is the mean per-benchmark Test SR rank, with lower values indicating stronger performance. All learning and model-selection stages use only the training manifests.

Controlled protocol. All comparisons use locally hosted Qwen2.5-32B-Instruct instances with a common agent interface, matched interaction budgets, and deterministic decoding. ProSEvo uses copies of this backbone for experience collection, compilation, and both agent roles. The planner–controller and executor employ separate rank-16 LoRA adapters initialized from the same checkpoint and trained for two epochs with AdamW at a learning rate of 10^{-5}. Experience collection covers 148 OfficeBench, 90 AppWorld, and 74 Retail training tasks. Retrieval uses local all-MiniLM-L6-v2 embeddings with cosine similarity and role- and metadata-compatibility filtering; $k = 3$ is selected on the training tasks and fixed before evaluation. OfficeBench permits 60 orchestration iterations and eight actions per subtask, AppWorld permits 40 steps, and Retail permits 15 steps. Complete prompts, baseline implementations, and per-benchmark configurations appear in the supplementary material.

5.2 Main Results

Table 1 reports training and held-out test success under the unified protocol. ProSEvo achieved the highest Train and Test SR on every benchmark and an Avg. Rank of 1.00. Compared with Base, it improved Test SR by 10.53 percentage points (pp) on OfficeBench, 7.14 pp on AppWorld, and 17.50 pp on Retail, raising Macro from 17.39% to 29.12%. Against the strongest competing variant on each benchmark, ProSEvo achieved gains of 1.97, 1.79, and 2.50 pp, respectively. Its Macro exceeded the strongest competing result by 4.22 pp.

Table 1: ProSEvo achieves the **highest train and test success** (%) under the unified Qwen2.5-32B-Instruct protocol. Macro averages the three Test SRs; best and second-best values are bold and underlined, respectively, and lower Avg. Rank is better.

	OfficeBench		AppWorld		Retail		Macro	Avg. Rank ↓
Method	Train	Test	Train	Test	Train	Test	Test	Test
Base multi-agent	23.65	20.39	16.67	14.29	22.97	17.50	17.39	6.00
LEGOMem-style	<u>30.41</u>	<u>28.95</u>	20.00	16.07	27.03	22.50	22.51	4.00
Skill-SD-style	28.38	25.66	<u>26.67</u>	<u>19.64</u>	29.73	25.00	23.43	3.33
HINT-SD-style	29.05	26.32	25.56	19.05	31.08	27.50	24.29	<u>3.00</u>
HERO-style	27.03	24.34	24.44	17.86	<u>36.49</u>	<u>32.50</u>	<u>24.90</u>	3.67
ProSEvo	**34.46**	**30.92**	**30.00**	**21.43**	**39.19**	**35.00**	**29.12**	**1.00**

5.3 Mechanism Analysis

Figure 2 summarizes the contributions of the typed memory components and capability carriers. Each cell reports the decrease in Test SR relative to full ProSEvo, so larger values indicate a larger contribution from the removed component. The Average column is the cross-benchmark mean.

RQ1: Compiled and role-aligned memory. Table 2 reports effect sizes rather than repeating the absolute full-system scores in Table 1. Structure Memory contributes 1.31, 1.19, and 2.50 pp on OfficeBench, AppWorld, and Retail, respectively, while Execution Memory contributes 1.97, 1.79, and 2.50 pp. Together with the Success- and Failure-Memory results in Figure 2, these component interventions show that planning structure, grounded execution, and recovery knowledge each contribute to task success.

The fixed-checkpoint contrasts hold the final student parameters constant and change only the inference-time memory artifact. Compiled, role-aligned memory exceeds raw trajectories by 2.38–2.63 pp and the no-memory condition by 2.98–5.00 pp across the three benchmarks. Its advantages over role-swapped and length-matched shuffled memory are substantially larger, demonstrating the importance of role alignment and task-relevant content.

RQ2: Complementary capability carriers. The matched 2×2 design evaluates external memory and role-specific parameter updates both independently and in combination. Table 3 expresses the result as marginal Macro gains. External memory improves Macro by 5.49 pp when introduced alone and by 5.46 pp when added to the parameter-updated student. Parameter updates improve

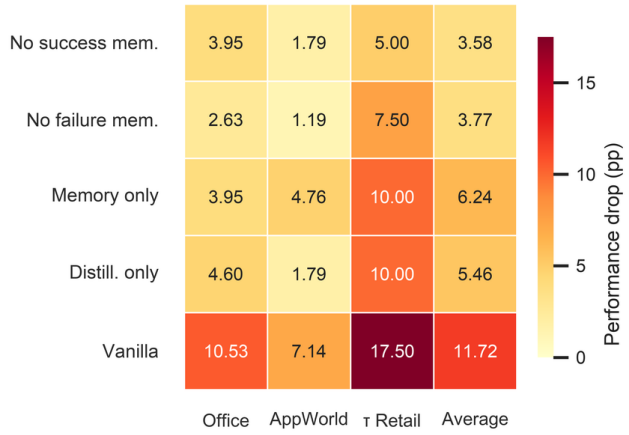


Figure 2: Both typed memory and parameter updates contribute across all three benchmarks. Cells report the Test SR decrease (pp) relative to full ProSEvo; Average is the cross-benchmark mean. "Distill. only" denotes the parameter-only variant.

Table 2: Effect sizes for memory components and fixed-checkpoint artifacts. Each row reports the Test SR advantage of the first condition over the second (pp); Mean is the cross-benchmark average.

Contrast	Office	AppW.	Retail	Mean
Full − w/o Structure	1.31	1.19	2.50	1.67
Full − w/o Execution	1.97	1.79	2.50	2.09
Compiled − Raw trajectory	2.63	2.38	2.50	2.50
Compiled − No memory	3.29	2.98	5.00	3.76
Compiled − Role-swapped	9.21	5.95	12.50	9.22
Compiled − Shuffled	14.47	10.12	20.00	14.86

Macro by 6.26 pp when introduced alone and by 6.24 pp when added to the memory-enabled agent. Both carriers therefore provide consistent gains with and without the other carrier.

Table 3: Marginal Macro gains from the two capability carriers. "Alone" measures the gain over Base; "With other" measures the incremental gain when the complementary carrier is already active. Values are pp.

Carrier added	Alone	With other	Average
External memory	5.49	5.46	5.48
Parameter updates	6.26	6.24	6.25

5.4 Cumulative Self-Evolution

RQ3: Cumulative experience and memory evolution. Figure 3 traces the three-round evolution of the cumulative student. Task success increases after each update on all three benchmarks, and the final student attains the strongest result. The matched controls isolate the sources of this continued improvement.

Table 4 reports the advantage of cumulative evolution over each matched control. Cumulative evolution exceeds Repeat R0 by 5.26, 5.95, and 12.50 pp, showing the contribution of newly collected environment experience under matched update budgets. Its gains over Fixed M0 show the contribution of memory evolution, while its gains over Current-only show the contribution of retaining historical experience. The effect is positive on every benchmark for all three comparisons.

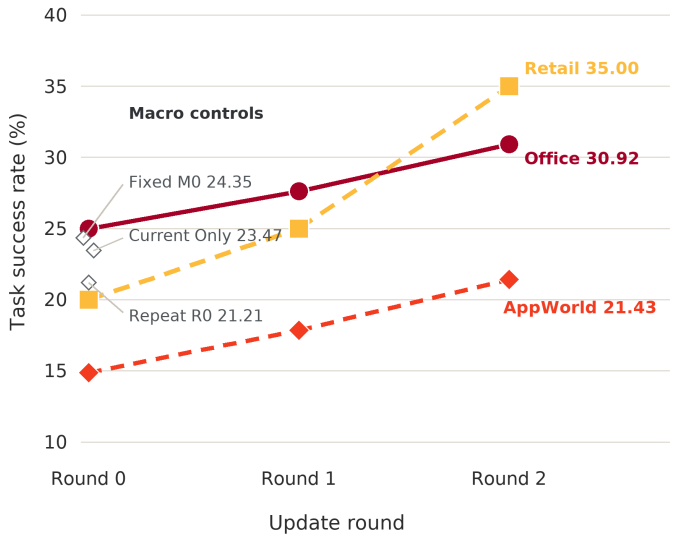


Figure 3: Cumulative updates improve task success on all three benchmarks. Colored lines trace Test SR across Rounds 0–2, while gray diamonds mark the final Macro of matched non-cumulative controls.

Table 4: Advantages of cumulative evolution over matched controls. Each row reports Cumulative − Control in Test SR (pp); Mean is the cross-benchmark average.

Control	Office	AppW.	Retail	Mean
Repeat R0	5.26	5.95	12.50	7.90
Fixed M0	2.63	4.17	7.50	4.77
Current-only	3.95	2.98	10.00	5.64

6 Limitations and Broader Impact

ProSEvo is evaluated in three verifiable environments under a fixed controller–executor interface. Terminal verification and deterministic validation constrain, but do not independently certify, compiler semantics or the temporal validity of every derived training view; independent semantic audits remain necessary. The current study also does not establish superiority over matched unified-agent, shared-adapter, or untyped-memory alternatives, and its point estimates do not capture independent training-run variability or paired statistical significance. The same-backbone constraint controls generative-model provenance while retaining pretrained knowledge, environment feedback, human-designed interfaces, and a local retrieval encoder. Broader domain transfer, robustness to weaker or changing verification, and deployment with sensitive data remain open and require appropriate access control, retention policies, and human review.

7 Conclusion

We presented ProSEvo, a closed-lineage agent self-evolution framework that reuses verified interaction through two persistent capability carriers: role-aligned procedural memory and role-specific parameters. Evidence-grounded admission constrains which self-generated experience is retained, while role-and-carrier attribution determines how that experience is routed and persisted. Successive students then generate new experience for their successors, enabling cumulative evolution within a single generative backbone lineage. Under the unified evaluation, Macro task success increases from 17.39% to 29.12%. Fixed-checkpoint, carrier, and round-wise controls further characterize the contributions of compiled memory, parameter updates, and cumulative experience within the evaluated interface.

