RETTT: REFLECTION-DRIVEN TEST-TIME TRAINING VIA EXPERIENCE-BASED PARAMETRIC MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents accumulate valuable task-specific information through their actions, observations, and environmental feedback during multi-turn task execution. When effectively used, this information can help agents refine subsequent decisions and improve task performance. However, the learning value of an interaction depends on its context and subsequent feedback, making it challenging to extract reliable, actionable lessons and revise them as the task unfolds. In this paper, we introduce ReTTT, which enables experience-driven self-adaptation through test-time training on lessons constructed and continually revised within an episode. ReTTT lets the agent reflect on interaction feedback, maintain the resulting lessons in an explicit memory, and learn from them by updating an episode-local LoRA adapter while keeping the backbone frozen. The adapted model then performs both subsequent actions and reflection, coupling policy adaptation with the construction of future learning experience. We evaluate ReTTT on ALFWorld, ScienceWorld, and SWE-bench Lite using two model families. ReTTT outperforms trajectory-based test-time training across all six model–benchmark settings, improving task success by up to 12.14 percentage points. A frozen-reflection comparison further shows a 7.14-point benefit from adapting the reflection policy, while trajectory analysis illustrates how adapted reflection derives more task-specific guidance from failed attempts. These findings suggest that learning to reflect on its own behavior can boost agents' performance on long-horizon tasks.

1 INTRODUCTION

Modern large language model (LLM) agents (Liu et al., 2024) tackle multi-turn tasks, such as software development (Yang et al., 2024; Jimenez et al., 2024), web navigation (Zhou et al., 2024), and scientific research (Wang et al., 2022), through an iterative agent loop. The agent alternates between reasoning, taking actions, and receiving environmental feedback, accumulating a trajectory of plans, intermediate reasoning, actions, and observations in its context (Yao et al., 2022). This trajectory records action outcomes and task progress, providing task-specific experience that can inform subsequent decisions and learning.

A substantial body of work investigates how agents can make better use of this interaction history. Context management methods organize, compress, and retrieve historical information to guide subsequent actions through the input context while keeping model parameters fixed (Park et al., 2023; Packer et al., 2023; Zhang et al., 2026; Mi et al., 2026). Parametric memory approaches consolidate accumulated experience into learned parameters, allowing information from past trajectories to shape the agent's policy beyond the immediate context (Yao et al., 2026; Ren et al., 2026). Recently, agentic test-time training (aTTT) has brought parameter adaptation into ongoing task execution, using signals from agent outputs, environmental observations, or generated progress summaries (Wang et al., 2026). It downweights repeated token spans to limit redundant self-training while retaining novel information. Each update changes the policy that acts next and thereby shapes the interaction history available for subsequent learning.

However, continual parameter updates leave a key question: what should an agent learn from its ongoing interactions? An interaction trajectory records what happened, but useful supervision must capture what should guide future behavior. Failure feedback, for example, can turn an unsuccessful action into a lesson about prerequisites for subsequent attempts(Figure 1a). New observations can

