Unified Context Evolution for LLM Agents

Zixuan Zhu[1,*], **Yitong Hu**[2,*], **Yong Dai**[3], **Junfeng Fang**[4], **Chunyang Jiang**[5]
Senkang Hu[6], **Yuzhi Zhao**[7]
[1]Nanyang Technological University,
[2]Beijing University of Posts and Telecommunications
[3]Fudan University, [4]National University of Singapore
[5]Hong Kong University of Science and Technology
[6]City University of Hong Kong
[7]Huazhong University of Science and Technology
zixuan012@e.ntu.edu.sg, senkang.forest@my.cityu.edu.hk,
zyz0808.hust@gmail.com

Abstract

LLM-based agents can solve multi-step interactive tasks by combining reasoning with environment feedback, yet each episode starts from the same fixed context and any useful strategy discovered along the way is lost once the task ends. Existing approaches either limit learning to the current task or pool all experience into a single untyped store, without distinguishing knowledge types, tracking quality through use, or balancing what the library still lacks. We introduce *Unified Context Evolution* (UCE), a gradient-free framework that externalizes agent experience into an evolving library of typed *Evolvable Context Units* (ECUs). UCE decomposes experience into four complementary types (Memory, Strategy, Workflow, and Skill), each generated from trajectories under type-specific conditions, retrieved at decision time, scored through repeated usage outcomes, and pruned when no longer valuable. A scheduling module allocates each cycle's generation budget toward the types where the library is weakest. Across two interactive benchmarks, UCE raises ALFWorld success from 75.4% to 96.3% and WebShop task score from 45.1% to 61.3%, and the accumulated library transfers to alternative actor backbones without retraining.

1 Introduction

Large language models have given rise to a new class of interactive agents that act over multiple steps to solve complex tasks. By interleaving reasoning with environment-grounded actions, an agent can carry out sequential decision tasks under environment feedback [1, 2, 3], invoke external tools when its internal knowledge is insufficient [4], and explore alternative reasoning paths through structured search [5]. Yet across these settings, each task is solved as an isolated event. The agent starts from the same fixed context (system instructions, few-shot demonstrations, tool schemas), and any mechanism it discovers during one episode disappears when the next task begins.

Existing approaches address this limitation at different levels of persistence. Reinforcement learning methods train the agent's policy through environment feedback [6, 7], improving capability but requiring gradient access and substantial cost. Within-task methods let the agent revise its approach during the current task through verbal self-critique [8] or iterative

*Equal contribution

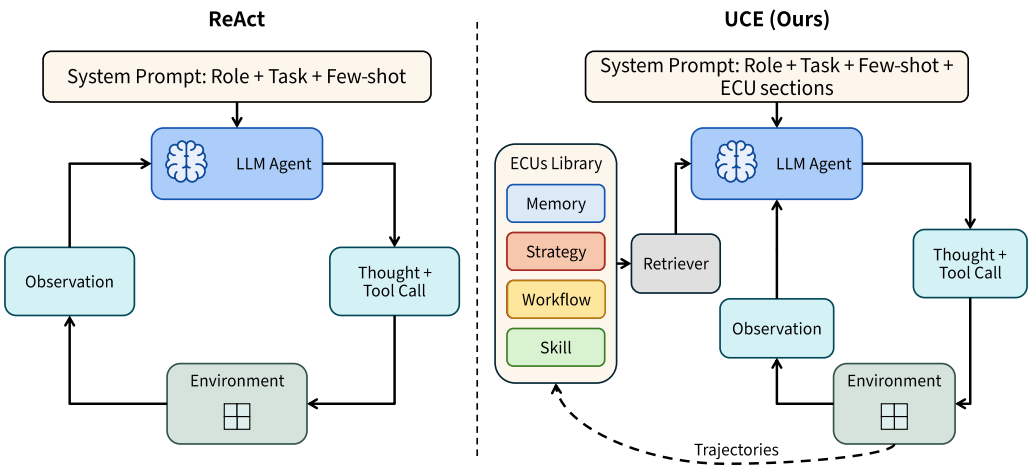

Figure 1: Comparison of the ReAct loop (left) and the UCE architecture (right).

Table 1: Four ECU types in UCE.

Type	Defining question	Generation condition	Representative knowledge
Memory	How does an environment mechanism behave?	One trajectory	A sinkbasin can clean portable items without toggling a faucet.
Strategy	What should the agent choose under a condition?	Multiple trajectories from the same task type	If `put` fails, open the receptacle and retry.
Workflow	What execution sequence should the agent follow?	Successful trajectories from the same task type	Find the target, transform it, then place it at the target receptacle.
Skill	How should a reusable operation be performed?	Trajectories from multiple task types	Search receptacles, open closed containers, and take the target when found.

refinement [9], but the resulting improvements are confined to that task and lost when a new one begins. Beyond single-task adaptation, several systems keep the model frozen and accumulate knowledge across tasks into a persistent, prompt-injectable library. Some extract mixed natural-language insights into an untyped pool [10], others specialize in a single knowledge form such as executable skills [11] or reusable workflows [12], and others learn retrieval utility through reinforcement learning on stored episodes [13]. These systems demonstrate the value of persistent knowledge, but leave open how functionally different knowledge types should coexist within one library, how their quality should be tracked through repeated use instead of one-time extraction, and how generation resources should be allocated across types as the library evolves.

We propose **Unified Context Evolution (UCE)** to make context itself the locus of learning. UCE maintains an evolving library of Evolvable Context Units (ECUs). Each ECU is a typed knowledge entry generated from collection trajectories, retrieved and injected into the actor prompt at decision time, evaluated through usage outcomes, and pruned when it becomes low-value or redundant. Figure 1 contrasts the resulting architecture with a conventional ReAct loop. In ReAct each episode runs against a fixed prompt, whereas in UCE four retrieved ECU sections augment the actor prompt at every step and the library grows across cycles without updating the model weights. The ECU library is independent of any specific actor prompt format and can be combined with different reasoning backbones.

To make the gap concrete, consider two episodes from our experiments. In a household environment [14], an agent discovers after fourteen exploratory steps that a sinkbasin alone suffices to clean a portable item, with no faucet toggle or preliminary placement in the sink required. In an online shopping environment [15], an agent clicks Buy Now from a read-only product sub-tab, receives repeated invalid-action errors, and exhausts its step budget without realizing that it must first navigate back to the main product page. Both mechanisms are reusable: the cleaning shortcut applies to every cleaning task, and the sub-tab navigation rule applies to every product page. Yet neither appears in the agent's few-shot demonstrations, and both discoveries vanish when the next task begins.

Our contributions are threefold:

- We introduce **Evolvable Context Units (ECUs)**, a typed knowledge representation that decomposes agent experience into four functionally complementary types (Memory, Strategy, Workflow, Skill), each with independent generation conditions, retrieval boundaries, and elimination rules.
- We propose **Unified Context Evolution (UCE)**, a framework that manages the full ECU lifecycle through Knowledge Yield Scheduling and a usage-based fitness score.
- We evaluate UCE against several prompt-based agent methods on two interactive benchmarks, where the ECU library delivers stable gains across evolution cycles and transfers effectively to four distinct actor backbones.

2 Related Work

2.1 Reinforcement Learning for LLM Agents

Reinforcement learning is a common route to improving LLM agents by updating model weights. WebRL [16] trains open-source web agents through a self-evolving online curriculum, and RAGEN [17] studies multi-turn agent RL in stochastic environments. Beyond direct policy training, HiPER [18] factors the policy into a high-level planner and a low-level executor for explicit credit assignment, SIOP [19] studies verifier-free turn-level credit assignment, retrieval-augmented methods use step-level experience retrieval or semantic information gain rewards [7, 20], and SkillRL [6] interleaves RL with the evolution of a reusable skill library. Broader reinforcement learning with verifiable rewards work also studies how to balance exploration and exploitation under noisy feedback [21]. While these methods can deliver strong absolute performance, they require weight access and substantial training infrastructure. Our approach takes the opposite stance. The agent backbone is left untouched and improvement comes entirely from the surrounding context, keeping the method applicable to closed-source LLM agents.

2.2 Within-Task Adaptation

Closer to our setting, a parallel line of research keeps the model frozen and adapts the agent's behavior within the current task. ReAct [1] interleaves reasoning traces with environment actions and remains a common backbone for text-based agents. Reflexion [8] appends verbal self-reflections after failed trials and retries the same task. Self-Refine [9] alternates generation and critique on a single instance. Other inference-time adaptation methods adjust search budgets or decoding distributions without model updates [22, 23]. More recent variants restructure the actor itself. ReflAct [24] rewrites the ReAct thought format so that each step explicitly reflects on the current state relative to the task goal, and MPO [25] adds a meta-planning layer that supplies high-level guidance to the actor. Although these methods refine the agent's reasoning or planning within a single task, the resulting improvements vanish once that task ends. To carry such learning forward, we store it as ECUs that later episodes can retrieve.

2.3 Cross-Task Knowledge Libraries

Most relevant to our work is a growing body of research that accumulates knowledge across tasks while keeping the model frozen. ExpeL [10] extracts mixed natural-language

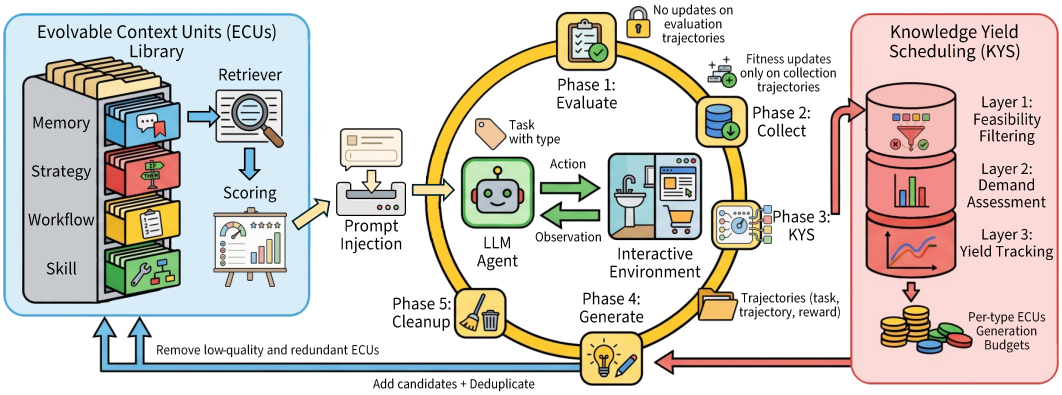

Figure 2: Overview of the UCE architecture. Five phases per cycle: evaluate, collect, KYS, generate, and cleanup.

insights into a flat pool, and Self-Generated ICE [26] curates successful trajectories as in-context examples for future tasks. Other systems specialize in a particular knowledge form. Mem0 [27] maintains long-term conversational memory, ReasoningBank [28] distills reasoning strategies from both successful and failed experiences, Agent Workflow Memory [12] induces reusable workflows from trajectories, and ACE [29] treats context as a playbook that is generated, reflected on, and curated. Voyager [11] grows an executable skill library for embodied exploration in Minecraft, and SkillWeaver [30] synthesizes reusable Python-API skills for web agents. MemRL [13] learns retrieval utility for episodic memory through reinforcement learning. Despite their diversity, these systems either store all knowledge in an untyped pool or commit to a single knowledge form, and they typically judge quality through one-shot LLM evaluation instead of repeated use. Our framework organizes four functionally complementary knowledge types within one library, each with its own generation conditions, retrieval boundaries, and lifecycle, and tracks quality through a usage-based fitness score while allocating generation budget across types via Knowledge Yield Scheduling.