AI USE STATEMENT

We used ChatGPT to assist with manuscript organization, English-language editing, LaTeX formatting, and the review of notation and method descriptions, as well as to identify potentially relevant literature. The authors are responsible for verifying the resulting text, technical claims, and bibliographic information against the implementation, experimental records, and original sources, and take full responsibility for the final manuscript.

REFERENCES

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

Zi-Yi Dou, Cheng-Fu Yang, Xueqing Wu, Kai-Wei Chang, and Nanyun Peng. Re-rest: Reflection-reinforced self-training for language agents. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15394–15411, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (ReST) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. LEGOMem: Modular procedural memory for multi-agent LLM systems for workflow automation. In *International Conference on Autonomous Agents and Multiagent Systems*, 2026.

Guangya Hao, Yunbo Long, and Zhuokai Zhao. Self-evolving multi-agent systems via decentralized memory. *arXiv preprint arXiv:2605.22721*, 2026.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Yubin Kim, Ken Gu, Chanwoo Park, et al. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025. URL https://arxiv.org/abs/2512.08296v3. Version 3, revised April 2026.

Haoran Liu, Yuwei Zhang, Xiyao Li, Bohan Lyu, and Jingbo Shang. HERO: Hindsight-enhanced reflection from environment observations for agentic self-distillation. *arXiv preprint arXiv:2606.11559*, 2026.

Yitao Liu, Chenglei Si, Karthik R Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.694. URL https://aclanthology.org/2025.acl-long.694/.

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. *arXiv preprint arXiv:2509.25140*, 2025. URL https://arxiv.org/abs/2509.25140.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

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

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

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

Hao Wang, Guozhi Wang, Han Xiao, Yufeng Zhou, Yue Pan, Jichao Wang, Ke Xu, Yafei Wen, Xiaohu Ruan, Xiaoxin Chen, and Honggang Qi. Skill-SD: Skill-conditioned self-distillation for multi-turn LLM agents. *arXiv preprint arXiv:2604.10674*, 2026.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. OfficeBench: Benchmarking language agents across multiple applications for office automation. *arXiv preprint arXiv:2407.19056*, 2024a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Jinyang Wu, Shuo Yang, Zhengxi Lu, Fan Zhang, Yuhao Shen, Lang Feng, Haoran Luo, Zheng Lian, Shuai Zhang, Zhengqi Wen, and Jianhua Tao. SEED: Self-evolving on-policy distillation for agentic reinforcement learning. *arXiv preprint arXiv:2607.14777*, 2026.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, et al. Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

Jiawei Xu, Arief Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Zheshen Wang, Peihao Wang, Pan Li, and Ying Ding. Rethinking the value of multi-agent workflow: A strong single agent baseline. *arXiv preprint arXiv:2601.12307*, 2026. URL https://arxiv.org/abs/2601.12307.

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 *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30424–30451, San Diego, California, United States, 2026. Association for Computational Linguistics. 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. In *International Conference on Learning Representations*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Woongyeng Yeo, Yumin Choi, Taekyung Ki, and Sung Ju Hwang. HINT-SD: Targeted hindsight self-distillation for long-horizon agents. *arXiv preprint arXiv:2605.17873*, 2026.

Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. Agent-R: Training language model agents to reflect via iterative self-training. *arXiv preprint arXiv:2501.11425*, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, 2022.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Ziqi Zhao, Xinyu Ma, Liu Yang, Yujie Feng, Daiting Shi, Jingzhou He, Xin Xin, Zhaochun Ren, and Xiao-Ming Wu. ROSD: Reflective on-policy self-distillation for language model reasoning across domains. *arXiv preprint arXiv:2605.28014*, 2026.

Shiding Zhu, Yudi Qi, Yajie Wang, Jiaze Li, Chao Song, Yaorui Shi, Yibo Miao, Hanqi Gao, and Kai Zhang. Escaping the self-confirmation trap: An execute-distill-verify paradigm for agentic experience learning. *arXiv preprint arXiv:2606.24428*, 2026.

This document provides the implementation details, task manifests, compiler contract, memory schemas, controlled results, and validation statistics referenced in the main paper. It accompanies the main paper as an anonymous appendix. All percentages are task success rates. Macro is the unweighted mean across OfficeBench, AppWorld, and τ-bench Retail.

A Reproducibility Protocol

A.1 Data Splits and Task Manifests

Table 5 specifies the manifests used for experience collection, model selection, and final evaluation. OfficeBench contains 148 training tasks and 152 held-out evaluation tasks. AppWorld uses the official 90-episode training split and the 168-episode `test_normal` split for the unified evaluation. Its 57-episode development protocol is reported only as an additional benchmark-specific comparison. τ-bench Retail uses all 74 official training tasks for the unified protocol and all 40 official test tasks for final evaluation. All learning and model-selection decisions use training-task interaction only.

Table 5: Task manifests under the unified controlled protocol.

Benchmark	Train	Test	Reported denominator	Split use
OfficeBench	148	152	152	Training interaction for learning and selection; held-out tasks for reporting
AppWorld	90	168	168	Training interaction for learning and selection; `test_normal` for reporting
τ-bench Retail	74	40	40	Official training tasks for learning and selection; official test for reporting

Task identifiers from evaluation splits are never provided to the seed collector, compiler, memory constructor, SFT builder, or model selection procedure. Every run validates split disjointness before collection and stores the ordered task manifest with its provenance record.

A.2 Benchmark Task Characteristics

The three benchmarks exercise complementary forms of procedural competence. OfficeBench evaluates multi-application office workflows with persistent artifacts and observable output contracts. AppWorld presents natural-language goals in a stateful application ecosystem and executes structured API calls. τ-bench Retail evaluates multi-turn customer service under an explicit domain policy and a policy-constrained tool interface.

Table 6: Task characteristics and environment-grounded success evidence.

Benchmark	Task setting	Agent actions	Success evidence
OfficeBench	Cross-application office workflows over persistent documents and application state	Controller delegations followed by application-grounded executor actions	Required output contract and final application state
AppWorld	Stateful application episodes defined by a natural-language user goal	Controller decomposition followed by schema-constrained application API calls	Official AppWorld evaluator over the resulting environment state
τ-bench Retail	Multi-turn retail support conversations governed by a domain policy	Policy-aware dialogue decisions and structured customer-service tool calls	Official task reward plus policy and state assertions

Together, the benchmarks cover artifact manipulation, API orchestration, and policy-constrained interaction. The compiler preserves each environment's native action interface and verifier while sharing one evidence schema and role-routing design.

A.3 Shared Model and Training Configuration

All generative roles use locally hosted copies of Qwen2.5 32B Instruct. The seed collector and frozen compiler use the same backbone as the controller and executor. No proprietary model, remote embedding service, external LLM, or stronger teacher is called. The two student roles use independent LoRA adapters initialized from the same backbone. Table 7 reports the training settings used by Full ProSEvo, Raw Replay SFT, and all trainable controlled variants.

Table 7: Shared model, serving, and adapter training configuration.

Setting	Value
Backbone	Qwen2.5 32B Instruct for collector, compiler, controller, and executor
Student parameters	Separate controller and executor LoRA adapters
Precision	BF16; gradient checkpointing enabled; model cache disabled during training
LoRA	Rank 16, scaling 32, dropout 0.05, no bias
Target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`
Optimization	AdamW, learning rate 1×10^{-5}, two epochs, weight decay 0
Batching	Per device batch size 1 and gradient accumulation 8
Maximum SFT length	4096 tokens for OfficeBench and 8192 for AppWorld and Retail
Loss	Completion token cross entropy; prompt tokens are masked
Serving	vLLM, tensor parallel size 4, context limit 16,384, BF16, GPU memory utilization target 0.90
Decoding	Temperature 0 for collection, compilation, controller, executor, and evaluation
Budget controls	Matched LoRA capacity, verified experience, optimizer updates, and supervised gradient tokens
External services	Zero external LLM API calls and zero remote embedding API calls

When a comparison would otherwise contain fewer examples, examples are cycled deterministically and the final microbatch is loss masked to match both the optimizer update count and the number of supervised gradient tokens. Every SFT prompt is memory-free. Parameter Only and Full ProSEvo therefore use matched adapter capacity, verified experience, optimizer updates, and supervised gradient-token budgets. Fixed-checkpoint interventions provide the separate test in which the student parameters are byte-identical.

A.4 Benchmark Specific Configuration

Table 8 gives the settings that differ by benchmark.

All benchmarks load all-MiniLM-L6-v2 locally and rank compatibility-filtered candidates by cosine similarity between query and memory embeddings, with the memory identifier as the final deterministic tie-breaker. The same retrieval configuration is used by all methods within a comparison. We selected $k = 3$ based on performance on the training tasks and fixed it before test evaluation.

A.5 Controlled Baseline Implementations

The suffix "style" in the main table is important. These are mechanism aligned implementations under a shared local backbone and interaction protocol, rather than copied published numbers. Table 9 states which defining mechanism is retained and which factors are standardized.

All baseline rollouts use the same benchmark task lists, tool interfaces, environment verifiers, decoding temperature, and interaction horizons. Their trainable variants use the same adapter capacity and matched optimization and supervised token budgets. Published results obtained with different backbones, task subsets, auxiliary models, or evaluation procedures are not inserted into the controlled table.

Table 8: Benchmark-specific rollout, retrieval, and evaluation settings.