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

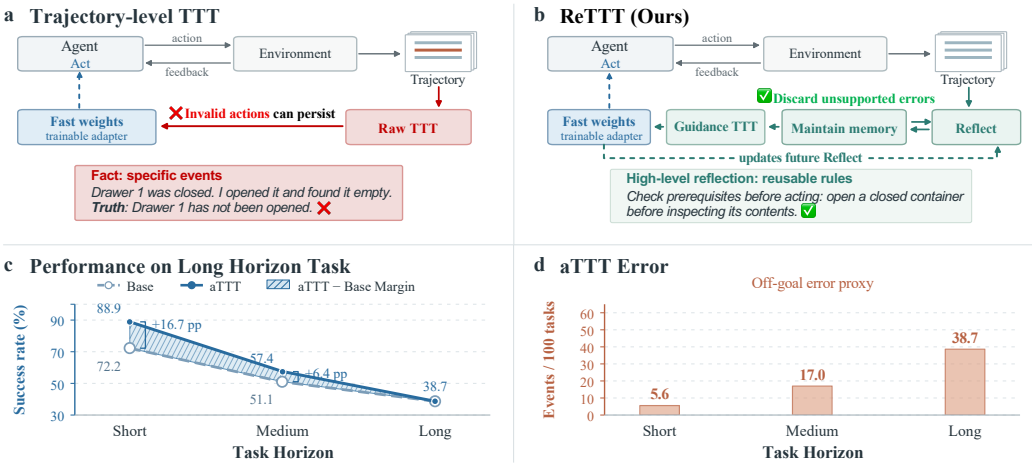


Figure 1: **Motivation: interaction trajectories do not directly provide reliable supervision.** **(a)** Trajectory-level TTT internalizes raw experience, allowing unsupported errors to persist. **(b)** ReTTT filters procedural guidance before internalization and adapts reflection through shared fast weights. **(c)** aTTT's gain over Base decreases from +16.7 pp to +6.4 pp and 0 with longer horizons. **(d)** aTTT's off-goal error proxy increases from 5.6 to 17.0 and 38.7.

also revise earlier conclusions, requiring the supervision used for later updates to change as the task unfolds. The challenge is therefore to continually derive reliable, decision-relevant lessons from the trajectory and use them to adapt the policy within the same episode.

To investigate the limitations of agentic test-time training over longer task horizons, we analyze aTTT's performance and execution errors. Its improvement over the frozen policy falls from 16.7 percentage points on short tasks to 6.4 on medium tasks and disappears on long tasks (Figure 1c). Meanwhile, aTTT's off-goal error proxy, measured as placements of target objects at non-goal locations per 100 tasks, rises from 5.6 to 17.0 and 38.7 across these groups (Figure 1d). These findings suggest that accumulating more interaction data does not necessarily provide better learning supervision, motivating the explicit construction and revision of task-relevant lessons before parameter updates.

To address these challenges, we introduce ReTTT(Figure 1b), which couples the construction and revision of learning experience with policy adaptation through test-time training. The same adapted model acts in the environment and constructs the lessons used for later updates within the ongoing episode. At periodic reflection steps, it interprets interaction feedback to extract task-relevant lessons, including action prerequisites, failure conditions, and guidance for remaining subgoals. These lessons are checked against interaction evidence and maintained in an explicit memory that is revised as new information arrives. New and revised lessons, together with selected prior experience, are then used to update an episode-local LoRA adapter through test-time training, while the backbone remains frozen. We instantiate these lessons as question–answer pairs (ReTTT-QA) and structured summaries (ReTTT-Sum).

We evaluate ReTTT on ALFWorld, ScienceWorld, and SWE-bench Lite using Qwen3.5-9B and Gemma4-12B. Both ReTTT-QA and ReTTT-Sum outperform aTTT in all six model–benchmark settings (Table 1). With Qwen3.5-9B, ReTTT-QA improves success over aTTT by 12.14, 11.67, and 3.33 percentage points on the three benchmarks, respectively. On ALFWorld, ReTTT-QA's advantage over aTTT grows from 5.6 percentage points on short tasks to 14.7 on long tasks, grouped by reference action count (Table 2).

Further analysis examines how adapting the reflection policy contributes to these improvements. On ALFWorld with Qwen3.5-9B, ReTTT-QA achieves 63.57% success, compared with 56.43% when the reflection policy remains frozen while action-policy adaptation is retained (Section 3.4). A trajectory-level example illustrates the difference: with the same interaction history, the adapted reflector derives task-specific search guidance from failed attempts, while the frozen reflector repeats generic advice (Figure 4). These observations suggest that adapting reflection helps the agent derive

more actionable lessons from its experience, supporting continued progress during long-horizon task execution.

Our contributions are threefold:

- We identify a mismatch between interaction history and useful learning supervision in agentic test-time training. Quantitative comparisons examine the learning signals used during adaptation and their relationship to reliable long-horizon task execution.
- We introduce ReTTT, which enables experience-driven self-adaptation through test-time training on lessons distilled and continually revised during task execution, with the adapted model responsible for both actions and subsequent reflection.
- Experiments across three benchmarks and two model families demonstrate higher task success than aTTT for both ReTTT variants. Frozen-reflection comparisons and trajectory analysis further provide evidence for the benefit of adapting the policy that constructs learning experience.