3 Methodology

3.1 Problem Setup

We consider an LLM agent operating in an interactive environment $\mathcal{E}$ on a task set $\mathcal{T} = \{t_1, \ldots, t_N\}$. Each task t_i carries a type label $\tau_i \in \mathcal{Y}$. In each episode the agent receives observation o_t, produces action a_t, and the environment returns a new observation o_{t+1}. The episode ends when the task completes or the per-episode step limit is reached, producing either a binary success signal or a continuous task score depending on the environment. UCE partitions $\mathcal{T}$ into disjoint evaluation and collection subsets, ensuring that ECU content and fitness score are updated only from collection trajectories and never from evaluation trajectories.

3.2 ECU Library and Retrieval

UCE maintains a structured knowledge library $\mathcal{L} = \{e_1, \ldots, e_M\}$ of ECUs. Each ECU is represented as a five-tuple:

$$e = (z_e,\ \tau_e,\ w_e,\ c_e,\ f_e), \tag{1}$$

Here $z_e \in \mathcal{Z}$ is the functional type, one of `memory`, `strategy`, `workflow`, or `skill`. The task type $\tau_e \in \mathcal{Y} \cup \{\emptyset\}$ records where the ECU applies, with Skill using $\emptyset$ to signal cross-type applicability. The field w_e is a natural-language retrieval condition, used for matching but not injected into the prompt. The field c_e is the knowledge content injected into the agent prompt, and f_e is the fitness score defined in Section 3.3. Separating w_e from c_e lets retrieval optimize for relevance while injection exposes only actionable content to the agent.

Table 2: Main results. ALFWorld reports per-task-type and overall success rate (%). WebShop reports average task score (%) and purchase success rate (%).

Method	ALFWorld							WebShop	
	Pick	Look	Clean	Heat	Cool	Pick2	All	Score	Succ.
ReAct	83.3	50.0	96.8	78.3	71.4	52.9	75.4	45.1	30.0
UCE, initial	100.0	50.0	100.0	91.3	95.2	64.7	86.6	51.0	29.0
UCE, peak	100.0	100.0	96.8	95.7	100.0	82.4	96.3	61.3	42.0
NoThinking	91.7	27.8	87.1	73.9	81.0	58.8	73.1	58.7	37.0
NoThinking + ECU, initial	95.8	38.9	93.5	78.3	76.2	70.6	78.4	65.6	40.0
NoThinking + ECU, peak	100.0	88.9	96.8	73.9	100.0	94.1	92.5	66.8	41.0
Plan-and-Act	100.0	22.2	67.7	73.9	90.5	76.5	73.1	52.0	31.0
Plan-and-Act + ECU, initial	95.8	33.3	100.0	82.6	81.0	76.5	81.3	56.8	37.0
Plan-and-Act + ECU, peak	95.8	94.4	100.0	100.0	100.0	94.1	97.8	62.7	42.0
ReflAct	91.7	50.0	93.5	82.6	90.5	70.6	82.1	39.6	24.0
ReflAct + ECU, initial	91.7	44.4	93.5	73.9	95.2	64.7	79.9	57.4	34.0
ReflAct + ECU, peak	95.8	100.0	96.8	87.0	100.0	76.5	93.3	61.5	39.0
ExpeL	91.7	77.8	93.5	73.9	95.2	70.6	85.1	31.6	18.0
Reflexion	66.7	94.4	51.6	52.2	95.2	35.3	64.9	69.0	45.0

Four ECU types. The four ECU types are defined by the question they answer and the generation condition that produces them. Table 1 summarizes the four types, with representative examples drawn from UCE runs.

Memory captures environment mechanisms discovered through trial-and-error within a single task instance, as a factual statement that asserts a property of the environment without prescribing any sequence of steps. Strategy encodes conditional decision rules at branching points within a task type, expressed in IF-THEN form. Workflow describes the canonical execution sequence for a task type, generalized so that it applies to all instances of that type. Skill specifies multi-step sub-operation methods that are reusable across task types. These four types coordinate at complementary granularities during agent execution. Memory explains the mechanisms behind specific environment interactions, Strategy guides the agent at decision points, Workflow provides the execution skeleton for a task type, and Skill expands specific steps within that skeleton.

Retrieval. Before executing task t_i (type τ_i, natural-language description d_i), the retriever scores each ECU $e \in \mathcal{L}$:

$$\begin{aligned} \text{score}(e, t_i) = \; & \underbrace{\text{sim}(\mathbf{h}_{w_e}, \mathbf{h}_{d_i})}_{\text{semantic similarity}} \\ & + \underbrace{\beta \cdot \mathbf{1}[\tau_e = \tau_i]}_{\text{type-match boost}} + \underbrace{\lambda \cdot \big(f_{\tau_i}(e) - 0.5\big)}_{\text{quality bias}}, \end{aligned} \tag{2}$$

Here $\mathbf{h}$ denotes sentence-transformer embeddings, and $\beta, \lambda > 0$ weight the task-type boost and quality bias. Before scoring, we apply a strict task-type filter. ECUs whose task type is nonempty and differs from the current task type are excluded, except for Skill ECUs, which are designed for cross-type reuse. Each ECU type is then independently ranked, and the top-K per type are injected into four dedicated prompt sections, preventing any single type from crowding out the others.

3.3 Fitness Score and Lifecycle

Each ECU tracks its empirical contribution through a Laplace-smoothed fitness score:

$$f_e = \frac{s_e + 1}{u_e + 2}, \tag{3}$$

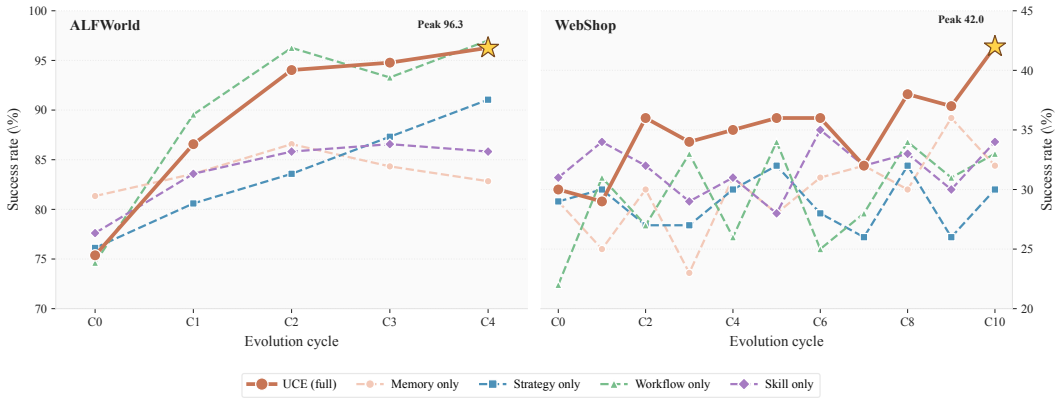

Figure 3: Per-cycle success rate on ALFWorld (C0–C4) and WebShop (C0–C10). Solid lines show full UCE, dashed lines show single-type ablations, and the star marks the peak cycle. C0 is the ReAct baseline.

Here u_e is the cumulative usage count of e, and s_e is the cumulative outcome credit. For binary-reward environments, s_e increments by one on success and by zero on failure. For continuous-reward environments, s_e increments by the final reward, so partial outcomes contribute fractional credit. A newly created ECU ($u_e = 0$) has $f_e = 0.5$, representing the prior belief that it is not yet verified.

For finer-grained quality assessment, we also maintain task-type-conditional statistics. When ECU e has been used at least u_{cond} times on task type τ, retrieval uses the conditional fitness score $f_\tau(e) = (s_\tau + 1)/(u_\tau + 2)$. Otherwise, retrieval falls back to the global fitness score f_e.

We define an *adequate ECU* as one whose fitness score has been verified by sufficient usage and remains at or above the uninformative prior. Memory, Strategy, and Workflow generation skip task types already covered by an adequate ECU of the corresponding type, and Skill generation skips once an adequate Skill ECU exists.

3.4 Evolution Cycle

Each evolution cycle c executes five phases that transform the library from $\mathcal{L}_c$ to $\mathcal{L}_{c+1}$, as illustrated in Figure 2.

Phase 1: Evaluate. The agent runs on the held-out evaluation task set with the current library $\mathcal{L}_c$ and records the success rate r_c. No fitness score is updated from these episodes, preserving the integrity of the evaluation signal.

Phase 2: Collect. The agent runs on the disjoint collection task set with the same retrieval policy, storing each episode as a (task, trajectory, reward) tuple and recording the collection success rate q_c. Fitness scores are updated only from collection outcomes.

Phase 3: Schedule. KYS reads the current library profile, the collection success rate q_c, the previous cycle's realized generation counts, and the nominal budget B, and produces integer per-type generation budgets $\{b_c^{(k)}\}_{k \in \mathcal{Z}}$ without mutating $\mathcal{L}_c$.

Phase 4: Generate. For each ECU type with a nonzero budget, a type-specific generator prompt distills candidate ECUs from the collected trajectories. Each prompt includes quality feedback drawn from high- and low-fitness-score ECUs of the same type when available, together with a reshuffle test that rejects instance-specific observations (full prompts in Appendix G). Retained candidates are added to $\mathcal{L}_c$ after within-type deduplication.

Phase 5: Cleanup. ECUs that have accumulated sufficient usage but whose fitness score falls below a threshold are removed. Within each (type, task_type) group, only the highest-scoring ECU is kept once members have sufficient usage. The surviving ECUs form $\mathcal{L}_{c+1}$.

3.5 Knowledge Yield Scheduling

KYS exploits structural information already produced by the library. Generator skip conditions, task-type coverage, and fitness-score distributions are deterministic properties of $\mathcal{L}$ that can be read directly without additional computation. KYS organizes these signals into three layers, each providing a different view of the library's state.

Layer 1: Feasibility filtering. The scheduler first determines whether each ECU type has remaining generation space. For Memory, Strategy, and Workflow, a type is feasible when at least one known task type lacks an adequate ECU of that type. For Skill, feasibility holds when no adequate skill exists yet. Infeasible types receive zero budget, which prevents saturated types from consuming generation calls.