Setting	OfficeBench	AppWorld	τ-bench Retail
Environment horizon	60 orchestration iterations; at most 8 actions per subtask	40 environment steps	15 environment steps
Update rounds	Three updates (Rounds 0–2); Base is pre-update	Three updates (Rounds 0–2)	Three updates (Rounds 0–2)
Randomness	Seed 42	Seed 42	Train seed 300; evaluation seed 101
Unified-protocol evaluation trials	One execution per task	One execution per episode	One execution per task
Verifier	OfficeBench task contract and final application state	Official AppWorld evaluator	Official Retail reward and policy assertions
Retrieval ranking	Local all-MiniLM-L6-v2 embeddings with cosine similarity	Local all-MiniLM-L6-v2 embeddings with cosine similarity	Local all-MiniLM-L6-v2 embeddings with cosine similarity
Compatibility filters	Memory type, consuming role, application, and task stage	Memory type, role, and application or API	Memory type, role, domain, and tool
Default depth	$k = 3$ for Structure, Execution, and Failure Memory	$k = 3$	$k = 3$
Additional threshold	Minimum cosine similarity 0.10	Minimum cosine similarity 0.10	Minimum cosine similarity 0.10

Table 9: Mechanism-aligned baselines under the unified local 32B protocol.

Method	Implemented mechanism	Shared controls
Base multi-agent	Hierarchical controller and executor without memory construction or parameter updating	Same backbone, tools, prompts, horizons, and evaluator
LEGOMem-style	Role-placed procedural memory retrieved by the orchestrating or specialized role; no student parameter update	The same local all-MiniLM-L6-v2 retrieval backend replaces proprietary model and embedding services
Skill-SD-style	Verified trajectories are converted into reusable skill guidance for student updating	Same backbone supplies guidance; no stronger teacher or external API
HinT-SD-style	Student visited states receive localized hindsight targets before parameter updating	Frozen local same-backbone copy supplies hindsight supervision
HERO-style	Later observations support localized reflection and self-distillation	Local same backbone supplies reflection; no external reflector
Raw-replay SFT	Verified successful traces are replayed directly without typed compilation or role-specific routing	Same accepted trajectories, adapter capacity, update count, and gradient-token budget

B Extended Method Specification

B.1 Formalization and Training Objective

This section gives the detailed construction omitted from Section 3, including cumulative evidence pools, role-specific memory partitions, parameter supervision, and the held-out evaluation objective.

Role-conditioned interaction. At orchestration step j, the planner–controller retrieves compatible procedural memory and produces either a delegation or a terminal decision:

$$m_j^{\mathrm{pc}} = \mathrm{Retrieve}_{\mathrm{pc}}(\mathcal{M}_r, h_j), \tag{9}$$

$$d_j \sim \pi_{\theta_r^{\mathrm{pc}}}\left(\cdot \mid h_j, m_j^{\mathrm{pc}}\right). \tag{10}$$

A delegation specifies the selected application or tool, the delegated subtask, and an observable completion condition.

Given delegation d_j, the executor retrieves compatible memory and produces one environment-grounded action:

$$m_t^{\mathrm{ex}} = \mathrm{Retrieve}_{\mathrm{ex}}(\mathcal{M}_r, h_t, d_j), \tag{11}$$

$$a_t \sim \pi_{\theta_r^{\mathrm{ex}}}\left(\cdot \mid h_t, d_j, m_t^{\mathrm{ex}}\right). \tag{12}$$

Both roles operate only on their decision-time visible context.

Closed-lineage collection and cumulative evidence. Let θ_{base} denote the pre-update student and define $\theta_{-1} \equiv \theta_{\mathrm{base}}$. Round 0 is collected by the frozen harnessed copy $\mathcal{G}_0$, while later rounds are collected by the preceding student as defined in Equation (5).

For collection round q, terminal verification produces

$$\mathcal{V}_q = \{\tau \in \mathcal{R}_q \mid V_{\mathcal{E}}(\tau) = 1\}. \tag{13}$$

For $r \in \{0, \ldots, R\}$, the cumulative verified trajectory pool and localized failure-evidence pool are

$$\begin{aligned} \mathcal{D}_r &= \mathrm{TraceDedup}\left(\bigcup_{q=0}^{r} \mathcal{V}_q\right), \\ \mathcal{F}_r &= \mathrm{EvidenceDedup}\left(\bigcup_{q=0}^{r} \mathrm{LocalFail}(\mathcal{R}_q)\right). \end{aligned} \tag{14}$$

Here, $\mathrm{TraceDedup}$ removes exact duplicates after deterministic normalization, while $\mathrm{LocalFail}$ denotes the source-linked localization and validation of failure evidence specified in the method. It does not introduce an additional generative model.

The cumulative construction satisfies

$$\mathcal{D}_{r-1} \subseteq \mathcal{D}_r, \qquad \mathcal{F}_{r-1} \subseteq \mathcal{F}_r, \tag{15}$$

so later students retain admitted historical evidence while contributing newly collected interaction.

Evidence and memory attribution. Let

$$\mathcal{Z} = \{\mathrm{str}, \mathrm{exe}, \mathrm{fail}\}, \qquad \mathcal{G} = \{\mathrm{pc}, \mathrm{ex}\}$$

denote the functional memory types and consuming roles. An evidence bundle e contains one or more referenced source events and their supporting context. Such a bundle may support a multi-step procedure, an executed action, or a localized failure. Let $\mathcal{E}_r^{\mathrm{src}}$ collect the source-grounded evidence bundles available through round r.

Write $\widetilde{\mathcal{M}}_r$ for validated and materialized memory records before memory-level deduplication. Each record m retains its source evidence $e(m)$, functional type $z(m)$, and consuming role $\Gamma(m)$. The accepted memory attribution is

$$\alpha_r : \mathcal{E}_r^{\mathrm{src}} \longrightarrow 2^{\mathcal{Z}\times\mathcal{G}}, \qquad \alpha_r(e) = \left\{(z(m), \Gamma(m)) : m \in \widetilde{\mathcal{M}}_r,\ e(m) = e\right\}. \tag{16}$$

An empty value means that no accepted memory artifact is constructed from that bundle. A source evidence bundle may support multiple records, but every materialized record has an explicit functional type and consumer. This mapping describes accepted artifacts; it is not an additional learned numerical credit estimator.

For each $(z, g) \in \mathcal{Z} \times \mathcal{G}$, define

$$\mathcal{E}_{r,\alpha}^{z,g} = \{(e, z, g) : e \in \mathcal{E}_r^{\mathrm{src}},\ (z, g) \in \alpha_r(e)\}\,. \tag{17}$$

The corresponding memory partitions are

$$\begin{aligned} \mathcal{M}_r^{z,g} &= \mathrm{MemoryDedup}\left(\left\{m \in \widetilde{\mathcal{M}}_r : (e(m), z(m), \Gamma(m)) \in \mathcal{E}_{r,\alpha}^{z,g}\right\}\right), \\ \mathcal{M}_r &= \bigcup_{(z,g)\in\mathcal{Z}\times\mathcal{G}} \mathcal{M}_r^{z,g}. \end{aligned} \tag{18}$$

This attribution governs the external-memory carrier. Parameter supervision is constructed separately from admitted positive source events and remains assigned to the role that produced the original output. Consequently, the same source evidence may contribute to both carriers without requiring the memory consumer $\Gamma(m)$ to equal the producing role g_t.

Role-specific parameter supervision. For role $g \in \mathcal{G}$, verified trajectories provide memory-free supervision

$$\mathcal{S}_r^g = \mathrm{RoleSFT}_g(\mathcal{D}_r) = \{(c_i^g, y_i^g)\}_{i=1}^{N_r^g}\,. \tag{19}$$

Here, c_i^g is the recorded decision-time input for role g, before any retrieved-memory block is appended, and y_i^g is the corresponding original output. An example is retained only after its trajectory and source event satisfy the respective admission criteria.

Controller targets preserve the original delegation, finish decision, or admitted customer-facing response when present in the source record. Executor targets preserve the original application action, API program, or tool call. The compiler provides labels and source references but does not rewrite these executable targets. Failure records themselves are not positive SFT targets; a later recovery action may qualify through its own positive admission.

For a target originating at event t, the recorded input is restricted to the decision-time context h_t. In particular, it excludes the observation produced by that target and all subsequent events. Retrieval-augmented text is also excluded from the recorded SFT input.

Optimization. The role-specific autoregressive objective is

$$\mathcal{L}_r^g(\theta) = -\frac{1}{N_r^g} \sum_{i=1}^{N_r^g} \sum_{j=1}^{|y_i^g|} \log p_\theta\left(y_{i,j}^g \mid c_i^g, y_{i,<j}^g\right). \tag{20}$$

For $g \in \mathcal{G}$ and $r \in \{0, \ldots, R\}$, the cumulative parameter update is

$$\theta_r^g = \mathrm{Train}\left(\theta_{r-1}^g; \mathcal{L}_r^g\right), \tag{21}$$

with $\theta_{-1} = \theta_{\mathrm{base}}$. Each role adapter is therefore warm-started from the preceding student and updated using admitted cumulative role supervision.

Training–evaluation separation and objective. All learned artifacts are constructed exclusively from training interactions:

$$\mathcal{X}_{\mathrm{tr}} \cap \mathcal{X}_{\mathrm{ev}} = \varnothing. \tag{22}$$

After the final update, deployment retains $\mathcal{A}_R = (\theta_R, \mathcal{M}_R)$. Its task-level utility on held-out evaluation tasks is

$$\mathcal{J}(\mathcal{A}_R) = \frac{1}{|\mathcal{X}_{\mathrm{ev}}|} \sum_{x \in \mathcal{X}_{\mathrm{ev}}} \mathbb{E}_{\tau \sim \mathcal{A}_R(\cdot|x,\mathcal{E})}\left[V_{\mathcal{E}}(\tau)\right]. \tag{23}$$

The learning problem is therefore to construct a sequence $\mathcal{A}_0, \ldots, \mathcal{A}_R$ in which successive students contribute verified experience to their successors while typed external memory and role-specific parameters evolve from cumulative training interaction. Cumulative evidence retention does not itself imply monotonic performance; improvement is assessed only on disjoint held-out evaluation tasks.