2 METHOD

Figure 2 illustrates the data flow of ReTTT, and Algorithm 1 summarizes the complete in-episode procedure. The agent interacts with the environment using episode-local fast weights and periodically performs three adaptation stages: it reflects on the observed trajectory to construct procedural candidates, maintains these candidates in a bounded procedural memory, and internalizes newly acquired procedures together with replayed prior experience. The backbone remains frozen throughout the episode, while action and reflection use two prompting interfaces to the same adapted model.

2.1 PROBLEM FORMULATION

We consider a long-horizon task as a partially observed interactive process. At environment step t, the agent observes o_t, selects an action a_t, and receives the next observation o_{t+1} together with feedback r_t. The interaction history before selecting a_t is

$$h_t = (o_1, a_1, r_1, \ldots, a_{t-1}, r_{t-1}, o_t). \tag{1}$$

We use k to index adaptation events and let t_k denote the environment step at which event k occurs. The backbone parameters θ remain frozen throughout an episode. ReTTT maintains episode-local fast weights ϕ_k, implemented as a lightweight trainable adapter, together with an explicit procedural memory $\mathcal{M}_k$. Between adaptation events, actions are generated by

$$a_t \sim \pi_{\theta,\phi_k}(\cdot \mid g, h_t), \tag{2}$$

where g denotes the task specification and ϕ_k is the current fast-weight state.

Action generation and reflection are two prompting interfaces to the same adapted language model and therefore share θ and ϕ_k. The explicit memory $\mathcal{M}_k$ is supplied only to the reflection interface; action generation conditions on the interaction history and the procedural knowledge already internalized in ϕ_k. Both $\mathcal{M}_k$ and ϕ_k are reset at the end of each episode.

2.2 ADAPTED-POLICY REFLECTION

At adaptation event k, ReTTT constructs a structured reflection context from the task specification, the observed interaction history, and the current procedural memory:

$$c_k^{\mathrm{ref}} = \mathrm{Reflect}\left(g, h_{t_k}, \mathcal{M}_k; \mathcal{S}_{\mathrm{ref}}\right), \tag{3}$$

where $\mathcal{S}_{\mathrm{ref}}$ denotes the reflection prompt. The current adapted model then generates a set of candidate procedural experiences:

$$\mathcal{Z}_k = \{z_{k,i}\}_{i=1}^{m_k} \sim R_{\theta,\phi_k}\left(\cdot \mid c_k^{\mathrm{ref}}\right). \tag{4}$$

The reflection organizes interaction experience into four procedural categories. **Progress** records completed state transitions or subgoals supported by environment feedback. **Failure** records unsuccessful or inadmissible transitions and their observed conditions. **Remaining subgoals** captures