Layer 2: Demand assessment. For each feasible type, demand $d_c^{(k)}$ measures the current knowledge gap. Let $|\mathcal{Y}|$ denote the number of task types, and let $|\mathcal{Y}_{\text{cov}}^{(k)}|$ denote the number of task types covered by adequate ECUs of type k. Memory and Strategy benefit from quality refinement even after initial coverage, so their demand combines a coverage gap with a quality gap:

$$d_c^{(k)} = \left(1 - \frac{|\mathcal{Y}_{\text{cov}}^{(k)}|}{|\mathcal{Y}|}\right) + \max\left(0,\, 0.5 - \bar{f}^{(k)}\right). \tag{4}$$

Workflow needs one adequate procedure per task type, so demand measures coverage alone:

$$d_c^{(k)} = \left(|\mathcal{Y}| - |\mathcal{Y}_{\text{cov}}^{(k)}|\right) / |\mathcal{Y}|. \tag{5}$$

Skill is cross-type, so demand is driven by the total count of adequate skills relative to a soft reference n_{ref}, with a floor to prevent starvation:

$$d_c^{(k)} = \max\left(0.1,\, 1 - n_{\text{adeq}}^{(k)} / n_{\text{ref}}\right). \tag{6}$$

Layer 3: Yield tracking. KYS maintains an exponential moving average of each type's historical return:

$$y_c^{(k)} = (1-\gamma) y_{c-1}^{(k)} + \gamma \Delta q_c \frac{g_{c-1}^{(k)}}{\sum_j g_{c-1}^{(j)}}, \tag{7}$$

where $\Delta q_c = q_c - q_{c-1}$ is the collection set success-rate change, and $g_{c-1}^{(k)}$ is the number of type k ECUs generated in the previous cycle. The EMA is initialized to 0.5 for every ECU type. The yield update is skipped in the first cycle because no previous generation counts exist.

Score fusion and budget assignment. For each feasible type, KYS computes

$$s_c^{(k)} = \alpha_c d_c^{(k)} + (1 - \alpha_c) y_c^{(k)}, \tag{8}$$

$$\alpha_c = \max(\alpha_{\min}, \alpha_{\text{start}} - \alpha_{\text{decay}} c). \tag{9}$$

Scores are normalized over feasible types, then each feasible type receives a minimum share ρ. The remaining share is assigned in proportion to normalized scores and rounded to integer budgets summing to B. The realized number of added ECUs can still be lower than the assigned budget if the generator returns fewer valid candidates or deduplication removes near duplicates.

4 Experiments

4.1 Setup

Benchmarks. We evaluate UCE on two text-based interactive benchmarks, ALFWorld [14] and WebShop [15]. ALFWorld contains 134 household tasks across six task types. WebShop uses a 100-task split over Amazon product pages. Full benchmark statistics, task-type definitions, WebShop category mapping, and the k=2 stratified fold split are in Appendix A.

Table 3: Single-type ablation. Each row retains only one ECU type in the library.

ECU type	Snapshot	ALFWorld							WebShop	
		Pick	Look	Clean	Heat	Cool	Pick2	All	Score	Succ.
Memory	Initial	87.5	61.1	90.3	87.0	81.0	88.2	83.6	47.0	25.0
Memory	Peak	100.0	61.1	93.5	87.0	81.0	88.2	86.6	54.8	36.0
Strategy	Initial	87.5	66.7	93.5	87.0	81.0	52.9	80.6	48.7	30.0
Strategy	Peak	95.8	72.2	96.8	91.3	95.2	88.2	91.0	50.6	32.0
Workflow	Initial	95.8	83.3	96.8	100.0	95.2	52.9	89.6	50.1	31.0
Workflow	Peak	100.0	77.8	100.0	100.0	100.0	100.0	97.0	54.7	34.0
Skill	Initial	91.7	50.0	96.8	91.3	85.7	70.6	83.6	58.2	34.0
Skill	Peak	100.0	50.0	90.3	91.3	90.5	88.2	86.6	55.0	35.0

Evaluation protocol. ALFWorld is run for five cycles and WebShop for eleven cycles. Cycle 0 is the ReAct baseline, and Cycle k reports performance after k rounds of evolution. Each task is executed once per cycle with no multi-attempt retry.

Models and hyperparameters. The acting agent is `gpt-4.1-mini` [31] and the generator is `gpt-5.2` [32]. Retrieval uses sentence-transformer embeddings [33]. Full configuration is in Appendix B.

Compared methods. We compare UCE against two cross-paradigm baselines (ExpeL [10] and Reflexion [8]) and test ECU injection on three alternative actor backbones (NoThinking [34], Plan-and-Act [24], and ReflAct [24]). All methods are reproduced under the same actor model, generator/insight model, task sets, and step budgets. Per-method protocol details are deferred to Appendix E.

4.2 Main Comparison

Table 2 reports the main results. On ALFWorld, UCE reaches 96.3% success rate at its peak, a gain of 20.9 percentage points over the ReAct baseline. The largest improvements appear on the weakest baseline task types. Look rises from 50.0% to 100.0% and Pick2 from 52.9% to 82.4%. On WebShop, purchase success improves from 30.0% to 42.0% and task score from 45.1% to 61.3%.

Most ALFWorld gains arrive early, success jumps from 75.4% at Cycle 0 to 86.6% after the initial library and 94.0% after one further update. The trajectory is not strictly monotonic for every task type, because each task is evaluated once per cycle under a newly retrieved library. Figure 3 gives the full evolution curves. Per-cycle tables for both benchmarks are in Appendix D.

The ECU library transfers across actor backbones. On ALFWorld, Plan-and-Act with UCE attains the overall best peak of 97.8%, up from its 73.1% no-library baseline. ReflAct with UCE improves from 82.1% to 93.3%, and NoThinking with UCE improves from 73.1% to 92.5%. On WebShop, task score improves by 8.1 to 21.9 percentage points across the three backbones. The knowledge captured in the ECU library is therefore not tied to the ReAct prompt format.

Among the cross-paradigm baselines, ExpeL reaches 85.1% on ALFWorld but only 18.0% purchase success on WebShop, where its flat insight pool provides less actionable guidance for the multi-step shopping flow. Reflexion reaches 64.9% on ALFWorld and 45.0% on WebShop under a pass@3 protocol. Its per-task multi-trial budget gives it a compute advantage on WebShop that single-trial methods do not share.

UCE's improvement over the ReAct baseline is statistically significant on both benchmarks: paired McNemar and bootstrap tests reject the null at $p < 0.05$, with all 95% confidence intervals strictly above zero (Appendix C).

Main-text case study: WebShop pillow-cover task

Task. The agent must buy yellow, machine-washable Batmerry decorative pillow covers under $50.

Configuration	Steps	Reward	#ECUs	Outcome
Full UCE	9	1.00	3	succeeds after recovering from a sub-tab
Memory only	15	0.00	3	invalid-action timeout
Strategy only	15	0.00	3	invalid-action timeout
Workflow only	15	0.00	1	invalid-action timeout
Skill only	15	0.00	1	invalid-action timeout

All configurations locate the relevant product page. The restricted runs click `Buy Now` while trapped inside the read-only `Features` sub-tab, receive repeated invalid-action feedback, and exhaust the step budget. Full UCE retrieves a workflow ECU for search and option selection, a memory ECU stating that product sub-tabs are read-only states, and a strategy ECU for returning with `< Prev`. After the first invalid `Buy Now`, the agent returns to the product page and completes the purchase. Case Study F.4 gives the full side-by-side trajectory.

Figure 4: A WebShop task that requires multiple ECU types to succeed.

Table 4: WebShop failure modes at Cycle 8.

Failure mode	Share
Partial-mid attribute mismatch	37%
Max-step timeout	27%
Zero-match product failure	15%
Partial-high attribute mismatch	13%
Partial-low attribute mismatch	8%

4.3 Single-Type Ablation

Table 3 constrains the library to a single ECU type. On ALFWorld, Workflow alone reaches 97.0%, close to the full library's 96.3%, which indicates that reusable procedural skeletons explain the majority of the gain on this benchmark. Strategy peaks at 91.0% and contributes at a finer decision-point granularity, while Memory (86.6%) and Skill (86.6%) provide smaller but complementary gains. On WebShop, the picture differs: no single type matches the full library, and the best single-type peak (Memory, 36.0% success) remains 6.0 percentage points below the full library's 42.0%. This gap reflects the multi-faceted nature of the shopping task, where the agent needs procedural workflow guidance, recovery strategies for navigation traps, and factual memory about page mechanisms to complete purchases reliably.

4.4 Error Analysis and Case Study

Table 4 categorizes the 62 failed WebShop episodes at Cycle 8. Half of them (partial-mid plus partial-high, 50%) are attribute mismatches on a broadly correct product, where the agent identifies the right item type but selects a wrong size, color, or quantity option. These errors mark a natural boundary of prompt-level knowledge. UCE captures *what to know* about environment mechanisms, decision rules, and procedures, while precise selection from a page-specific attribute panel primarily requires *how to operate*, which is the strength of policy-optimizing reinforcement-learning methods. The next-largest category is max-step timeouts (27%), which typically arise when the agent enters repeated search-click loops without committing to a purchase. Zero-match failures (15%) reflect a different mode, where the agent commits to a product whose top-level category is wrong, indicating that the initial search query missed the intended item space. Partial-low failures (8%) are residual cases where the agent identifies the right category but matches only a small fraction of the requested attributes. ALFWorld failures at the peak cycle (5 of 134) concentrate on

Pick2 timeouts, the longest task type, where workflow and skill ECUs still leave room for improvement.

Figure 4 illustrates the dominant failure mode through a matched WebShop trajectory. The task is to purchase yellow, machine-washable Batmerry decorative pillow covers under $50, and all five configurations issue the same opening search and successfully open the correct product page. From this shared state the trajectories diverge sharply. The four restricted libraries (Memory only, Strategy only, Workflow only, Skill only) each click into the `Features` sub-tab to inspect the listing, then attempt `Buy Now` while still inside the sub-tab. WebShop returns `Invalid action!`, and without any rule for recovering, the agent re-issues `Buy Now` repeatedly until the 15-step budget runs out with no purchase made. Full UCE retrieves three complementary ECUs that together resolve the same situation: a Workflow ECU encoding the search-then-purchase sequence, a Memory ECU recording the fact that product sub-tabs are read-only navigation states, and a Strategy ECU prescribing `< Prev` as the recovery action when an attempted purchase fails. After the first invalid `Buy Now`, the Strategy ECU triggers a `< Prev` that returns the agent to the main product page, where `Buy Now` succeeds with a reward of 1.0. This trajectory illustrates a general pattern visible across our Appendix case studies: when a single knowledge type drives the agent into a local trap, the surrounding ECU types supply the orthogonal information needed to escape it, and successful runs typically combine two or three types. Additional case studies are in Appendix F.

5 Conclusion

We have presented Unified Context Evolution (UCE), a gradient-free framework in which an LLM agent improves across episodes by maintaining an evolving library of typed Evolvable Context Units. A usage-based fitness score tracks each unit's value, and Knowledge Yield Scheduling directs generation resources toward the weakest regions of the library. UCE raises ALFWorld success from 75.4% to 96.3% and WebShop task score from 45.1% to 61.3%, and the accumulated library transfers across actor backbones.