B.2 FROZEN COMPILER CONTRACT

The compiler is an offline, frozen copy of the same Qwen2.5 32B Instruct backbone. It does not act in the environment and is removed at deployment. The implemented flow is

> completed trace → deterministic projection and ordering → frozen compiler at temperature 0 → strict JSON parse → schema and evidence validation → accepted or rejected extraction → deterministic typed materialization.

The three benchmark adapters share the same output contract and evidence semantics. Their versioned prompt identifiers are `officebench-frozen-teacher-v2`, `appworld-frozen-teacher-v2`, and `tau2-frozen-teacher-v2`. The instruction requires the compiler to use only the supplied task, capabilities or policy, events, observations, and official outcome. It may not invent or rewrite actions, arguments, responses, or step identifiers. Final success does not imply that every intermediate step is positive. Omitted events are neutral. Structure support may reference only positive steps. Every negative step requires exactly one Failure Memory, and a nonnull recovery identifier must reference a later positive step; a null identifier retains correction guidance without claiming an observed recovery. The response must be one JSON object without prose, Markdown, or code fences.

B.2.1 EXACT COMPILER SYSTEM PROMPTS

The following blocks reproduce the system prompts used by the reported compiler. The legacy internal compiler identifier NeoMem is retained verbatim because it is part of the frozen prompt text for ProSEvo. No prompt is rewritten after a rollout has been observed.

Input projection. Events receive unique step identifiers and are sorted by integer event index. The compact UTF 8 JSON input uses the fields in Table 10. Truncation is deterministic and applied before the prompt is constructed.

Table 10: Benchmark-specific compiler inputs and deterministic projections.

Benchmark	Top level input	Projection rules
OfficeBench	case identifier, task, required outputs, runtime capabilities, events, and official success	Credentials become `<secret>`; controller capability is limited to 6000 characters; each observation keeps its final 3000 characters; outcome reason keeps its final 1500 characters; execution and ordering flags are retained
AppWorld	task identifier, task, runtime, events, and official success	Application and API capabilities are retained; each executor observation keeps its final 3000 characters; accepted actions and actual API calls remain linked to the source event
τ-bench	task identifier, domain, policy and tools, messages, events, official reward, and success	System messages, timestamps, usage fields, and raw tool payloads are removed; each executor observation keeps its final 3000 characters; visible dialogue and domain policy are retained

Exact request construction. The initial user message is not a natural language summary. It is the projected payload serialized as compact UTF 8 JSON with `ensure_ascii=False` and comma and colon separators. The request therefore has the following form:

```
system = exact benchmark compiler prompt above
user = compact_json(projected_trace)
temperature = 0
```

OfficeBench uses the fields `case_id`, `task`, `required_outputs`, `runtime_capabilities`, `events`, and `official_success`. AppWorld uses

Listing 1: Exact OfficeBench compiler system prompt.

```
You are the frozen NeoMem Experience Compiler for one
completed OfficeBench trace. Use only the supplied task, required
    outputs, runtime
capabilities, events, observations, and official outcome. Official
    success does not
make every step correct. Do not invent or rewrite actions, parameters,
    prompts,
responses, or step IDs.

Return exactly one strict JSON object with no prose, Markdown, or code
    fence:
{
  "steps":[{"step_id":"C1|E1|D1","label":"positive|negative|neutral","
    reason":"..."}],
  "structure_memory":{"summary":"...","supporting_step_ids":["..."]},
  "failure_memories":[{
    "failed_step_id":"...",
    "cause":"...",
    "recovery":"...",
    "recovered_by_step_id":"OPTIONAL_LATER_POSITIVE_STEP"
  }]
}

Step labels:
- positive: a real observation shows that the step correctly advanced
    the task;
- negative: technical error, wrong file/entity/parameter/output target,
    rejected
  action, redundant work, premature completion, unsupported claim, or
    contradiction;
- neutral: the trace does not establish either conclusion. Omitted steps
     become
  neutral deterministically after parsing.

Structure Memory is optional (use null when unsupported). Its summary
    must contain
only a reusable high-level procedure and every supporting_step_id must
    be positive.
Do not output Execution Memory text: after validation, each positive,
    actually executed
Executor event is deterministically projected from its original action
    and observation.

Every negative Controller or Executor step must have exactly one
    failure_memories
entry. Anchor cause to failed_step_id. recovery must be supported by the
     observed
error, runtime action schema, required outputs, or a real later positive
     step. Set
recovered_by_step_id only when that cited step is positive, occurs later
    , and both step
orders are reliable; otherwise omit it or use null. Do not claim that an
     uncited or
earlier action recovered the failure.
```

`task_id`, `task`, `runtime`, `events`, and `official_success`. Retail uses `task_id`, `domain`, `domain_policy_and_tools`, `messages`, `events`, `official_reward`, and `official_success`. Each event includes its step identifier, integer event index, role, and original payload. Executor events additionally retain their delegation and observation. AppWorld

Listing 2: Exact AppWorld compiler system prompt.

```
You are the frozen NeoMem Experience Compiler for one
completed AppWorld trace. Use only the supplied task, runtime
    capabilities, events,
observations, and official outcome. Final success does not make every
    event correct.
Do not invent or rewrite actions, code, parameters, responses, or step
    IDs.

Return exactly one strict JSON object with no prose, Markdown, or code
    fence:
{
  "steps":[{"step_id":"C1|E1","label":"positive|negative|neutral","
    reason":"..."}],
  "structure_memory":{"summary":"...","supporting_step_ids":["..."]},
  "failure_memories":[{
    "failed_step_id":"...",
    "cause":"...",
    "recovery":"...",
    "recovered_by_step_id":"OPTIONAL_LATER_POSITIVE_STEP"
  }]
}

Step labels:
- positive: the event is supported by a real observation and advances
    the task;
- negative: technical error, wrong entity/parameter, policy violation,
    unsupported
  completion, redundant action, or contradiction;
- neutral: the trace does not establish either conclusion. Omitted
    events become
  neutral deterministically after parsing.

Structure Memory is optional (use null when unsupported). Its summary
    must describe
only a reusable high-level procedure and every supporting_step_id must
    be positive.
Do not output Execution Memory text: after validation, each positive,
    actually accepted
Executor event is deterministically projected from its original action
    and observation.

Every negative Controller or Executor step must have exactly one
    failure_memories
entry. Anchor cause to failed_step_id. recovery must be a correction
    supported by the
observed error, runtime schema, or a real later positive step. Set
    recovered_by_step_id
only when that cited step is positive and occurs later; otherwise omit
    it or use null.
Do not claim that an uncited or earlier action recovered the failure.
```

also retains the actual API calls, while OfficeBench retains action execution, status, outcome, and ordering fields.

Exact serialization retry. Only a JSON parsing failure triggers a second model call. AppWorld and Retail append the following text to the unchanged serialized trace:

```
Previous output was not strict JSON. Return one JSON object
with no prose or code fence.
```

Listing 3: Exact τ-bench Retail compiler system prompt.

```
You are the frozen NeoMem Experience Compiler for one
completed tau2 trace. Follow the supplied domain policy and use only the
    supplied
messages, events, observations, and official outcome. Final reward 1
    does not make
every event correct. Do not invent or rewrite tool calls, arguments,
    customer
responses, or step IDs.

Return exactly one strict JSON object with no prose, Markdown, or code
    fence:
{
  "steps":[{"step_id":"C1|E1","label":"positive|negative|neutral","
    reason":"..."}],
  "structure_memory":{"summary":"...","supporting_step_ids":["..."]},
  "failure_memories":[{
    "failed_step_id":"...",
    "cause":"...",
    "recovery":"...",
    "recovered_by_step_id":"OPTIONAL_LATER_POSITIVE_STEP"
  }]
}

Step labels:
- positive: policy-compliant evidence shows that the event advanced the
    user task;
- negative: technical error, wrong entity/parameter, policy violation,
    unsupported
  claim, redundant action, or contradiction;
- neutral: the trace does not establish either conclusion. Omitted
    events become
  neutral deterministically after parsing.

Structure Memory is optional (use null when unsupported). Its summary
    must contain
only a reusable high-level procedure and every supporting_step_id must
    be positive.
Do not output Execution Memory text: after validation, each positive,
    actually accepted
Executor tool call is deterministically projected from its original name
    and arguments.

Every negative Controller or Executor step must have exactly one
    failure_memories
entry. Anchor cause to failed_step_id. recovery must be supported by the
    observed
error, supplied domain policy/tool schema, or a real later positive step
    . Set
recovered_by_step_id only when that cited step is positive and occurs
    later; otherwise
omit it or use null. Do not claim that an uncited or earlier action
    recovered failure.
```

OfficeBench sends the same compiler system prompt with a repair user message containing the unchanged source case, the invalid response, and the parser message:

```
Your previous response violated only the serialization
contract.  Return one strict JSON object with no prose,
Markdown, or code fence.  The previous response was invalid:
[parser message].  Source case:  [unchanged compact JSON].
Invalid response:  [previous response].
```

The retry does not contain schema or evidence validation feedback. An output that parses as JSON but fails any rule in Table 11 is rejected immediately.

Raw output schema. The root permits exactly three keys. No unknown fields are accepted. The machine readable schema is equivalent to the following abbreviated form.

```
{
  "steps": [
    {"step_id": string,
     "label": "positive" | "negative" | "neutral",
     "reason": string}
  ],
  "structure_memory": null | {
    "summary": string,
    "supporting_step_ids": [string]
  },
  "failure_memories": [{
    "failed_step_id": string,
    "cause": string,
    "recovery": string,
    "recovered_by_step_id": string | null
  }]
}
```

Parsing and retry. The returned text is stripped and passed once to a JSON parser. The root must be an object. The implementation does not search explanatory text or code fences for a JSON substring. A serialization failure permits one format only retry that repeats the strict JSON requirement without changing the trace or providing validator feedback. A second parse failure is rejected. Outputs that parse but violate schema or evidence constraints are rejected without retry.