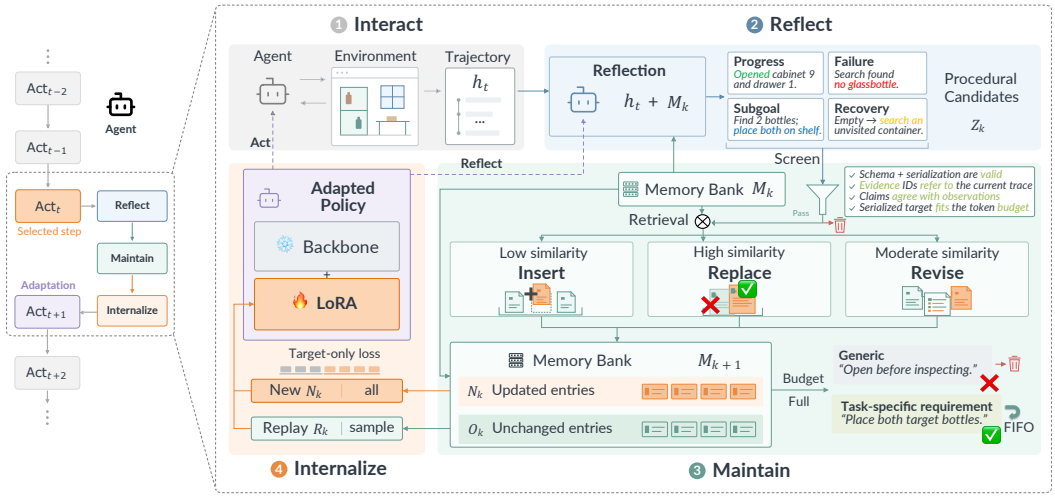


Figure 2: **Overview of ReTTT. (1) Interact:** the adapted policy acts in the environment. **(2) Reflect:** interaction history is converted into procedural candidates $\mathcal{Z}_k$. **(3) Maintain:** candidates are screened and reconciled with memory, producing changed entries $\mathcal{N}_k$ and unchanged entries $\mathcal{O}_k$. **(4) Internalize:** $\mathcal{N}_k$ and replay $\mathcal{R}_k \subseteq \mathcal{O}_k$ update episode-local fast weights while the backbone remains frozen.

unresolved requirements and missing prerequisites. **Recovery** specifies reusable actions or recovery strategies. Each item is accompanied by supporting trajectory evidence. Accordingly, a candidate is represented as

$$z_{k,i} = (\kappa_{k,i}, u_{k,i}, E_{k,i}), \qquad \kappa_{k,i} \in \{\textsc{Progress}, \textsc{Failure}, \textsc{Subgoal}, \textsc{Recovery}\}, \tag{5}$$

where $u_{k,i}$ is the procedural content and $E_{k,i}$ contains its supporting trajectory evidence. At this stage, $\mathcal{Z}_k$ contains only candidate experience; admission to memory is determined by the maintenance stage.

We instantiate the same procedural schema using two textual serializations. In the QA representation, each item is written as a decision-relevant question, answer, and supporting evidence:

$$z^{\mathrm{QA}}_{k,i} = \left(q_{k,i}, a^{\mathrm{ans}}_{k,i}, E_{k,i}\right). \tag{6}$$

In the Summary representation, the same information is organized as a structured checkpoint containing progress, failures, remaining subgoals, procedures, and supporting evidence.

2.3 Procedural Memory Maintenance

The candidate set $\mathcal{Z}_k$ is processed before it can be used for parameter updates. ReTTT first screens the candidates for structural validity and consistency with the observed trajectory, discarding unsupported or malformed experience before memory integration. The surviving candidates are then reconciled with the current memory $\mathcal{M}_k$:

$$\mathcal{M}_{k+1} = \mathrm{Maintain}\left(\mathcal{M}_k,\ \mathrm{Screen}(\mathcal{Z}_k; h_{t_k})\right). \tag{7}$$

Candidate screening. A candidate is retained only if it follows the expected procedural schema, its supporting evidence refers to observations in the current trajectory, and its content is consistent with the observed interaction and task outcome. Candidates that fail these checks are discarded before they can enter memory or contribute to parameter updates. We denote the surviving set by

$$\widetilde{\mathcal{Z}}_k = \{z \in \mathcal{Z}_k \mid \mathrm{Valid}(z, h_{t_k}) = 1\}, \tag{8}$$

where Valid applies deterministic schema, evidence-grounding, trajectory-consistency, and serialization checks. Candidates whose serialized targets exceed the allowed memory-token budget are also discarded.