Looking ahead, the four ECU types are currently hand-designed. An automatic type-discovery mechanism could adapt library structure to new domains. More broadly, coupling the fitness-driven lifecycle with lightweight parameter updates could yield agents that learn both what to retrieve and how to act on what they retrieve.

6 Limitations

We evaluate UCE on two text-based interactive environments with discrete action spaces, using a single closed-source model pair (`gpt-4.1-mini` as agent, `gpt-5.2` as generator), so the current results do not disentangle how much of the improvement stems from the framework's design, the generator's strong distillation capability, or the relative regularity of these two environments. Testing on settings with continuous actions, multimodal observations, or longer planning horizons such as AndroidWorld [35] or VisualWebArena [36], and varying the generator model independently, would help isolate these factors. Our task pools are also relatively small (134 ALFWorld tasks and 100 WebShop tasks), and the scheduling signals that drive KYS are estimated from them. Whether these signals remain reliable, and whether the library continues to improve, plateaus, or eventually degrades under substantially larger task sets or longer evolution horizons remains an open question.

References

[1] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*, 2023.

[2] Senkang Hu, Zhengru Fang, Zihan Fang, Yiqin Deng, Xianhao Chen, and Yuguang Fang. AgentsCoDriver: Large Language Model Empowered Collaborative Driving with Lifelong Learning, 2024.

[3] Senkang Hu, Zhengru Fang, Zihan Fang, Yiqin Deng, Xianhao Chen, Yuguang Fang, and Sam Tak Wu Kwong. AgentsCoMerge: Large Language Model Empowered Collaborative Decision Making for Ramp Merging. *IEEE Transactions on Mobile Computing*, 24(10):9791–9805, 2025. doi: 10.1109/TMC.2025.3564163.

[4] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc., 2023.

[5] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 11809–11822. Curran Associates, Inc., 2023.

[6] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving Agents via Recursive Skill-Augmented Reinforcement Learning, 2026.

[7] Prince Zizhuang Wang and Shuli Jiang. SLEA-RL: Step-Level Experience Augmented Reinforcement Learning for Multi-Turn Agentic Training, 2026.

[8] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023.

[9] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc., 2023.

[10] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. *journaltitle = Proceedings of the AAAI Conference on Artificial Intelligence,*, 38(17):19632–19642, 2024. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v38i17.29936.

[11] 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. *journaltitle = Transactions on Machine Learning Research,*, pages 1–41, 2024. ISSN 2835-8856.

[12] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory, 2024.

[13] Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-Evolving Agents via Runtime Reinforcement Learning on Episodic Memory, 2026.

[14] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *International Conference on Learning Representations*, 2021.

[15] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 20744–20757. Curran Associates, Inc., 2022.

[16] Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM Web Agents via Self-Evolving Online Curriculum Reinforcement Learning. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 79791–79821, 2025.

[17] Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding Self-Evolution in LLM Agents via Multi-Turn Reinforcement Learning, 2025.

[18] Jiangweizhi Peng, Yuanxin Liu, Ruida Zhou, Charles Fleming, Zhaoran Wang, Alfredo Garcia, and Mingyi Hong. HiPER: Hierarchical Reinforcement Learning with Explicit Credit Assignment for Large Language Model Agents, 2026.

[19] Senkang Hu, Yong Dai, Xudong Han, Zhengru Fang, Yuzhi Zhao, Sam Tak Wu Kwong, and Yuguang Fang. Self-Induced Outcome Potential: Turn-Level Credit Assignment for Agents without Verifiers. *arXiv preprint arXiv:2605.04984*, 2026.

[20] Senkang Hu, Yong Dai, Yuzhi Zhao, Yihang Tao, Yu Guo, Zhengru Fang, Sam Tak Wu Kwong, and Yuguang Fang. Optimizing Agentic Reasoning with Retrieval via Synthetic Semantic Information Gain Reward. *arXiv preprint arXiv:2602.00845*, 2026.

[21] Donglai Xu, Hongzheng Yang, Yuzhi Zhao, Pingping Zhang, Jinpeng Chen, Wenao Ma, Zhijian Hou, Mengyang Wu, Xiaolei Li, Senkang Hu, et al. From Exploration to Exploitation: A Two-Stage Entropy RLVR Approach for Noise-Tolerant MLLM Training. *arXiv preprint arXiv:2511.07738*, 2025.

[22] Zhengru Fang, Senkang Forest Hu, Zhonghao Chang, Yu Guo, Yihang Tao, Hongyao Liu, Mengzhe Ruan, Jun Huang, and Yuguang Fang. Inference-Time Budget Control for LLM Search Agents. *arXiv preprint arXiv:2605.05701*, 2026.

[23] Senkang Hu, Xudong Han, Jinqi Jiang, Yihang Tao, Zihan Fang, Yong Dai, Sam Kwong, and Yuguang Fang. Distribution-Aligned Decoding for Efficient LLM Task Adaptation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026.

[24] Jeonghye Kim, Sojeong Rhee, Minbeom Kim, Dohyung Kim, Sangmook Lee, Youngchul Sung, and Kyomin Jung. ReflAct: World-Grounded Decision Making in LLM Agents via Goal-State Reflection. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 33421–33453. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1697.

[25] Weimin Xiong, Yifan Song, Qingxiu Dong, Bingchan Zhao, Feifan Song, XWang, and Sujian Li. MPO: Boosting LLM Agents with Meta Plan Optimization. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 3914–3935. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.210.

[26] Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-Generated in-Context Examples Improve LLM Agents for Sequential Decision-Making Tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 64392–64425. Curran Associates, Inc., 2025.

[27] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory, 2025.

[28] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. In *The Fourteenth International Conference on Learning Representations*, 2026.

[29] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. In *The Fourteenth International Conference on Learning Representations*, 2026.

[30] Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web Agents Can Self-Improve by Discovering and Honing Skills, 2025.

[31] OpenAI. Introducing GPT-4.1 in the API, April 2025. https://openai.com/index/gpt-4-1/.

[32] OpenAI. Introducing GPT-5.2, December 2025. https://openai.com/index/introducing-gpt-5-2/.

[33] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings Using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3980–3990. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1410.

[34] Wenjie Ma, Jingxuan He, Charlie Snell, Tyler Griggs, Sewon Min, and Matei Zaharia. Reasoning Models Can Be Effective without Thinking, 2025.

[35] Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A Dynamic Benchmarking Environment for Autonomous Agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 406–441, 2025.

[36] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 881–905. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.50.

[37] Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated Generation and Selection of Context-Aware Guidelines for Large Language Model Agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 119919–119948. Curran Associates, Inc., 2024. doi: 10.52202/079017-3811.

[38] Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. RAP: Retrieval-Augmented Planning with Contextual Memory for Multimodal LLM Agents, 2024.

[39] Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing Instruction Manuals by LLM Agents via Interactive Environmental Learning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 589–631. Curran Associates, Inc., 2024. doi: 10.52202/079017-0019.

[40] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.147.

A Benchmark Details

A.1 ALFWorld

ALFWorld [14] is a text-based household environment in which an agent receives natural-language observations and produces natural-language actions to complete domestic tasks such as cleaning, heating, cooling, and placement. Following the standard protocol used by ExpeL [10], we evaluate on the 134-task subset.

The 134 tasks span six task types: *Pick*, *Look*, *Clean*, *Heat*, *Cool*, and *Pick2* (the underlying ALFWorld labels are `put`, `examine`, `clean`, `heat`, `cool`, `puttwo`). Table 5 reports the per-type distribution, and each fold preserves the proportion of the full set so within-fold success rates remain comparable to literature reports on the full 134-task set.

Table 5: ALFWorld task type distribution.

	Clean	Cool	Look	Heat	Pick	Pick2	Total
Full 134 tasks	31	22	18	23	24	16	134
Fold 1 (Group A)	15	11	9	11	12	8	67
Fold 2 (Group B)	16	11	9	12	12	8	67

A.2 WebShop

WebShop [15] is a text-based online shopping environment with a catalog of 1.18 million real Amazon products. We adopt the 100-task evaluation set released by ExpeL [10], partitioned into two folds of 50 tasks each.

For each evaluated task we report two metrics. *Success rate* is the fraction of episodes whose final reward equals exactly 1.0, indicating that the agent purchased a product matching all attributes specified in the instruction. *Task score* is the average per-episode continuous reward, which credits partial attribute matches and can exceed binary success rate when failed purchases still match some requested attributes. Both metrics are computed using the standard WebShop reward function exactly as released by the WebShop server.

A.2.1 Task Subtype Mapping

WebShop does not provide a per-task type label, but each task carries a `key.query` field that records the Amazon catalog query used to populate its product candidates. We map each task deterministically to one of five subtypes (*beauty*, *clothing*, *electronics*, *food*, *home*) by keyword-matching the query against curated category vocabularies derived from the Amazon catalog hierarchy. *Beauty* covers personal-care queries such as "bath & bathing accessories", "body care", "eye makeup", "face care", and "dental care". *Clothing* covers apparel and footwear such as "men's jackets", "women's dresses", and "women's mules & clogs". *Electronics* covers consumer-electronics such as "bluetooth headsets", "chargers", and "home theater systems". *Food* covers grocery queries such as "alcoholic beverages", "cheese & charcuterie gifts", and "meat & seafood". *Home* covers furniture and home goods such as "bedroom sets", "decorative pillows", and "kitchen islands".

Table 6 reports the resulting distribution. The five-category decomposition supports UCE's task-type-conditioned retrieval, because memory, strategy, and workflow ECUs are bound to their generating task type.

B Implementation Details

B.1 Evolution Cycle Algorithm

Algorithm 1 summarizes one complete evolution run of UCE. The algorithm preserves the read-only contract for the evaluation set and the disjoint collection set used for fitness-score

Table 6: WebShop task subtype distribution.

	Beauty	Clothing	Electronics	Food	Home	Total
Tasks	31	14	21	15	19	100
Fold 1	16	7	10	8	9	50
Fold 2	15	7	11	7	10	50

Algorithm 1 Unified Context Evolution

Require: Initial library $\mathcal{L}_0 = \emptyset$, agent π, eval tasks $\mathcal{E}$, collect tasks $\mathcal{Q}$, budget B, cycles C

$\mathbf{g}_{-1} \leftarrow \mathbf{0}$ ▷ per-type generation counts from prev cycle
for cycle $c = 0, \ldots, C-1$ **do**
  Phase 1 (Evaluate): Run π with $\mathcal{L}_c$ on $\mathcal{E}$ and record success rate r_c (no fitness update)
  Phase 2 (Collect): Run π with $\mathcal{L}_c$ on $\mathcal{Q}$, record $\mathcal{T}_c$, update ECU fitness scores, and compute collection success rate q_c
  Phase 3 (Schedule): $\{b^{(k)}\}_{k \in \mathcal{Z}} \leftarrow \mathrm{KYS}(\mathcal{L}_c, q_c, B, \mathbf{g}_{c-1})$ ▷ Section 3.5
  $\mathbf{g}_c \leftarrow \mathbf{0}$
  for each ECU type $k \in \mathcal{Z}$ with $b^{(k)} > 0$ **do**
    Phase 4 (Generate): Generate $\leq b^{(k)}$ candidate ECUs of type k from $\mathcal{T}_c$ using the type-specific prompt with quality-feedback exemplars and reshuffle test, then add survivors to $\mathcal{L}_c$ and record their count in $g_c^{(k)}$
  end for
  Dedup: within each type, remove pairs with cosine similarity > 0.85 (keep the higher-fitness-score one)
  Phase 5 (Cleanup): drop ECUs with $u_e \geq u_{\min}$ and $f_e \leq f_{\max}$, and within each (type, task_type) group, keep only the highest-fitness-score ECU
  $\mathcal{L}_{c+1} \leftarrow \mathcal{L}_c$
end for
return $\mathcal{L}_C$

updates, so the success rate r_c reported per cycle is not inflated by fitness-score updates on evaluation trajectories.