Validation. Table 11 lists the rejection rules. Rejected outputs are retained for audit but generate no memory record and no SFT example.

Table 11: Compiler validation and rejection rules.

Rule	Rejection condition
PARSE_NOT_JSON, ROOT_NOT_OBJECT	Output is not one strict JSON object
OUTPUT_KEYS	A required root key is missing or an unknown key appears
STEP_SHAPE, STEP_REFERENCE, STEP_LABEL	Malformed step, unknown or repeated identifier, invalid label, or empty reason
POSITIVE_SFT_EVIDENCE	A positive step lacks its original memory-free prompt or payload
POSITIVE_EXECUTION_EVIDENCE	A positive executor action was not executed or accepted by the environment
STRUCTURE_SHAPE, STRUCTURE_SUPPORT	Empty summary, duplicate support, or support from a nonpositive or unknown step
FAILURE_SHAPE, FAILURE_SOURCE	Empty cause or recovery, duplicate failed step, or failure not anchored to a negative step
RECOVERY_ORDER	Recovery is not a later positive step or event ordering is unreliable
FAILURE_COVERAGE	Negative steps and Failure Memory entries are not in one to one correspondence
SCHEMA_VERSION, PROMPT_VERSION	Artifact uses an incompatible schema or prompt version
Identity rules	Benchmark, task identifier, case identifier, or source hash differs from the input trace

B.3 Typed Memory Schema and Materialization

The frozen compiler labels evidence and writes the Structure summary and Failure cause and recovery fields. It does not generate executable action text. Execution Memory is projected deterministically from an action that was labeled positive and was actually executed or accepted by the environment. This distinction prevents invented calls or arguments from entering memory.

Table 12: Typed procedural memory records.

Type	Retrieval key	Stored content	Consumer
Structure	Task family, persistent state, completed stage	Application flow, ordered stages, delegation conditions, observable completion checks, fallback conditions, evidence, and provenance	Controller
Execution	Delegated subtask, application or tool, current observation	Executed action intent, compatible signature, grounded argument constraints, expected observation, evidence, and provenance	Executor
Failure	Application or tool, error signal, current stage	Unsupported pattern, observed error, correction, supported recovery condition, evidence, and provenance	Controller for decomposition or completion failures; executor for tool or argument failures

Structure Memory is constructed only from an officially successful trace with nonempty positive supporting steps. Benchmark specific abstraction replaces transient entities without changing the stage order. Execution Memory is constructed only from an officially successful trace and a positive executor step whose original action was executed or accepted. Tool name, application, arguments, and confirming observation come from the trace. Failure Memory is constructed for each negative controller or executor step. If a valid later positive recovery exists, its source action is linked. Otherwise, the record retains evidence grounded correction guidance with a null recovery identifier. Failure records associated with task decomposition or completion checks are routed to the controller, whereas records associated with tool selection or arguments are routed to the executor. Records that satisfy neither routing condition are rejected.

B.4 Prompt Inventory and Reproducibility Boundary

Table 13 enumerates every generative prompt family in the reported ProSEvo pipeline. Compiler prompts are reproduced verbatim above. The remaining prompts are specified below as literal instructions plus their deterministic, environment-derived fields. Date, user, application, tool-schema, task-state, file-inventory, and conversation fields are instance variables rather than model-generated text.

Dynamic capability blocks are obtained from the active benchmark runtime. OfficeBench renders available application actions and the official output contract, AppWorld renders the available application APIs, and Retail renders the domain policy and selected tool schema. Consequently, a single static block would represent these instance-dependent prompts less faithfully than specifying each field's source and placement. The implementation stores the expanded prompt with each model call, enabling instance-level audit.

B.5 Bootstrap Collection Harness Prompt

Round 0 uses a frozen, harnessed copy of the base student. The collection planner receives the following system instruction. Bracketed fields are deterministically substituted from the benchmark instance.

```
Today is [DATE] ([WEEKDAY]). The current time is [TIME].
You are the privileged workflow planner assisting user [USER]
```

Table 13: Generative prompt inventory for ProSEvo.

Stage	Prompt family	Model input	Persisted output
Bootstrap collection	Harness planner	Task, runtime applications and capabilities, candidate typed memories, fixed constraints, and harness schema	Stage-oriented workflow and role-specific memory placement
Environment interaction	Controller	Runtime capabilities or policy, task or visible transcript, state, completion evidence, and optional compatible memory	Delegation, completion check, response, or finish decision
Environment interaction	Executor	Selected application or tool schema, delegation, completion check, state, and optional compatible memory	One application action, API program, or tool call
AppWorld admission	Step and pending-write verifiers	Original task, delegation, executed action, direct or later read-only observation	Positive, negative, or neutral evidence label with reason
AppWorld recovery	Failure diagnosis	Original task and active execution chain	Earliest supported cause and replanning constraints
Offline compilation	Benchmark compiler	Deterministically projected completed trace and official outcome	Step labels, optional Structure Memory summary, and Failure Memory entries
Parameter update	Role SFT	Stored system instruction and original memory-free role prompt	Original accepted controller or executor payload

```
in a multi-agent office automation system.
First retrieve procedural experience, then synthesize a high-level
    harness.
You must output a workflow and a role-aware memory placement plan.
Plan using only explicitly available apps and capabilities.
Keep simple single-app tasks direct and use the minimum necessary stages
    .
Never invent input file names or paths. If an input name is unknown,
    plan
shell file discovery first.
For bulk spreadsheet edits, plan a precise shell/openpyxl transformation
after discovering the real file path.
For multiple email recipients or calendar users, plan separate one-user
actions and verify every named person is completed.
Treat paths in memories as examples only and preserve exact required
    output
paths. Use only candidate memory ids shown in the prompt.
Output exactly one JSON object and nothing else.
Required schema: [HARNESS_SCHEMA]
```

The user message is assembled in the following fixed order:

```
Task: [TRAINING_TASK]

Available apps: [APPLICATION_LIST]

Available app capabilities: [RUNTIME_CAPABILITIES]

Candidate structure memories: [TOP_COMPATIBLE_STRUCTURE_RECORDS]

Candidate execution memories for app=[APP]:
[TOP_COMPATIBLE_EXECUTION_RECORDS]

Candidate failure memories for app=[APP]:
[TOP_COMPATIBLE_FAILURE_RECORDS]
```

```
Constraints:
- Select at most the configured number of records for each role and type
    .
- Keep the workflow compact and stage-oriented.
- Discover unknown paths before using them.
- Preserve exact required outputs and treat memory entities as examples.
```

The required harness object is

```
{
  "workflow_summary": "short summary",
  "stages": [{
    "stage_id": "stage_1",
    "goal": "what this stage should achieve",
    "preferred_app": "APP_OR_EMPTY",
    "success_check": "what success looks like",
    "fallback": "what to do if progress fails"
  }],
  "placement": {
    "orchestrator": {
      "structure_memory_ids": ["memory_id"],
      "guidance": "high-level orchestration guidance"
    },
    "app_agents": {
      "APP": {
        "execution_memory_ids": ["memory_id"],
        "failure_memory_ids": ["failure_id"],
        "guidance": "local application guidance"
      }
    }
  }
}
```

The harness then exposes its workflow, completion checks, bounded fallback, and role-assigned memories to the collection controller and executor. Later rounds use the deployed student prompts described next.

B.6 Controller and Executor System Prompts

OfficeBench deployment roles. The controller system instruction is the following parameterized literal template:

```
Today is [DATE] ([WEEKDAY]). The current time is [TIME].
You are the student planner/controller assisting user [USER].
Choose one next app delegation or finish. Do not output advance_stage.
Never invent file names or paths; delegate shell discovery when the real
     path
is unknown. Use only tools already available in the environment. Do not
install packages. Do not finish until the final answer or every required
output is locally verified. Retrieved memories are optional,
non-authoritative patterns; current task/state always wins.
Return exactly one JSON object and nothing else.
{"delegate_to":"APP_NAME","subtask":"SUBTASK_DESCRIPTION",
 "completion_check":"OBSERVABLE_SUCCESS_CONDITION"}
{"finish":"FINAL_ANSWER","reason":"WHY_FINISH_IS_ALLOWED"}
```

The executor system instruction is:

```
Today is [DATE] ([WEEKDAY]). The current time is [TIME].
You are the student app agent for '[APP]', assisting user [USER].
Generate one valid JSON action with exactly "app": "[APP]".
```

```
Use current state and confirmed /testbed paths as the source of truth.
Never install packages, invent file names, or substitute output paths.
Retrieved memories are optional action patterns, never literal arguments
    .
Return exactly one JSON object and nothing else.
Current case runtime capabilities: [RUNTIME_ACTION_SCHEMA]
```

The memory-free controller and executor user messages use the exact field order in Table 14.