Algorithm 1 ReTTT: In-Episode Procedural Adaptation

Require: Task g, frozen backbone θ, fast weights ϕ_0, replay budget $r_{\max}$

$\mathcal{M}_0 \leftarrow \emptyset,\ k \leftarrow 0$
for $t = 1, 2, \ldots$ until termination **do**
  $a_t \sim \pi_{\theta,\phi_k}(\cdot \mid g, h_t)$
  Execute a_t and update h_{t+1}
  if t is an adaptation point **then**
    $\mathcal{Z}_k \sim R_{\theta,\phi_k}(\cdot \mid g, h_{t+1}, \mathcal{M}_k)$
    $\widetilde{\mathcal{Z}}_k \leftarrow \mathrm{Screen}(\mathcal{Z}_k; h_{t+1})$
    $(\mathcal{M}_{k+1}, \mathcal{N}_k) \leftarrow \mathrm{Maintain}(\mathcal{M}_k, \widetilde{\mathcal{Z}}_k)$
    $\mathcal{O}_k \leftarrow \mathcal{M}_{k+1} \setminus \mathcal{N}_k$
    if $\mathcal{N}_k \neq \emptyset$ **then**
      $r_k \leftarrow \min(r_{\max}, |\mathcal{O}_k|)$
      $\mathcal{R}_k \sim \mathrm{Sample}(\mathcal{O}_k, r_k)$
      $\mathcal{D}_k \leftarrow \mathcal{N}_k \cup \mathcal{R}_k$
      $\phi_{k+1} \leftarrow \mathrm{TTT}(\phi_k; \mathcal{D}_k, \theta)$
    else
      $\phi_{k+1} \leftarrow \phi_k$
    end if
    $k \leftarrow k + 1$
  end if
end for

Memory reconciliation. For each surviving candidate, ReTTT computes its similarity to existing memory entries based on normalized procedural type and decision-relevant content. If the highest similarity is low, the candidate is inserted as a new entry. If it is high, the most similar entry is replaced by the newly observed version. For moderate similarity, the existing entry is revised by merging the old and new information. Entries unaffected by the current candidates remain unchanged.

After reconciliation and memory-budget enforcement, we denote by $\mathcal{N}_k$ the entries in $\mathcal{M}_{k+1}$ that were inserted, replaced, or revised at event k. The remaining unchanged entries are

$$\mathcal{O}_k = \mathcal{M}_{k+1} \setminus \mathcal{N}_k. \tag{9}$$

Memory bounds. The procedural memory is constrained by

$$|\mathcal{M}_{k+1}| \leq B, \qquad \sum_{m \in \mathcal{M}_{k+1}} \tau(m) \leq T, \tag{10}$$

where B is the entry budget, $\tau(m)$ is the number of target tokens associated with entry m, T is the total target-token budget. When either budget is exceeded, generic procedures are evicted before task-specific ones, with remaining ties resolved by First-In-First-Out (FIFO) order.

2.4 Test-Time Procedural Internalization

At adaptation event k, ReTTT trains on all newly inserted or modified procedures in $\mathcal{N}_k$ together with a bounded replay subset from the unchanged memory $\mathcal{O}_k$. We set

$$r_k = \min\left(r_{\max}, |\mathcal{O}_k|\right), \qquad \mathcal{R}_k \sim \mathrm{Unif}\left(\{\mathcal{R} \subseteq \mathcal{O}_k \mid |\mathcal{R}| = r_k\}\right), \tag{11}$$

where $r_{\max}$ is the replay budget, r_k is the number of unchanged procedures replayed at event k, and $\mathcal{R}_k$ is sampled uniformly from all subsets of $\mathcal{O}_k$ of size r_k. We then construct the training set

$$\mathcal{D}_k = \mathcal{N}_k \cup \mathcal{R}_k, \tag{12}$$

so that every newly inserted or modified procedure is used for adaptation, while only a bounded number of unchanged procedures are replayed. If $\mathcal{N}_k = \emptyset$, no parameter update is performed.

Each $(x_i, y_i) \in \mathcal{D}_k$ is serialized as a procedural training example, where x_i is the procedural query or checkpoint context and y_i is the corresponding evidence-supported target. Only target tokens contribute to the test-time objective:

$$\mathcal{L}_{\mathrm{ReTTT}}(\phi_k) = -\sum_{(x_i,y_i)\in\mathcal{D}_k} \sum_{j=1}^{|y_i|} \log p_{\theta,\phi_k}\left(y_{i,j} \mid x_i, y_{i,<j}\right). \tag{13}$$

Table 1: Task success (%), averaged over three seeds. ALFWorld uses 140 tasks, ScienceWorld 120 instances, and SWE-bench Lite 60 instances.