B.2 Hyperparameters

Table 7 lists all hyperparameters used in the main results.

B.3 Models and Compute

The agent backbone (`gpt-4.1-mini`) is invoked at every environment step in both the evaluation and collection phases. The generator backbone (`gpt-5.2`) is invoked only during the Generate phase of each cycle. A full main-results run consumes on the order of tens of thousands of agent calls but only a few hundred generator calls. The generator nevertheless dominates wall-clock cost because of its longer outputs.

All experiments run against the official OpenAI API endpoints and require no local GPU. The sentence-transformer embedding model (`all-MiniLM-L6-v2`) runs on a single CPU thread. WebShop additionally requires running its Flask server on a separate process, which loads the full 1.18-million-product Amazon catalog and a per-product search index into memory. The server needs roughly 40 GB of RAM and several minutes of cold-start time before the agent can issue queries.

ALFWorld [14] and WebShop [15] are released under the MIT License, and the `all-MiniLM-L6-v2` sentence-transformer [33] is released under Apache 2.0. All artifacts are used in a manner consistent with their intended research use.

Table 7: Full hyperparameter list.

Component	Setting	Role
Agent model	`gpt-4.1-mini`, T=0.7 `max_tokens`= 256	Acting policy for each environment step.
Generator model	`gpt-5.2`, T=0.4 `max_tokens`= 1024	Stable ECU extraction.
Embedding	`all-MiniLM-L6-v2`	Similarity space for ECUs and tasks.
Retrieval	$K = 3$, $\beta = 0.15$, $\lambda = 0.1$ dedup = 0.85	Per type top-K injection, task boost, fitness-score bias, and duplicate cutoff.
KYS	$B = 6$, $\gamma = 0.7$ α_{start}=1.0, α_{decay}=0.1 $\alpha_{\min}$=0.5, `min_share`= 0.10 $n_{\text{ref}} = 3$, skill floor = 0.1	Adaptive per type generation budget.
Cleanup	$u_{\min} = 5$ $f_{\max} = 0.3/0.2$	Usage gate and benchmark-specific fitness-score cutoff.
Generator skip	u_{adq}=2, f_{adq}=0.5	Skip generation when enough adequate ECUs exist.
Evolution loop	5 / 11 cycles 30 / 20 collect tasks	Number of evolution cycles and collect tasks.

Table 8: Statistical significance of UCE vs. the ReAct.

Benchmark	Δ vs. C0	95% bootstrap CI	p-value
ALFWorld succ.	+20.9 pp	[+13.4, +29.1]	7.7×10^{-7}
WebShop succ.	+12.0 pp	[+3.0, +21.0]	0.017
WebShop score	+16.2 pp	[+7.3, +24.7]	$< 10^{-3}$

C Statistical Significance

We test UCE's improvement over the ReAct baseline with an exact McNemar test on binary success and a 10,000-resample paired bootstrap on the per-task delta between Cycle 0 and the peak cycle. Table 8 reports the results.

The first two rows use McNemar's exact test (binary outcome), and the third uses the bootstrap (continuous reward). All three comparisons reject the null at $p < 0.05$, and all confidence intervals sit strictly above zero.

D Per-Cycle Evolution Trajectories

Table 9 reports the ALFWorld per-cycle, per-task-type success rate. Table 10 reports the per-cycle combined success rate and task score on WebShop. Cycle 0 is the ReAct baseline, equivalent to the *ReAct* row in Table 2.

E Compared Methods

We compare against two cross-paradigm methods (ExpeL and Reflexion) and three alternative actor backbones (NoThinking, Plan-and-Act, ReflAct). All share UCE's actor model (`gpt-4.1-mini`), generator/reflector model (`gpt-5.2`), task sets, and per-episode step bud-

Table 9: ALFWorld per-cycle per-task-type success rate (%) on the 134-task evaluation set across 5 cycles.

Cycle	Pick	Look	Clean	Heat	Cool	Pick2	**All**
C0	83.3	50.0	96.8	78.3	71.4	52.9	75.4
C1	100.0	50.0	100.0	91.3	95.2	64.7	86.6
C2	100.0	100.0	100.0	82.6	100.0	76.5	94.0
C3	100.0	94.4	100.0	95.7	95.2	76.5	94.8
C4	**100.0**	**100.0**	**96.8**	**95.7**	**100.0**	**82.4**	**96.3**

Table 10: WebShop per-cycle success rate and task score (%) on the 100-task evaluation set across 11 cycles.

Cycle	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	**C10**
Success rate	30.0	29.0	36.0	34.0	35.0	36.0	36.0	32.0	38.0	37.0	**42.0**
Task score	45.1	51.0	56.3	54.8	52.3	58.3	54.1	54.6	57.7	55.0	**61.3**

gets (50 on ALFWorld, 15 on WebShop). ExpeL and ReflAct use the same k=2 stratified split as UCE, since their library or insight-mining stages require a disjoint collect set. Reflexion has no across-task information flow and is therefore run on the merged full task set without folds, matching the convention of its original paper and of follow-up work [24, 37, 38, 39].

ExpeL is a three-stage pipeline that gathers training-task trajectories with a reflective ReAct agent, extracts a flat pool of natural-language insights from successful trajectories (capped at 10 for ALFWorld and 8 for WebShop), and at evaluation time injects this pool together with k nearest-neighbour past-trajectory few-shots into the actor prompt for a single attempt per task.

Reflexion gives each task up to N trials, with a reflector LLM summarising each failed trial into a plan that the next trial reads. We adopt N=3, the convention shared by AutoGuide [37], ReflAct [24], RAP [38], and AutoManual [39]. Table 2 reports the resulting pass@3 success.

The three actor backbones are reasoning-format variants of the same ReAct loop. No-Thinking [34] drops the reasoning step entirely and emits an action at every turn. Plan-and-Act [24], inspired by Plan-and-Solve prompting [40], reasons only at the first turn (a high-level plan) and emits direct actions thereafter. ReflAct [24] prepends every turn with a one-sentence reflection on the agent's current state relative to the task goal. For each backbone, the prompt design and the procedure for deriving its few-shot examples from ReAct's base examples are taken from ReflAct [24]. The corresponding rows in Table 2 run each backbone first with an empty UCE library and then inside the full UCE pipeline, substituting the chosen backbone for the default ReAct actor.

F Case Studies

This appendix traces four task instances across the full UCE library and the four single ECU type ablations at each configuration's peak cycle, all on the same agent backbone with the same initial observation. We cover two ALFWorld tasks, Look-Book (Case Study F.1) and Cool-Mug (Case Study F.3), and two WebShop tasks, Mules-Clogs (Case Study F.2) and Pillow-Cover (Case Study F.4). For each instance we report a summary table across all five configurations and a side-by-side trajectory comparison for the configuration pair most informative about the underlying mechanism. green marks steps where the additional ECUs of the full library produce a successful pivot, and red marks steps where the restricted configuration enters an unrecoverable failure mode.

Case Study F.1: Look-Book Task (ALFWorld)

Task instruction: `examine the book with the desklamp.`

	Steps	Reward	#ECUs	Outcome
Full UCE	22	1.00	11	success
Memory only	50	0.00	1	step-budget timeout
Strategy only	50	0.00	1	step-budget timeout
Workflow only	50	0.00	1	step-budget timeout
Skill only	50	0.00	1	step-budget timeout

Workflow only, ECUs [`wfl_003`]

S1: `go_to(desk 1)` (sees desklamp 1, no book).
S2–3: `go_to(desklamp 1)` returns "Nothing happens",
then `use(desklamp 1)` turns on the lamp with no book in hand.
S4–21: search `drawer 1`–`drawer 6`, `shelf 1`–`shelf 6`, `desk 2` (no book).
S22–36: confused loop, repeatedly tries `go_to(book 1)` returning "Nothing happens" because the book has not yet been located.
S37: `go_to(bed 1)` finally finds *book 1*.
S38: `take(book 1)`.
S39–50: alternates `examine(book 1)` and `use(desklamp 1)` *while standing at bed 1*. Both fail because the agent is not co-located with the desklamp.

Outcome: step-budget exhausted, reward 0.0.

Full UCE, 11 ECUs across all four types (e.g. `str_007, str_020, wfl_003, wfl_006, wfl_008, skl_005, mem_031`)

S1–2: `go_to(desk 2)`, `go_to(desk 1)` (sees desklamp 1).
S3–8: visit `shelf 6`–`shelf 1` (no book).
S9–18: open and inspect `drawer 1`–`drawer 6` (no book).
S19: `go_to(bed 1)` finds *book 1*.
S20: `take(book 1)`.
S21: `go_to(desk 1)` returns to the desklamp.
S22: `use(desklamp 1)` succeeds.

Outcome: task complete in 22 steps, reward 1.0.

The Memory and Strategy ECUs supply two pieces of knowledge that Workflow alone misses: (i) books are commonly placed on `bed` surfaces in this scene type, which lets the agent stop wasting steps on desks and drawers, and (ii) `use(desklamp)` only triggers when the agent is co-located with the lamp, which lets the agent return to `desk 1` before the final action.

Analysis. Look-Book case study on ALFWorld. All four single ECU type ablations exhaust the 50 step budget on this task. Workflow alone correctly recovers the high-level recipe (find the target, take it, use the desklamp on it) but lacks the location prior that books live on beds and the co-location constraint for `use(desklamp)`. Full UCE composes Workflow with Memory, Strategy, and Skill ECUs that supply these missing pieces and solves the task in 22 steps.