AppWorld deployment roles. The literal system instructions are:

```
You are the Controller in a two-layer AppWorld agent.
Choose one app-specific delegation at a time, or finish only when the
    task is
complete. Runtime capabilities and the current task state are
    authoritative.
Retrieved memories are optional non-authoritative references. Return
    exactly
one JSON object:
{"delegate_to":"APP","subtask":"GOAL","completion_check":"CHECK"}
or {"finish":"CONCISE_ANSWER_OR_None"}.
```

```
You are an AppWorld Executor for exactly one delegated app.
Return exactly one JSON object: {"app":"APP","code":"PYTHON"}.
The code is executed by AppWorld and may use only the selected `apis.APP
    `
namespace. Print useful observations. Do not call supervisor.
    complete_task;
only the Controller finishes.
```

τ-bench Retail deployment roles. The literal system instructions are:

```
You are the Controller in a two-layer customer-service agent.
The domain policy, risk labels, conversation, and actual tool results
    are
authoritative. Before delegating, check every policy prerequisite,
identity-verification requirement, required disclosure, and user-
    confirmation
requirement. If any prerequisite is missing, respond to the customer
    instead
of delegating. At each turn either delegate one tool call to Executor or
respond to the customer. Retrieved memories are optional and non-
    authoritative.
Return exactly one JSON object:
{"delegate_to":"TOOL","subtask":"GOAL","completion_check":"CHECK"}
or {"respond":"CUSTOMER_MESSAGE"}.
```

```
You are the Executor for one selected tau2 tool.
Use the supplied function schema and return exactly one JSON object:
{"tool_name":"TOOL","arguments":{...}}.
Do not send customer-facing text and do not select any other tool. The
Controller's target and restrictions are hard constraints. Instance
    values
such as order IDs, customer IDs, emails, phone numbers, and dates must
    come
from the visible conversation or a real tool result; never invent or
substitute them.
```

B.7 AppWorld Verification and Diagnosis Prompts

AppWorld uses three auxiliary prompts during collection. They do not produce positive SFT targets directly; their outputs supply evidence for admission or localized recovery.

```
You verify one completed AppWorld Executor step. Use only the original
    task,
delegation, executed action, and observation. Mark positive only when
    the
observation gives concrete evidence that the action correctly advanced
    the
delegated subtask and its completion check. Mark negative for errors,
    wrong or
unrelated actions, policy violations, or contradicted completion checks.
     Use
neutral when evidence is insufficient. Return exactly:
{"outcome":"positive|negative|neutral","reason":"CONCISE_EVIDENCE"}.
```

```
You verify one pending AppWorld write using a later read-only
    observation.
Mark positive only when the read proves the exact pending write
    correctly
achieved its delegated subtask. Mark negative when it proves the write
    used a
wrong entity, value, order, or violated the task. Use neutral when it
    still
cannot be verified. Return exactly:
{"outcome":"positive|negative|neutral","reason":"CONCISE_EVIDENCE"}.
```

```
You diagnose one semantic AppWorld failure. Review the original task and
active execution chain, identify the earliest supported root cause, and
provide constraints for replanning. Do not generate API code or a new
    tool
call. Return exactly:
{"error_type":"wrong_entity|missed_constraint|wrong_order|
    duplicate_action|
 unsupported_inference|wrong_api_or_parameters|other",
 "root_cause_step":"EXISTING_STEP_ID","evidence":"CONCISE_EVIDENCE",
 "correction_constraints":["ONE_TO_FOUR_SHORT_CONSTRAINTS"]}.
```

B.8 Controller, Executor, Retrieval, and SFT Construction

The compiler prompts above are distinct from the prompts used to act in the environment. Table 14 gives the exact field order used to construct the role inputs. Benchmark specific capability blocks contain the currently available applications, APIs, tools, action schemas, or domain policy. They are generated from the environment rather than from a remote model.

For AppWorld, the memory-free controller input is the runtime controller capability block, `Task`, and the JSON `Current state`. Its executor input is the application API schema, `Delegated subtask`, `Completion check`, `Original task`, and the same state. For Retail, the state is represented as `Visible conversation and tool results` plus `Last tool error`. OfficeBench uses the corresponding task, output requirements, file inventory, state summary, recent execution history, and last error fields. These strings are stored before the optional retrieval block is appended.

The three benchmark adapters instantiate the following two canonical memory-free layouts, omitting only fields unavailable in the active environment:

```
Controller user message
[RUNTIME_CAPABILITIES_OR_POLICY]
Task or visible conversation: [TASK_STATE]
Official success contract: [OUTPUT_REQUIREMENTS]
```

Table 14: Role-prompt construction and inference-time memory placement.

Role	Memory free prompt recorded as `prompt_no_memory`	Inference only addition
Controller	Runtime capability or policy block; original task or visible conversation; persistent state or recent events; last error; OfficeBench output requirements and file inventory when applicable	Header `Optional retrieved references:`, followed by `Task procedure pattern` from Structure Memory and `Potential mistake` from compatible Failure Memory
Executor	Application or tool schema; delegated subtask; completion check; original task or visible conversation; persistent state or recent events; last error	Header `Optional retrieved references:`, followed by `Same-app execution pattern` or `Same-tool execution pattern` from Execution Memory and `Error recovery` from compatible Failure Memory

```
File inventory / persistent state / recent events: [STATE_EVIDENCE]
Last error: [LAST_ERROR_OR_NONE]
Return one delegation with a completion check, or a verified finish/
    response.

Executor user message
[SELECTED_APPLICATION_OR_TOOL_SCHEMA]
Delegated subtask: [SUBTASK]
Completion check: [CHECK]
Original task or visible conversation: [TASK_STATE]
File inventory / persistent state / recent events: [STATE_EVIDENCE]
Last error: [LAST_ERROR_OR_NONE]
Return one action for the selected application or tool.
```

For Retail, `TASK_STATE` is a deterministic visible transcript that retains the initial user goal, feasible user constraints, and recent tool results within the prompt budget. Provider metadata, timestamps, usage fields, and system messages are excluded. AppWorld uses a JSON state snapshot of the six most recent events plus pending-write or correction constraints when present. OfficeBench retains its official required outputs, testbed inventory, persistent state summary, recent raw events, and last error.

Exact retrieval placement and visible serialization. Retrieval is appended only at inference. OfficeBench prepends `Optional retrieved references (non-authoritative; ignore any literal argument that is not confirmed by the current task/state):` and uses the titles `Task procedure patterns`, `Potential mistakes to avoid`, `Same-task execution patterns`, and `Error-specific recovery patterns`. AppWorld and Retail append `Optional retrieved references:` and use controller titles `Task procedure pattern` and `Potential mistake`, plus executor titles `Same-app execution pattern` or `Same-tool execution pattern` and `Error recovery`.

Visible records are rendered from the compiled fields rather than hidden model state. A Structure record exposes its reusable summary or procedural note. An Execution record exposes its accepted action template, grounded argument template, and confirming observation. A Failure record exposes the failed pattern, observation, and supported recovery. The Raw-Trajectory control replaces these fields with the bounded temporal JSON trace. Each retrieval event stores returned and used memory identifiers, visible character and token counts, and a hash of the rendered block for audit.

The retrieval queries also follow a fixed role boundary. Controllers query with the task plus current state or visible transcript; a last error is added for Failure Memory. Executors additionally include the delegated subtask and selected application or tool. Candidates pass memory-type, role, and application or tool compatibility filters before semantic ranking.

Each accepted role example is serialized as

```
{
  "system": "ROLE_SYSTEM_INSTRUCTION",
  "prompt": "MEMORY_FREE_ROLE_PROMPT",
  "completion": "JSON_ACCEPTED_ROLE_PAYLOAD",
  "meta": {
    "trace_id": "SOURCE_TASK_ID",
    "role": "controller" | "executor",
    "step": "POSITIVE_STEP_ID"
  }
}
```

Only admitted positive steps from an officially successful trace enter role SFT. Each target preserves the original role payload. Controller outputs follow the benchmark-specific delegation or finish schema and preserve a customer-response payload only when one is present in the admitted source record; executor outputs preserve the accepted application action, API code, or tool call. The builder pairs a target with the role prompt stored before the retrieval block was appended and rejects inputs containing that block's identifiers or serialized content. This check establishes retrieval exclusion. Temporal validity additionally requires that every retained input field and derived view use only information available at the corresponding decision time.

This construction makes the carrier separation executable: the adapter learns from the original memory-free decision context, whereas typed records remain an inference-time input whose presence can be intervened on at a fixed checkpoint.

B.9 Audited OfficeBench Case: Task 1-11/0

Official task and evaluation contract. OfficeBench task 1-11/0 asks the agent to sort `/testbed/data/score.xlsx` by `midterm1` in descending order and save the result as `/testbed/data/sorted_score.xlsx`. The official configuration applies `evaluate_exact_match` to the generated and reference workbooks. The evaluator compares both worksheets cell by cell, so success requires the complete output in Table 15, including the header and all nonsort columns.

Table 15: Official input and reference ordering for OfficeBench task 1-11/0.

Position	Name	midterm1	midterm2
Input 1	Liam	74	72
Input 2	Ivy	64	90
Input 3	Alice	78	75
Input 4	Jack	71	73
Output 1	Alice	78	75
Output 2	Liam	74	72
Output 3	Jack	71	73
Output 4	Ivy	64	90

Verified source trajectory. The stored full-task record identifies this case as a harness-assisted bootstrap trajectory and records an official evaluator pass. Its six actions comprise three spreadsheet reads followed by directory inspection, output-file initialization, and an OpenPyXL transformation. The final command loads the source worksheet, sorts its data rows by the second column in reverse order, reconstructs the original three-column header, and writes the required output:

```
wb = openpyxl.load_workbook('/testbed/data/score.xlsx')
sheet = wb.active
rows = sorted(sheet.iter_rows(min_row=2, values_only=True),
              key=lambda row: row[1], reverse=True)
out = openpyxl.Workbook()
out.active.append(['Name', 'midterm1', 'midterm2'])
for row in rows:
```

```
    out.active.append(row)
out.save('/testbed/data/sorted_score.xlsx')
```

The stored action sequence is summarized in Table 16. The archived observation field contains the execution status value `0`; consequently, this audit attributes semantic claims to the recorded actions and official evaluator rather than to unrecorded natural-language observations.

Table 16: Stored action sequence for the officially verified 1-11/0 bootstrap trajectory.

Step	App	Recorded action
1	Excel	Read `/testbed/data/score.xlsx`.
2–3	Excel	Read `/testbed/data/sorted_score.xlsx`.
4	Shell	List the contents of `/testbed/data`.
5	Shell	Copy `score.xlsx` to the required output path.
6	Shell	Sort the source rows with OpenPyXL and overwrite the output with the required ordering.

Compiled-artifact audit. Table 17 reports the artifacts materialized for this case in the audited compiler outputs. The Structure Memory preserves the spreadsheet-to-shell workflow and the distinction between source reads and required output writes. The two admitted Execution Memories preserve source inspection and path discovery. The recorded Failure Memory contains four records associated with the harness-assisted trajectory and four records from a separate unsuccessful student rollout of the same task. These records cover stage-control prompts, a repeated controller decision, invalid structured output, and a repeated shell action.