	ALFWorld		ScienceWorld		SWE-bench Lite	
Method	**Qwen**	**Gemma**	**Qwen**	**Gemma**	**Qwen**	**Gemma**
Baselines						
Base	47.14	52.86	10.83	7.50	33.33	8.33
qTTT	50.00	54.29	11.67	11.67	36.67	10.00
aTTT	51.43	53.57	15.00	10.00	40.00	11.67
ReTTT						
ReTTT-QA	**63.57**	57.86	**26.67**	15.83	**43.33**	18.33
ReTTT-Sum	60.00	**59.29**	23.33	**16.67**	**43.33**	**20.00**

The episode-local fast weights are then updated by

$$\phi_{k+1} = \phi_k - \eta \nabla_\phi \mathcal{L}_{\mathrm{ReTTT}}(\phi_k), \tag{14}$$

while the backbone parameters θ remain frozen. Raw trajectory tokens are never directly optimized as targets; parameter updates are driven only by the maintained procedural experience in $\mathcal{D}_k$.

These design choices separate experience construction from parameter write-back. Only screened and maintained procedural experience receives gradient updates, while the explicit memory remains directly revisable across adaptation events. Because action generation and reflection generation share the same fast weights ϕ_k, each update changes both subsequent behavior and the policy that constructs later procedural experience. ReTTT therefore forms a closed in-episode adaptation loop in which maintained procedural experience is progressively internalized into the policy that continues the task.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate on three long-horizon interactive benchmarks: ALFWorld (Shridhar et al., 2020) for embodied household tasks, ScienceWorld (Wang et al., 2022) for multi-step scientific procedures, and SWE-bench Lite (Jimenez et al., 2024) for repository-level software engineering. We use all 140 ALFWorld tasks, first 120 ScienceWorld instances, and first 60 SWE-bench Lite instances. The subsets are determined solely by the benchmark ordering and are not selected or filtered based on model performance.

Models and adaptation. We evaluate Qwen3.5-9B (Qwen Team, 2026) and Gemma4-12B (Team et al., 2026). The backbone remains frozen within each episode, and test-time updates are restricted to lightweight LoRA fast weights shared by action and reflection generation.

Baselines. We compare against Base, the frozen agent without parameter updates; qTTT (Bansal et al., 2026), which performs query-only test-time training; and aTTT (Wang et al., 2026), which continually trains on agent trajectories. Our two variants, ReTTT-QA and ReTTT-Sum, differ in whether procedural experience is serialized as QA-style lessons or structured progress summaries.

Implementation details. We use a procedural-memory capacity of 12, reflect every 5 interaction steps, replay up to 2 previous procedures per update, and learning rate is 5×10^{-4}. Additional implementation details are provided in Appendix A.2.

3.2 Main Results

Table 1 reports task success averaged over three seeds. Across all three benchmarks and both backbones, both ReTTT variants outperform aTTT, and the best ReTTT variant achieves the strongest result in every model–benchmark setting.

ALFWorld. With Qwen3.5-9B, ReTTT-QA improves success from 51.43% with aTTT to 63.57%; with Gemma4-12B, ReTTT-Sum improves from 53.57% to 59.29%.

Table 2: ALFWorld horizon-stratified results with Qwen3.5-9B. Each cell reports **success (%) / steps**.

Method	Short	Medium	Long	Overall
Baselines				
Base	72.22 / 20.50	51.06 / 30.87	38.67 / 39.52	47.14 / 34.17
qTTT	83.33 / 15.72	59.57 / 29.43	36.00 / 39.93	50.00 / 33.29
aTTT	88.89 / 17.33	57.45 / 28.47	38.67 / 40.20	51.43 / 33.32
ReTTT				
ReTTT-QA	94.44 / 13.06	**68.09 / 26.57**	**53.33 / 33.92**	**63.57 / 28.77**
ReTTT-Sum	**100.00 / 9.33**	65.96 / 27.36	46.67 / 36.69	60.00 / 30.04

We further stratify ALFWorld tasks by reference low-level actions $L^\star$ in Table 2, defining Short, Medium, and Long tasks as $L^\star \leq 25$, $26 \leq L^\star \leq 40$, and $L^\star \geq 41$, respectively. For ReTTT-QA, the gain over aTTT increases from 5.6 points on Short tasks to 10.6 on Medium and 14.7 on Long tasks. ReTTT is particularly effective on more demanding multi-stage tasks.