Case Study F.2: Mules-Clogs Task (WebShop)

Task instruction: `i am looking for a grey mules & clogs for day comfert, and price lower than 100.00 dollars.`

	Steps	Reward	#ECUs	Outcome
Full UCE	7	1.00	4	success
Memory only	15	0.00	1	invalid-action timeout
Strategy only	15	0.00	3	invalid-action timeout
Workflow only	8	0.67	0	partial: wrong size attribute
Skill only	7	0.67	1	partial: wrong size attribute

Workflow only, no ECU retrieved for this task

S1: `search("grey mules clogs comfort")` returns a product list.
S2: `think`: filter products at exactly $100 versus those under.
S3: `click(B094H755LH)` opens the product page.
S4: `think`: pick the grey colour option.
S5: `click(grey)`.
S6: `click(7)` selects size 7 when the task target is size `7.5 wide`.
S7: `think`: "price under $100, attributes selected, ready to buy".
S8: `click(Buy Now)` returns reward 0.67.

Outcome: correct product and colour, wrong size attribute. Partial reward 0.67.

Workflow alone enforces the search→click→attribute→buy template but provides no attribute-precision rule for matching "size 7.5 wide" rather than the visually similar "size 7".

Full UCE, ECUs `[wfl_001, mem_002, str_040, skl_004]`

S1: `search("grey mules clogs comfort under 100")` returns a product list.
S2: `click(B09P33T3GG)` opens a different candidate that exposes a size+width attribute panel.
S3: `think`: confirm mule/clog style, price under $100, multi-attribute panel present.
S4: `click(100-microsuede-grey)` picks the exact colour name from the panel rather than a generic "grey".
S5: `click(7.5)` picks the exact size-width attribute.
S6: `think`: all required attributes selected, price $32.99 under $100.
S7: `click(Buy Now)` returns reward 1.0.

Outcome: task complete in 7 steps, reward 1.0.

The Memory and Strategy ECUs encode that WebShop tasks often require an *exact* attribute option (colour name, size width) rather than a paraphrased label, which lets Full UCE both choose the right product page and click the correct size variant.

Analysis. Mules-Clogs case study on WebShop. The restricted runs split into two failure modes: Memory and Strategy stall in an invalid-action click loop, while Workflow and Skill complete the procedural template but pick the wrong size attribute and receive partial reward. Full UCE combines all four ECU types and selects the exact size-width variant, completing the purchase in seven steps.

Case Study F.3: Cool-Mug Task (ALFWorld)

Task instruction: `cool some mug and put it in coffeemachine.`

	Steps	Reward	#ECUs	Outcome
Full UCE	18	1.00	11	success
Memory only	50	0.00	1	step-budget timeout (stuck loop)
Strategy only	19	1.00	2	success
Workflow only	19	1.00	1	success
Skill only	50	0.00	1	step-budget timeout

Memory only, ECUs `[mem_002]`

S1–9: open and inspect `cabinet 1`–`cabinet 6` (no mug found).
S10–12: visit `shelf 1`–`shelf 3` (no mug).
S13: `go_to(coffeemachine 1)` (no mug).
S14–16: visit `countertop 1`–`countertop 3` (no mug).
S17–22: open `drawer 1`–`drawer 3` (no mug).
S23: `go_to(sinkbasin 1)` (no mug).
S24–25: `take(cup 1)` (wrong object).
S26–28: cool the cup, try to place it.
S29–30: return the cup to `cabinet 6`.
S31–50: 20 consecutive `go_to(cabinet 6)` → "Nothing happens."

Outcome: step-budget exhausted, reward 0.0.

Full UCE, 11 ECUs across all four types (e.g. `mem_001, wfl_001, str_001, skl_005`)

S1–9: open and inspect `cabinet 1`–`cabinet 6` (no mug found).
S10–12: 3 consecutive `go_to(cabinet 6)` → "Nothing happens."
S13: `go_to(countertop 1)` → finds *mug 1*.
S14: `take(mug 1)`.
S15: `go_to(fridge 1)`.
S16: `cool(mug 1, fridge 1)`.
S17: `go_to(coffeemachine 1)`.
S18: `put(mug 1, coffeemachine 1)` → success.

Outcome: task complete in 18 steps, reward 1.0.

Analysis. Cool mug case study on ALFWorld. All five configurations use the same task instance at their peak cycle. Full UCE finds the mug on `countertop 1` and finishes in 18 steps. Memory only and Skill only exhaust the 50 step budget, while Strategy only and Workflow only solve the task in 19 steps.

Case Study F.4: Pillow-Cover Purchase Task (WebShop)

Task instruction: `i want yellow machine washable batmerry summer bright decorative pillow covers, and price lower than 50.00 dollars.`

	Steps	Reward	#ECUs	Outcome
Full UCE	9	1.00	3	success
Memory only	15	0.00	3	step-budget timeout (Invalid-action loop)
Strategy only	15	0.00	3	step-budget timeout (Invalid-action loop)
Workflow only	15	0.00	1	step-budget timeout (Invalid-action loop)
Skill only	15	0.00	1	step-budget timeout (Invalid-action loop)

Workflow only, ECUs `[wfl_004]`

S1: `search("yellow ... pillow covers")`
→ product list.
S2: `click(B0824CBD9P)`
→ product page.
S3: `think`: confirm product matches description.
S4: `click(Features)`
→ enters sub-tab.
S5: `think`: confirm machine washable.
S6: `click(sunshine yellow)`.
S7: `click(16 x 16 inches)`.
S8 and S9: `click(Buy Now)`
→ "Invalid action!" twice
S10: `think`: "Buy Now is not clickable."
S11: `click(Buy Now)`
→ "Invalid action!"
S12 to S14: three `think` actions summarizing the situation.
S15: `click(Back to Search)`
step budget exhausted.

Outcome: reward 0.0.

Full UCE, ECUs `[wfl_008, mem_057, str_056]`

S1: `search("yellow ... pillow covers")`
→ product list.
S2: `click(B0824CBD9P)`
→ product page.
S3: `click(color yellow)`.
S4: `click(16 x 16 inches)`.
S5: `click(Features)`
→ enters sub-tab.
S6: `think`: confirm machine washable.
S7: `click(Buy Now)`
→ "Invalid action!"
S8: `click(< Prev)`
→ returns to product page.
S9: `click(Buy Now)`
→ score 1.0.

Outcome: task complete in 9 steps, reward 1.0.

Analysis. WebShop pillow cover case study. Workflow only finds the product but remains trapped inside the read only Features tab after invalid Buy Now actions. Full UCE retrieves page mechanism memory and a recovery strategy, returns with `< Prev`, and completes the purchase in nine steps.

G Full Prompts

This appendix lists every prompt UCE uses at runtime. ECU-generating prompts are color-coded by target type: Memory in blue, Strategy in red, Workflow in yellow, and Skill in green. Framework, role, formatting, anatomy, and feedback prompts use the default gray border.

Prompt G.1 shows the anatomy of the agent system prompt. The system message concatenates an environment role description (Prompt G.3 for ALFWorld, Prompt G.4 for WebShop), a fixed framework rules block (Prompt G.2), the four ECU sections in their canonical injection order (Prompt G.5), and the current task description. Few-shot examples are appended afterwards as multi-turn user/assistant/tool sequences. Prompt G.6 reproduces one abbreviated ALFWorld *put* example. Tool actions are exposed as OpenAI function-calling schemas.

Prompt G.7 shows the anatomy of a generator prompt. Each ECU type uses its own template, filled at Generate time with the per-type quality-feedback context (Prompt G.25), the per-environment reshuffle test block, and the collected trajectory text. The Memory generator (Prompt G.8) extracts one reusable rule per trajectory with an outcome-conditioned guidance block (Prompt G.9 on success, Prompt G.10 on failure). The Strategy generator (Prompt G.11) extracts up to three decision rules per call, with an analysis-mode block that adapts to the trajectory mix (Prompt G.12 for mixed batches, Prompt G.13 for failure-only, Prompt G.14 for success-only). The Workflow generator (Prompt G.15) extracts a single standard step-sequence procedure from multiple successful trajectories of the same task type, and the Skill generator (Prompt G.16) extracts one or two cross-task-type sub-operation methods.

The reshuffle test distinguishes environment mechanisms from instance-specific observations and is inserted at the `{reshuffle_block}` placeholder of each generator prompt. Each block contains positive (PASSES) and negative (FAILS) examples drawn from the target benchmark, with the FAILS examples tailored per ECU type. For ALFWorld the blocks are Prompts G.17 to G.20. For WebShop the blocks are Prompts G.21 to G.24.

The quality-feedback context block (Prompt G.25) is prepended to each generator prompt and contains up to three components: a quality reference drawn from the highest- and lowest-fitness-score ECUs of the same type, a library-coverage overview, and a coverage-gap nudge for under-covered task types. The reference component is omitted in the first few cycles when the library is still empty.

Prompt G.1: Agent System Prompt anatomy

[Role and Domain] You are an agent operating in a text-based interactive environment. The task type is `<task_type>`.
[Task Instructions] Read the task description below and act step by step. Use the `think` action to plan, and use the environment-specific tool actions to interact.
[Few-Shot Examples] *(category-specific, e.g. 1 example per WebShop subtype, 2 multi-turn examples per ALFWorld task type)*

Strategic Guidelines
(top-3 retrieved STRATEGY ECUs for the current task)

Task Workflow
(top-3 retrieved WORKFLOW ECUs for the current task)

Relevant Experience
(top-3 retrieved MEMORY ECUs for the current task)

Available Skills
(top-3 retrieved SKILL ECUs, cross-task-type, only injected when adequate skills exist)

[Task Description] `<current task instruction>`
[Initial Observation] `<env reset output>`

Prompt G.2: Framework rules appended to every system prompt

```
- Call exactly ONE tool per turn.
- You may include your reasoning in the message content before calling the
    tool.
```

Prompt G.3: ALFWorld role + agent instructions

```
[ROLE]
You are an expert household agent.

[AGENT INSTRUCTIONS]
Interact with a household environment to complete a given task.

IMPORTANT RULES:
- You must FIRST 'go to' an object/location before you can interact with it
    (open, take, put, use, etc.).
```

Prompt G.4: WebShop role + agent instructions

```
[ROLE]
You are an expert online shopping agent.

[AGENT INSTRUCTIONS]
Interact with an online shopping website to find and purchase a product that
     matches the given instruction.

IMPORTANT RULES:
- You can search for products, click on items to view details, select
    product options (size, color, etc.), and click 'Buy Now' to purchase.
- You must select ALL required options before clicking 'Buy Now'.
- Check that the product matches the instruction's requirements (features,
    price limit, etc.) before purchasing.
- You have a LIMITED step budget per task. Aim to click 'Buy Now' within the
     first 10 steps; do NOT spend many turns re-searching or comparing more
    than 3 products. If the first acceptable match meets the constraints,
    BUY IT --- do not over-explore looking for a marginally better option.
```

Prompt G.5: ECU section headings (injection order)

```
## Strategic Guidelines
- <Strategy ECU 1 content>
- <Strategy ECU 2 content>
- <Strategy ECU 3 content>

## Task Workflow
- <Workflow ECU 1 content>
- ...

## Relevant Experience
- <Memory ECU 1 content>
- ...

## Available Skills
- <Skill ECU 1 content>
- ...
```

Prompt G.6: Abbreviated ALFWorld few-shot example (put task, first of two)

```
[user] [Example]
      You are in the middle of a room. Looking quickly around you, you see
      a cabinet 4, a cabinet 3, ..., a toilet 1, ...
      Your task is to: put some spraybottle on toilet.

[assistant content] To solve the task, I need to find and take a spraybottle
    ,
      then put it on toilet. A spraybottle is more likely to appear in
      cabinet (1-4), countertop (1), toilet (1), sinkbasin (1-2),
      garbagecan (1). I can check one by one, starting with cabinet 1.
[assistant tool_call] go_to({"target": "cabinet 1"})
[tool] On the cabinet 1, you see a cloth 1, a soapbar 1, a soapbottle 1.

[assistant tool_call] go_to({"target": "cabinet 2"})
[tool] The cabinet 2 is closed.

[assistant tool_call] open({"target": "cabinet 2"})
[tool] You open the cabinet 2. ... you see a candle 1, and a spraybottle 2.

[assistant content] Now I find a spraybottle (2). Next, I need to take it.
[assistant tool_call] take({"object": "spraybottle 2", "source": "cabinet
    2"})
[tool] You pick up the spraybottle 2 from the cabinet 2.

[assistant content] Now I take a spraybottle (2). Next, I need to put it
      in/on toilet 1.
[assistant tool_call] go_to({"target": "toilet 1"})
[tool] On the toilet 1, you see a soapbottle 2.

[assistant tool_call] put({"object": "spraybottle 2", "target": "toilet 1"})
[tool] You put the spraybottle 2 in/on the toilet 1.
```

Prompt G.7: Generator Prompt anatomy (one per ECU type)

[Role] You are an experience-extraction assistant for the `<environment>` agent.