Table 17: Audited compiler outputs for OfficeBench task 1-11/0. Counts and contents are copied from the audited JSONL records.

Artifact	Count	Audited content	Consumer
Structure Memory	1	Six-stage Excel–shell workflow with completion checks, path discovery, required-output creation, and fallback guidance	Planner–controller
Execution Memory	2	`excel::read_file(score.xlsx)` and `shell::command(ls /testbed/data)`	Executor
Failure Memory	8	Four harnessed-trace records and four records from a separate unsuccessful student rollout	Controller, executor, or general retrieval

This case establishes the correspondence among the official task contract, the verified source trajectory, and the materialized JSONL records. The decisive OpenPyXL transformation remains available in the stored source trajectory but is not materialized as an Execution Memory. The audited artifacts also do not contain case-linked controller/executor SFT rows or a fixed-checkpoint retrieval log for 1-11/0. Accordingly, this case audits collection and compilation rather than a retrieval-mediated held-out outcome.

C Additional Experimental Results

C.1 Original-Protocol Three-Round Results

Table 18 records the complete three-round trajectory under each benchmark's original reporting protocol. OfficeBench uses one run over 152 tasks. AppWorld reports Avg@4 and Best@4 over 168 episodes. τ-bench Retail follows the HERO-aligned TauBench-Retail split with 500 training tasks and a disjoint 115-task test manifest; four trials per test task produce mean@4 success and average turns. This original-protocol split is distinct from the official 74/40 Retail split in Table 5. Original-protocol results provide within-benchmark trajectories, while the unified results supply model selection, controlled comparisons, and cross-benchmark macro averages.

Table 18: Three-round ProSEvo results under the original benchmark protocols. All success values are percentages.

	OfficeBench		AppWorld			τ-bench Retail		
Round	Train	Overall	Train	Avg@4	Best@4	Train	mean@4	Turns
Round 0	27.70	25.00	22.22	14.43	25.00	44.20	36.09	11.4
Round 1	29.73	27.63	26.67	16.22	27.98	47.40	41.30	10.7
Round 2	34.46	30.92	30.00	17.86	29.76	51.20	46.09	10.0

Cross-protocol consistency. All three benchmark-specific protocols show successive gains from Round 0 through Round 2. The cumulative Round 0-to-Round 2 improvements reach 5.92 pp on OfficeBench, 3.43 pp on AppWorld, and 10.00 pp on Retail, matching the direction of the unified evaluation. Because these protocols use different reporting metrics, cross-benchmark Macro values in the main paper are computed only under the unified protocol.

Table 19: Round-wise improvements under benchmark-specific protocols. Office uses Overall SR, AppWorld uses Avg@4, and Retail uses mean@4 SR. Values are within-benchmark gains (pp).

Update contrast	**Office**	**AppWorld**	**Retail**
Round 1 − Round 0	2.63	1.79	5.21
Round 2 − Round 1	3.29	1.64	4.79
Round 2 − Round 0	**5.92**	**3.43**	**10.00**

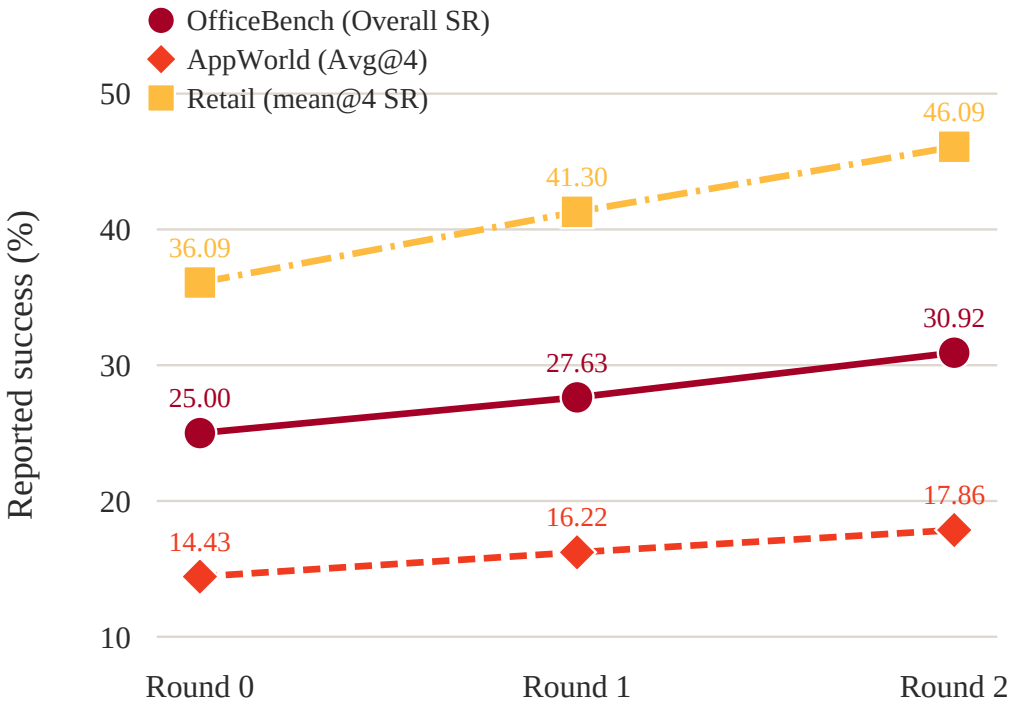


Figure 4: Success improves after both cumulative updates under every benchmark-specific protocol. OfficeBench reports Overall SR, AppWorld reports Avg@4, and Retail reports mean@4 SR; each curve is interpreted within its own protocol.

C.2 Fixed Checkpoint Memory Interventions

Table 20 reports the benchmark-level values underlying the fixed-checkpoint memory intervention in the main paper. The student checkpoint, task order, environment horizon, and decoding settings remain fixed. Only the memory supplied at inference changes. Raw trajectory uses a verified temporal trace instead of a compiled record. Role-Swapped exchanges Structure and Execution Memory between consumers. The Length-Matched Shuffle control uses content from another task with the same type and exactly the same visible token unit count.

Table 20: Fixed-checkpoint memory interventions. Parentheses report successful evaluation units.

Condition	OfficeBench	AppWorld	Retail	Macro
Correct compiled	**30.92 (47/152)**	**21.43 (36/168)**	**35.00 (14/40)**	**29.12**
Raw trajectory	28.29 (43/152)	19.05 (32/168)	32.50 (13/40)	26.61
No Memory	27.63 (42/152)	18.45 (31/168)	30.00 (12/40)	25.36
Role-Swapped	21.71 (33/152)	15.48 (26/168)	22.50 (9/40)	19.90
Length-Matched Shuffle	16.45 (25/152)	11.31 (19/168)	15.00 (6/40)	14.25

Correct compilation is 2.51 Macro points above raw trajectory retrieval and 3.76 points above No Memory. Correct role assignment adds 9.22 Macro points over Role-Swapped, while task-matched content adds 14.86 points over the Length-Matched Shuffle. The benchmark-level changes from No Memory are $+3.29/+2.98/+5.00$ points for Correct Compiled and $+0.66/+0.60/+2.50$ points for Raw Trajectory. These interventions isolate semantic compilation, task relevance, and role alignment at fixed student parameters.

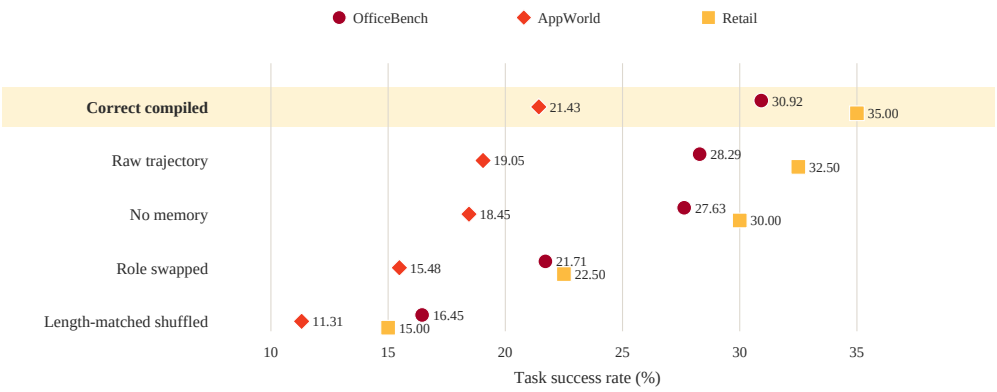


Figure 5: Correctly compiled memory achieves the highest fixed-checkpoint task success on all three benchmarks. Every condition uses identical student parameters, task order, interaction horizon, and decoding settings; only the inference-time memory artifact changes.

C.3 Broader Component Ablations

Table 21 reports component variants that may change both training and inference behavior. The corresponding heatmap appears in the main paper. Full ProSEvo leads every variant on all three benchmarks. Structure and Execution Memory each contribute measurable gains, while Success Memory, Failure Memory, and parameter updates provide larger benchmark-dependent gains. The fixed-checkpoint experiment above separately identifies the causal effect of the inference-time artifact.

C.4 Capability Carrier Controls

Table 23 instantiates the 2×2 design from the main paper. Memory-only uses the base student with the compiled external artifact. Parameter-only uses role-specific adapters with retrieval disabled. Full ProSEvo combines both carriers under matched interaction and training budgets.

Table 21: Component ablations under the unified protocol. Parentheses report successful evaluation units.