The gains are accompanied by shorter and more stable execution. On Long tasks, ReTTT-QA and ReTTT-Sum use 33.92 and 36.69 steps on average, respectively, compared with 40.20 for aTTT, while both reach higher success versus aTTT. Table 3 further shows that ReTTT-QA reduces total interaction steps from 4665 to 4028 and lowers the unparseable- and inadmissible-action rates from 2.29% and 19.57% to 0.99% and 15.32%. ReTTT-Sum further lowers the inadmissible-action rate to 13.12%.

Table 3: ALFWorld execution statistics with Qwen3.5-9B. Rates are normalized by total average executed steps over three seeds.

Method	Total Steps	Unparseable	Inadmissible
Base	4784	65 (1.36%)	860 (17.98%)
qTTT	4661	91 (1.95%)	686 (14.72%)
aTTT	4665	107 (2.29%)	913 (19.57%)
ReTTT-QA	**4028**	**40 (0.99%)**	617 (15.32%)
ReTTT-Sum	4206	50 (1.19%)	**552 (13.12%)**

ScienceWorld and SWE-bench Lite. The gains extend to substantially different interaction domains. On ScienceWorld, the best ReTTT results reach 26.67% with Qwen3.5-9B and 16.67% with Gemma4-12B, compared with 15.00% and 10.00% for aTTT. On SWE-bench Lite, ReTTT reaches 43.33% with Qwen3.5-9B and 20.00% with Gemma4-12B, compared with 40.00% and 11.67% for aTTT. These results indicate that the benefit of procedural adaptation extends beyond embodied state tracking to scientific interaction and software engineering.

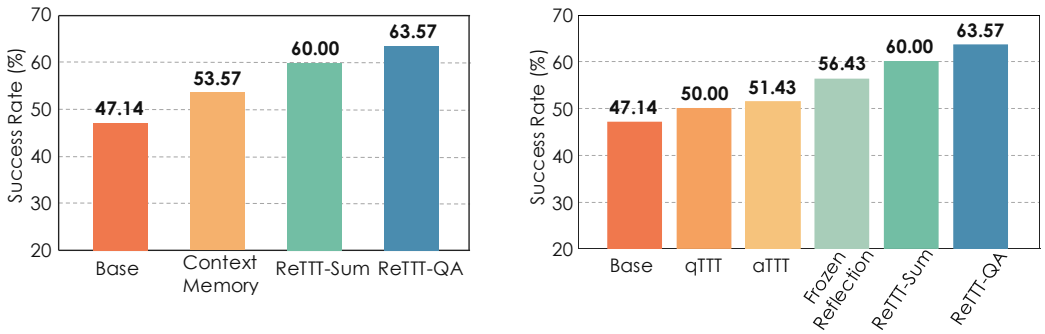
3.3 Is Parametric Procedural Memory More Effective than Explicit Context Memory?

We first ask whether extracted procedural experience is better retained explicitly in context than internalized into model parameters. To isolate this factor, we construct Context Memory, which uses the same procedural reflection and memory pipeline as ReTTT but keeps the model frozen and supplies the maintained experience only as textual context.

As shown in Figure 3a, Context Memory improves success from 47.14% to 53.57%, showing that explicit procedural memory is already useful. Parametric internalization yields substantially larger gains: ReTTT-Sum reaches 60.00% and ReTTT-QA 63.57%, with ReTTT-QA outperforming Context Memory by 10.00 points. Thus, procedural experience is more effective when internalized into fast weights than when reused only as explicit context.

3.4 Is Procedural TTT Effective without Reflection-Policy Adaptation?

We next isolate the role of adapting the reflection policy. Frozen Reflection retains ReTTT's procedural reflection, memory maintenance, and fast-weight updates for action generation, but generates every reflection with the initial policy ϕ_0 rather than the evolving ϕ_k.