Component 1: Type Definition and Output Format
Enforces the role of the target ECU type. For example, MEMORY must output a single factual statement about an environment mechanism, STRATEGY must output IF-THEN conditional rules, WORKFLOW must output a numbered step list using generalized references, SKILL must output a multi-step sub-operation method applicable across task types.

Component 2: Quality Feedback Exemplars
POSITIVE: highest-fitness-score ECU of this type currently in $\mathcal{L}$
NEGATIVE: lowest-fitness-score ECU of this type currently in $\mathcal{L}$
(real fitness-score values shown, guiding the generator toward higher-quality patterns through measured outcomes).

Component 3: Reshuffle Test (mandatory)
Thought experiment: "If all item locations were completely randomized (ALFWorld) or the product catalog were entirely replaced (WebShop), would this rule still hold?"
PASSES (environment mechanism examples)
FAILS (instance-specific observation examples)
The model must argue that each candidate ECU passes the test before emitting it.

[Source Trajectories] `<batch of Phase 2 collected trajectories with rewards>`
[Output Format] JSON list of ECU candidates, each with `when_to_use`, `content`, and `task_type` fields.

Prompt G.8: Memory generation prompt

```
You are extracting a REUSABLE RULE from a {domain_word} trajectory.

The trajectory outcome: {outcome}.

== WHAT TO EXTRACT ==
{outcome_guidance}

{reshuffle_block}

Output a single JSON object:
{
  "when_to_use": "<task type + triggering condition, e.g. 'heat tasks where
      the target object is on a countertop'>",
  "content": "<concrete reusable rule with action sequences, object types,
      or failure conditions>"
}

[Trajectory]:
{trajectory}

Output ONLY the JSON object, no additional text:
```

Prompt G.9: Memory outcome guidance — SUCCESS trajectory

```
This trajectory SUCCEEDED. Extract the most SURPRISING or NON-OBVIOUS
    finding from this experience:
- Any counter-intuitive environment behavior (an action that works without a
     prerequisite you'd expect, or a prerequisite you didn't expect)
- A non-obvious ordering constraint or action precondition
- An unexpected interaction between objects, containers, or appliances
If there is no surprising mechanism, extract the single most critical action
     or condition that determined success --- the one thing another agent is
     most likely to get wrong.
Do NOT output a complete step-by-step task procedure (that belongs to
    workflow). Focus on ONE specific insight.
Generalize from the specific objects to the task TYPE.
```

Prompt G.10: Memory outcome guidance — FAILURE trajectory

```
This trajectory FAILED. Extract WHAT WENT WRONG and THE FIX:
- Identify the critical mistake or wasted steps
- Explain what the agent SHOULD have done instead
- If the agent ran out of steps, identify where it got stuck in a loop
- If the partial score is high (0.5+), the agent found a near-correct item
    but missed a specific attribute --- focus on WHICH attribute was wrong
    and HOW to verify it before purchasing
- If the partial score is 0, the agent never completed a valid purchase ---
    analyze whether it was stuck in navigation, search failure, or a loop
Generalize the fix so it applies to similar tasks, not just this instance.
    Output a SPECIFIC corrective rule, NOT a full task procedure.
```

Prompt G.11: Strategy generation prompt

```
You are analyzing "{task_type}" task trajectories to extract reusable
    decision rules.

{analysis_mode}

== RULES, NOT PROCEDURES ==
Output decision rules that help at SPECIFIC CHOICE POINTS --- "do X instead
    of Y when Z", "skip step W because ...". Do NOT output a complete
    numbered step-by-step task procedure; that belongs to workflow, not
    strategy.

{reshuffle_block}

Output a JSON array of 1-3 rules:
[
  {
    "when_to_use": "<task type + specific triggering condition>",
    "content": "<decision rule with concrete actions or failure conditions>"
  }
]

[Trajectories]:
{trajectories}

Output ONLY the JSON array, no additional text:
```

Prompt G.12: Strategy analysis mode — contrastive (mixed success and failure)

```
You have BOTH successful and failed trajectories. This is ideal for
    contrastive analysis.
For each trajectory pair, find the EARLIEST decision point where the
    successful and failed agents diverged. Explain:
1. What the failed agent did wrong (with exact action)
2. What the successful agent did instead (with exact action)
3. The general rule that explains the difference
```

Prompt G.13: Strategy analysis mode — failure only

```
You have ONLY failed trajectories. Analyze the common failure patterns:
1. Identify repeated mistakes across trajectories
2. Hypothesize the correct approach based on error feedback
3. Extract rules that would prevent these failures
```

Prompt G.14: Strategy analysis mode — success only

```
You have ONLY successful trajectories. Identify the shared NON-OBVIOUS
    choices and counter-intuitive actions that all successful agents
    performed:
1. What actions did successful agents take that a naive agent might skip or
    do differently?
2. Were there any steps where the environment behaved unexpectedly but all
    agents handled it correctly?
3. Generalize into decision rules for this task type
```

Prompt G.15: Workflow generation prompt

```
You are extracting a STANDARD TASK PROCEDURE from multiple successful "{
    task_type}" task trajectories.

== WHAT TO EXTRACT ==
Identify the COMMON step sequence shared across all successful trajectories
    and output it as a numbered procedure template.
- Use generic references: "the target object", "the target receptacle" ---
    NOT specific object names or locations from any single trajectory
- Include ONLY steps that appear in ALL (or nearly all) successful
    trajectories --- omit exploratory detours or recovery steps unique to
    one trajectory
- For each step where the environment behaves counter-intuitively, add a
    parenthetical note explaining the mechanic (e.g. "Heat target with
    microwave 1 (works even when closed --- no need to open first)")

== WHAT NOT TO EXTRACT ==
- Do NOT output conditional decision rules like "if X then Y instead of Z"
    --- those belong to strategy, not workflow
- Do NOT specify which locations to search or in what order --- search paths
     vary per game instance
- Do NOT include error-recovery or backtracking steps --- output the HAPPY
    PATH only

{reshuffle_block}

Output a single JSON object:
{
  "when_to_use": "<task type description, e.g. 'heat tasks --- heating an
      object and placing it at a target location'>",
  "content": "<numbered step-by-step procedure with parenthetical mechanism
      notes where relevant>"
}

[Successful "{task_type}" Trajectories]:
{trajectories}

Output ONLY the JSON object, no additional text:
```

Prompt G.16: Skill generation prompt

```
You are identifying REUSABLE SUB-TASK METHODS that appear across multiple
    different {domain_word} task types.

The trajectories below come from DIFFERENT task types ({task_types}). Find
    operation patterns that recur across these types --- methods for sub-
    problems that any task might encounter.

== WHAT TO EXTRACT ==
Identify a sub-operation that appears in >= 2 different task types and
    describe HOW to execute it reliably.
- Focus on sub-task-level methods: searching for objects, handling
    containers, recovering from failed actions --- not full task procedures
- The method must be TASK-TYPE-AGNOSTIC: equally useful in {task_type_list}
    tasks
- Describe the method as a multi-step protocol, not as a single IF-THEN
    decision rule

== WHAT NOT TO EXTRACT ==
- Do NOT output a complete task procedure like "1. Find 2. Take 3. Heat 4.
    Put" (that belongs to workflow, not skill)
- Do NOT output a single-point decision rule like "if X then Y" (that
    belongs to strategy, not skill)
- Do NOT output an environment mechanism fact like "fridge doesn't need
    opening" (that belongs to memory, not skill)

{reshuffle_block}

Output a JSON array of 1-2 methods:
[
  {
    "when_to_use": "<functional trigger, e.g. 'when you need to locate a
        specific object in the environment'>",
    "content": "<multi-step method description>"
  }
]

[Trajectories from different task types]:
{trajectories}

Output ONLY the JSON array, no additional text:
```

Prompt G.17: ALFWorld reshuffle test — Memory

```
== RESHUFFLE TEST (mandatory) ==
Before outputting, verify your rule passes this test:
"If ALL objects were placed in completely different locations in a new game
    instance, would this rule still be correct and helpful?"

PASSES (environment MECHANISM --- true across all game instances):
- "The 'cool X with fridge 1' action succeeds even when fridge is closed ---
     no need to 'open fridge 1' first."
- "In 'examine' tasks, the task completes when you 'use desklamp' while
    holding the target object --- the trigger is 'use', not 'examine'."
- "'clean X with sinkbasin 1' is a single action --- no need to put X in
    sinkbasin then take it back out."

FAILS (instance-specific OBSERVATION --- wrong in a different game):
- "Potato 1 is on countertop 1" --- objects are reshuffled each game.
- "Cabinet 6 cannot accept items" --- cabinet behavior varies per game.
- "Mug 2 is on cabinet 4" --- specific locations change every game.

ONLY output rules that PASS the reshuffle test.
```

Prompt G.18: ALFWorld reshuffle test — Strategy

```
== RESHUFFLE TEST (mandatory) ==
Every rule must pass: "If all objects were in different locations in a new
    game, would this rule still be correct and helpful?"

PASSES (environment mechanism --- decision rules):
- "'cool X with fridge 1' works even when fridge is closed --- do NOT waste
    steps opening the fridge before cooling."
- "'heat X with microwave 1' works even when microwave is closed --- skip
    the 'open microwave' step."
- "In 'examine' tasks, the completion trigger is 'use desklamp', NOT '
    examine [object]'. Attempting to examine the object directly will not
    complete the task."

FAILS (DO NOT output):
- "The egg is in the fridge" / "Cabinet 3 contains a mug" (instance-specific
    )
- "Explore efficiently" / "plan ahead" (generic, any LLM knows this)
- "1. Find object 2. Take it 3. Heat it 4. Put it" (step-by-step procedure
    --- belongs to workflow, not strategy)
- "Search method: visit each receptacle, open closed ones, check contents,
    move to next" (sub-task method --- belongs to skill, not strategy)
```