Variant	OfficeBench	AppWorld	Retail	Macro
Full ProSEvo	**30.92 (47/152)**	**21.43 (36/168)**	**35.00 (14/40)**	**29.12**
w/o Structure Memory	29.61 (45/152)	20.24 (34/168)	32.50 (13/40)	27.45
w/o Execution Memory	28.95 (44/152)	19.64 (33/168)	32.50 (13/40)	27.03
w/o Success Memory	26.97 (41/152)	19.64 (33/168)	30.00 (12/40)	25.54
w/o Failure Memory	28.29 (43/152)	20.24 (34/168)	27.50 (11/40)	25.34
Memory-only	26.97 (41/152)	16.67 (28/168)	25.00 (10/40)	22.88
Parameter-only	26.32 (40/152)	19.64 (33/168)	25.00 (10/40)	23.65
Base multi-agent	20.39 (31/152)	14.29 (24/168)	17.50 (7/40)	17.39

Table 22: Train-set success rates for the principal component and carrier variants.

Variant	OfficeBench	AppWorld	Retail
Full ProSEvo	34.46	30.00	39.19
w/o Success Memory	31.08	26.67	36.49
w/o Failure Memory	32.43	27.78	35.14
Memory-only	29.73	22.22	28.38
Parameter-only	29.05	26.67	31.00
Base multi-agent	23.65	16.67	22.97

Table 23: External-memory and parameter-update controls under the unified protocol.

Variant	Adapter	Memory	OfficeBench	AppWorld	Retail	Macro
Base	×	×	20.39	14.29	17.50	17.39
Memory-only	×	✓	26.97	16.67	25.00	22.88
Parameter-only	✓	×	26.32	19.64	25.00	23.65
Full ProSEvo	✓	✓	**30.92**	**21.43**	**35.00**	**29.12**

Memory-only adds 5.49 Macro points to Base, and Parameter-only adds 6.26 points. Their combination adds 6.24 points to Memory-only and 5.47 points to Parameter-only. The result supports complementary state-dependent retrieval and parameterized role skills.

C.5 Unified Three-Round and Matched Controls

Table 24 reports every unified single-run checkpoint and the three final-round controls. Repeat R0 replays Round 0 experience at later updates. Fixed M0 incorporates later supervision while retaining the Round 0 retrieval bank. Current Only warm-starts the student and uses the current round's supervision and memory artifact.

Table 24: Unified three-round trajectory and matched final controls. Parentheses report successful evaluation units.

Policy	OfficeBench	AppWorld	Retail	Macro
Base (pre-update)	20.39 (31/152)	14.29 (24/168)	17.50 (7/40)	17.39
Round 0 Student	25.00 (38/152)	14.88 (25/168)	20.00 (8/40)	19.96
Round 1 Student	27.63 (42/152)	17.86 (30/168)	25.00 (10/40)	23.50
Round 2 / Cumulative	**30.92 (47/152)**	**21.43 (36/168)**	**35.00 (14/40)**	**29.12**
Repeat R0	25.66 (39/152)	15.48 (26/168)	22.50 (9/40)	21.21
Fixed M0	28.29 (43/152)	17.26 (29/168)	27.50 (11/40)	24.35
Current Only	26.97 (41/152)	18.45 (31/168)	25.00 (10/40)	23.47

Round 2 exceeds Repeat R0 by 5.26, 5.95, and 12.50 points across the three benchmarks. Its gains over Fixed M0 are 2.63, 4.17, and 7.50 points, while its gains over Current Only are 3.95, 2.98, and 10.00 points. This pattern connects continual improvement to new experience, evolving memory, and historical accumulation.

C.6 Task-Level Uncertainty

Table 25 complements the unified point estimates with 95% Wilson score intervals computed from the exact successful-task counts. For each benchmark, it reports Full ProSEvo and the strongest controlled baseline. These intervals characterize uncertainty in each marginal success proportion; paired method-difference tests additionally require matched task-level outcome vectors.

Table 25: Task success and 95% Wilson score intervals under the unified protocol. Values are percentages.

Benchmark	Method	Success	95% Wilson CI
OfficeBench	**Full ProSEvo**	**30.92**	**[24.12, 38.66]**
OfficeBench	LEGOMem-style	28.95	[22.33, 36.61]
AppWorld	**Full ProSEvo**	**21.43**	**[15.90, 28.24]**
AppWorld	Skill-SD-style	19.64	[14.34, 26.30]
Retail	**Full ProSEvo**	**35.00**	**[22.13, 50.49]**
Retail	HERO-style	32.50	[20.08, 47.98]

C.7 Retrieval Depth Sensitivity

We selected $k = 3$ using training-task performance and fixed it before test evaluation. Figure 6 subsequently varies OfficeBench retrieval depth as a fixed-checkpoint diagnostic, with the student checkpoint, memory bank, and task order fixed. Success is 27.63% (42/152) at $k = 0$, 28.95% (44/152) at $k = 1$, 30.26% (46/152) at $k = 2$, 30.92% (47/152) at $k = 3$, and 30.26% (46/152) at $k = 4$. The fixed default $k = 3$ attains the highest point estimate in this diagnostic sweep. The decrease at $k = 4$ supports a shallow optimum but does not by itself establish the cause.

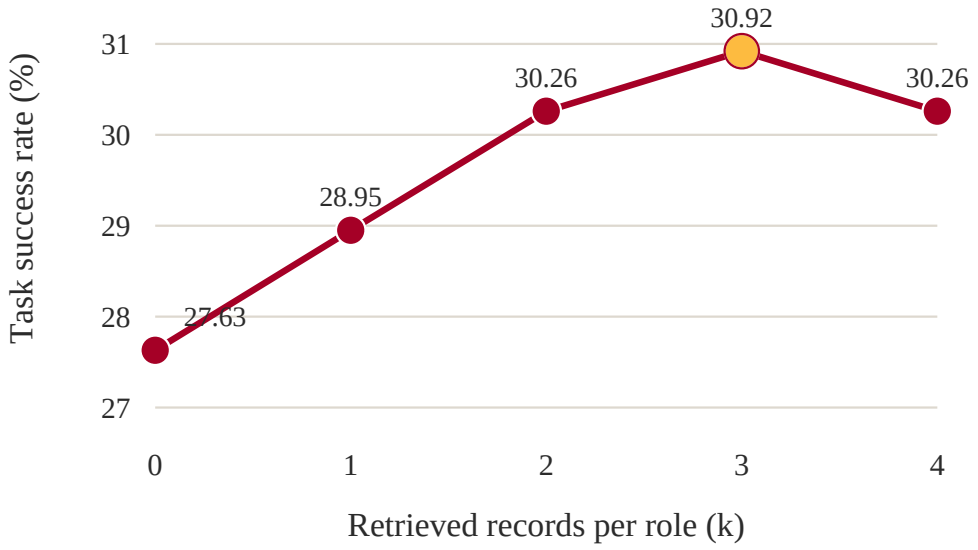


Figure 6: The training-selected depth $k = 3$ attains the highest OfficeBench point estimate in the fixed-checkpoint retrieval diagnostic. The student, memory bank, task order, and decoding settings remain fixed.

C.8 Compiler Validation

The compiler processes 738 trajectories and accepts 718 after structured output and evidence validation. Of 33 initial failures, 13 recover after the single format only retry. The remaining 20 outputs are rejected, giving a final rejection rate of 2.71%. Benchmark specific final rejection rates are 2.70% for OfficeBench, 2.22% for AppWorld, and 2.80% for Retail.

Table 26: Compiler validation and final rejection causes.

Statistic	Count
Input trajectories	738
Accepted trajectories	718
Initial validation failures	33
Recovered after retry	13
Final rejections	20
Evidence coverage failure	9
Recovery order violation	4
JSON parsing failure	3
Schema mismatch	2
Invalid positive execution evidence	2

Compiler acceptance is a validity diagnostic rather than a semantic accuracy estimate. The validation rules establish structural consistency and source grounding, but they do not measure the precision or recall of all potentially useful procedures. A separate human annotation study would be required for that claim.

D CONTROL CONSTRUCTION AND AUDIT BOUNDARIES

Trace and memory deduplication. Exact trace deduplication hashes the normalized ordered action and observation sequence together with benchmark and task identity. Repeated copies of the same trace receive one pool entry, while semantically different traces remain distinct. Exact memory deduplication hashes the memory type, normalized procedural payload, applicability conditions, and source evidence. It does not merge records merely because their natural language summaries are similar.

Trace balanced supervision. Semantic preserving views are grouped by source trace. The loss first averages views within a trace and then averages across traces, so producing additional views does not increase that trace's total weight. Controller and executor examples are formed independently and never contain retrieved memory.

Shuffled control. A shuffled donor must have the same memory type, a different source task and memory identifier, and the same application or tool when applicable. It must also have exactly the same visible token unit count after prompt formatting. The token units follow the deterministic Unicode expression `\w+|[^\w\s]`. If no exact donor exists, the run halts and records the unmatched condition. This design preserves prompt position and visible length while changing task content.

Round controls. Repeat R0 performs the Round 0 update normally and replays the same Round 0 artifacts during Rounds 1 and 2. Fixed M0 admits later SFT data during Rounds 1 and 2 but keeps retrieval memory fixed at its Round 0 state. Current Only warm-starts from the preceding adapters but limits training and retrieval artifacts to the current round. Optimizer updates and supervised gradient tokens are matched to the three-round cumulative run.

Deployment boundary. Deployment retains the local student backbone, the two LoRA adapters, the training-derived memory bank, local embedding encoder, environment interface, action validation, persistent state, and recent events. It removes the seed collector, frozen compiler, collection self consistency, collection stage verifier, collection-time recovery mechanism, and benchmark fallback policy. This removal does not disable retrieval from Failure Memory. The deployed system makes no calls to an external LLM or embedding API.

Reporting limitation. Each configuration in the current controlled study has one aggregate outcome. Table 25 reports 95% Wilson intervals for selected marginal task-success proportions, but these intervals do not capture training-run variability or establish significance for paired method differences. We therefore make no statistical-significance claims. Future work should complement these results with paired task-level analyses and independent semantic audits under the same manifests and matched budgets.