Prompt G.19: ALFWorld reshuffle test — Workflow

```
== RESHUFFLE TEST (mandatory) ==
Verify: "If all objects were renamed and placed in completely different
    locations, would this step sequence still be the correct procedure for {
    task_type} tasks?"

PASSES (generalised procedure with mechanism notes):
- "Heat: 1. Find the target object (search receptacles until found) 2. Take
    it 3. Go to microwave 1 4. Heat target with microwave 1 (works even when
     closed) 5. Go to target receptacle 6. Put target in/on target
    receptacle"
- "Examine: 1. Find the target object 2. Take it 3. Go to desklamp 1 4. Use
    desklamp 1 (task completes on 'use' while holding the object --- '
    examine' alone does NOT complete it)"

FAILS (DO NOT output):
- "1. Go to countertop 1 2. Take potato 1 3. Go to microwave 1 ..." --- uses
     specific object names and locations from one game instance
- "1. Find the object 2. Do the required action 3. Place it" --- too vague
    to be actionable
```

Prompt G.20: ALFWorld reshuffle test — Skill

```
== RESHUFFLE TEST (mandatory) ==
Verify: "Would this method work identically in a completely different game
    instance with different object placements?"

PASSES (cross-type reusable method):
- "Systematic object search: Visit each receptacle in the room. For closed
    containers (cabinet/drawer), open before checking contents. When the
    target appears in the observation, immediately take it. If 'take'
    returns 'Nothing happens', move to the next receptacle."
- "Failure recovery for 'Nothing happens': (1) Verify co-location by doing '
    go to <target>'. (2) If the target is a container, open it. (3) Retry
    the action once. (4) If still failing, switch to a different instance or
     location."

FAILS (DO NOT output):
- "Heat task: 1. Find object 2. Take 3. Microwave 4. Put" (task-level
    procedure --- belongs to workflow)
- "If 'put' returns Nothing happens, open the container" (single IF-THEN
    rule --- belongs to strategy)
- "The fridge doesn't need to be opened for cooling" (environment fact ---
    belongs to memory)
```

Prompt G.21: WebShop reshuffle test — Memory

```
== RESHUFFLE TEST (mandatory) ==
Before outputting, verify your rule passes this test:
"If the entire product catalog changed (different products, prices, and
    options), would this rule still be correct and helpful?"

PASSES (environment MECHANISM --- true across all shopping sessions):
- "You must select ALL required options (size, color, etc.) before clicking
    'Buy Now', or the purchase will receive a score of 0."
- "The 'think[...]' action does not change the page state --- use it freely
    to plan without consuming navigation steps."
- "Clicking 'Back to Search' from any product page returns to the initial
    search page, allowing you to start a completely new query."

FAILS (instance-specific OBSERVATION --- wrong with different products):
- "B078GWRC1J is the best match for deodorant" --- product IDs and inventory
     change across sessions.
- "The first search result is always the cheapest" --- result ordering
    varies by query and session.
- "Earth Mama deodorant costs $10.99" --- prices are session-specific.

ONLY output rules that PASS the reshuffle test.

== BUDGET CONSTRAINT ==
Each episode has a strict per-episode step budget. Memory ECUs that add
    verification clicks consume budget; memory ECUs that warn against
    unsalvageable states save budget.

PREFER rules of the form "Treat <state> as <non-purchasable / stale>; do NOT
     do <action> from this state":
- "Treat Attributes/Description/Features sub-tabs as read-only views; 'Buy
    Now' from these tabs returns Invalid Action. Use '< Prev' to return to
    the main product page before purchasing."
- "Treat the generic [Search]-only screen as a reset state; do NOT click
    product IDs or variants here. Click 'Search' to reload the result list."

AVOID rules that require additional verification clicks before Buy Now:
- "Before purchasing, verify every constraint by opening Description,
    Features, Attributes, Reviews." Each sub-tab click is a step; this adds
    4 steps before the actual purchase.
- "Always read at least three Reviews before deciding." Each review click is
     a step.

If verification is necessary, encode it as a think[...] check rather than as
     additional click actions.
```

Prompt G.22: WebShop reshuffle test — Strategy

```
== RESHUFFLE TEST (mandatory) ==
Every rule must pass: "If the product catalog changed completely, would this
     rule still be correct and helpful?"

PASSES (environment mechanism --- decision rules):
- "If the initial search returns irrelevant results, click 'Back to Search'
    and try a shorter, more specific query rather than browsing multiple
    pages of poor results."
- "Always verify the item's price against the budget constraint in the
    instruction BEFORE clicking 'Buy Now' --- buying an over-budget item
    scores 0."
- "When a product has required options (color, size, etc.), select ALL of
    them before clicking 'Buy Now'. Missing any option results in a failed
    purchase."

FAILS (DO NOT output):
- "Search for 'bright citrus deodorant' to find the target product" (
    instance-specific query)
- "Always pick the cheapest option" (oversimplified, ignores feature
    matching)
- "1. Search 2. Click product 3. Select options 4. Buy" (step-by-step
    procedure --- belongs to workflow, not strategy)
- "Check every product on every search page systematically" (sub-task method
     --- belongs to skill, not strategy)

== BUDGET CONSTRAINT ==
Each episode has a strict per-episode step budget. Strategy ECUs that delay
    commitment by adding verification or comparison clicks consume budget;
    strategies that detect a bad state and immediately abandon save budget.

PREFER strategies of the form "IF <bad signal> THEN <abandon / reset / move
    on>":
- "IF clicking 'Buy Now' returns 'Invalid action!' AND you have already
    retried once from this listing, THEN abandon this listing and open a
    different search result."
- "IF a click on a result ASIN does not navigate to a product page, THEN do
    not retry the same ASIN; click 'Search' to reload the result list."

AVOID strategies that require additional verification or comparison clicks
    before Buy Now:
- "Always click 'Description' and 'Features' to confirm the product before
    Buy Now." This consumes 2 sub-tab clicks per attempt.
- "Compare at least three candidate products by opening each one before
    deciding." Multi-product comparison consumes 6+ steps.

If a strategy must guard against a constraint, encode the check as a think
    [...] verification rather than as additional click actions.
```

Prompt G.23: WebShop reshuffle test — Workflow

```
== RESHUFFLE TEST (mandatory) ==
Verify: "If the product catalog changed completely, would this step sequence
     still be the correct procedure for shopping tasks?"

PASSES (generalised procedure with mechanism notes):
- "Shopping: 1. Search with keywords from the instruction 2. Identify a
    matching product from results (check title + price) 3. Click the product
     ASIN to view details 4. Select all required options (color, size, etc.)
     5. Verify price is within budget 6. Click 'Buy Now' to complete
    purchase"

FAILS (DO NOT output):
- "1. Search 'deodorant' 2. Click B078GWRC1J 3. Select 'bright citrus' 4.
    Click 'Buy Now'" --- uses specific product IDs and options from one
    session
- "1. Search 2. Buy the first result" --- too vague to be actionable
```

Prompt G.24: WebShop reshuffle test — Skill

```
== RESHUFFLE TEST (mandatory) ==
Verify: "Would this method work identically with a completely different
    product catalog?"

PASSES (cross-session reusable method):
- "Efficient product evaluation: Read the title and price of each search
    result. Skip products that clearly don't match (wrong category, over
    budget, missing key features). Only click on products whose title
    partially matches the instruction to check their detailed options and
    price."
- "Search query refinement: If the first search returns irrelevant results,
    go back and try (1) fewer keywords focusing on the core product type,
    (2) different synonyms, or (3) brand name if mentioned in the
    instruction."

FAILS (DO NOT output):
- "Shopping procedure: Search -> Click -> Select -> Buy" (task-level
    procedure --- belongs to workflow)
- "If price exceeds budget, go back and find cheaper option" (single IF-THEN
     rule --- belongs to strategy)
- "'Buy Now' completes the task and reveals the score" (environment fact ---
     belongs to memory)

== BUDGET CONSTRAINT ==
Each episode has a strict per-episode step budget. Skill ECUs that ask the
    agent to "scan all variants" or "open every sub-tab to verify" consume
    budget; recovery protocols whose terminal step is "abandon and try next
    listing" save budget.

PREFER recovery / navigation skills whose terminal step is "abandon and move
     to the next candidate" or "reset to a known-good view in ONE step":
- "Invalid-action recovery: 1) identify which view you are on; 2) return to
    the main product page in ONE step (use '< Prev' or re-click the product
    from search results); 3) re-try the action ONCE from the restored view;
    4) if it still fails, abandon this listing and open a different search
    result."
- "Stale-state escape: when the page becomes a generic [Search] control with
     no result list, click 'Search' once to reload; if the result list does
    not return after one reload, issue a fresh search query rather than
    continuing to click."

AVOID skill protocols that require an analysis pass before any action:
- "Scan all variant selectors; list which are constrained vs free; locate
    each required value; click them in a stable order before 'Buy Now'."
    This consumes 4-6 steps before the first commitment.
- "Compare top-3 search results by opening each, recording prices and option
     ranges, then return to the best candidate." Multi-product comparison
    protocols inflate the step count beyond the per-episode budget on most
    environments.

If a skill must verify constraints, encode the verification as a think[...]
    step inside the protocol rather than as additional click actions.
```

Prompt G.25: Quality-feedback context template (one block per generator call)

```
== QUALITY REFERENCE (from previously generated ECUs) ==
HIGH quality (fitness={best_fitness}, used {best_usage_count} times):
  when_to_use: "{best_when_to_use}"
  content: "{best_content}"
  Why good: Captures a specific, counter-intuitive mechanism that the agent
      cannot learn from few-shot examples alone.

LOW quality (fitness={worst_fitness}, used {worst_usage_count} times):
  when_to_use: "{worst_when_to_use}"
  content: "{worst_content}"
  Why bad: Too generic or duplicates information already available in the
      few-shot demonstrations.

Generate an ECU that is more like the HIGH quality example.

== CURRENT LIBRARY STATUS ==
  memory: {n_memory} ECUs ({memory_task_types})
  strategy: {n_strategy} ECUs ({strategy_task_types})
  workflow: {n_workflow} ECUs ({workflow_task_types})
  skill: {n_skill} ECUs ({skill_task_types})

Gap for {ecu_type}: task types {uncovered_task_types} have no coverage yet.
    Prioritize generating knowledge for uncovered types if the trajectory
    data supports it.
```

H Statement on AI Assistance

We used general-purpose AI assistants for coding support (completion, refactoring, debugging) and for sentence-level writing polish. All algorithmic design, experimental decisions, results, and analyses are the authors' own.