(a) Explicit context memory vs. parametric internalization.

(b) Procedural TTT with frozen vs. adapted reflection.

Figure 3: **Two controlled comparisons on ALFWorld with Qwen3.5-9B.**

As shown in Figure 3b, Frozen Reflection reaches 56.43%, outperforming qTTT 50.00% and aTTT 51.43%. Thus, maintained procedural supervision remains effective even with a fixed reflector. Allowing reflection to share the evolving fast weights further raises ReTTT-QA to 63.57%, a 7.14-point gain, showing an additional benefit from adapting the policy that constructs future procedural experience.

Trajectory-level behavior. Figure 4 provides a controlled example: ReTTT and Frozen Reflection share the same trajectory through step 30, but the adapted reflector turns failed searches into task-specific exploration guidance, whereas the frozen reflector repeats generic advice. ReTTT subsequently completes both deliveries by step 43, while Frozen Reflection times out at step 50.

3.5 Ablation Study

We ablate three components of ReTTT-QA, as shown in Table 4: candidate screening, dynamic memory maintenance, and replay.

Candidate screening. Removing screening causes the drop from 63.57% to 59.29%. Without evidence- and trajectory-based validation, unsupported or inconsistent procedures can enter the training set and degrade the adaptation signal.

Dynamic memory maintenance. Disabling dynamic maintenance reduces success to 57.86%. Without similarity-based replacement and revision, redundant or outdated procedures can persist as new evidence arrives. It has a significant impact on performance.

Table 4: Ablation study on ALFWorld with Qwen3.5-9B.

Variant	Success (%)
ReTTT-QA	**63.57**
w/o Replay	62.14
w/o Candidate Screening	59.29
w/o Dynamic Maintenance	57.86

Replay. Removing replay yields a smaller decrease to 62.14%. Training only on newly inserted or revised procedures suggests that replay mainly stabilizes continual adaptation by revisiting previously retained experience.

Overall, these results indicate that the quality and consistency of procedural supervision are more important than simply increasing repeated training. Additional efficiency analysis is provided in Appendix A.1.

4 Related Work

4.1 Memory for Language Agents

Language-agent memory has evolved from explicit storage toward increasingly structured forms of experience reuse. MemGPT (Packer et al., 2023) manages external memory beyond the immediate

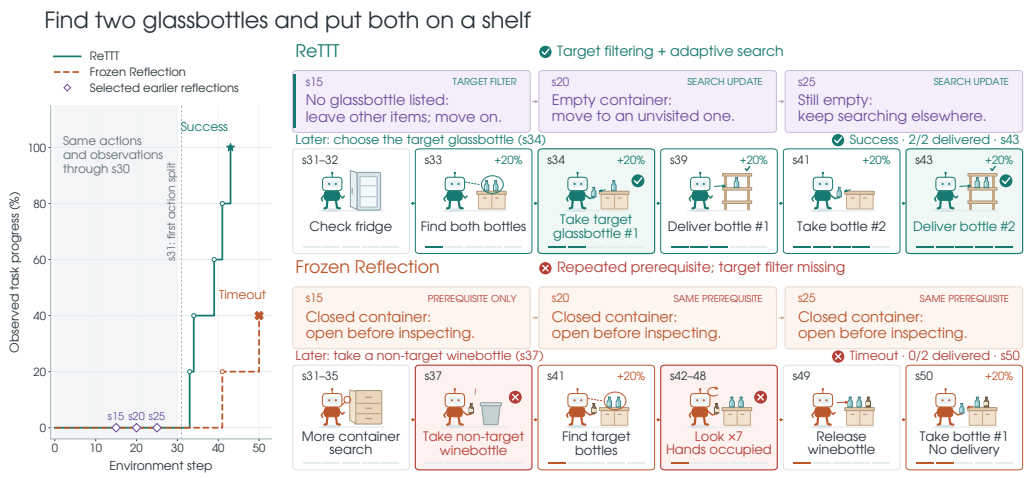


Figure 4: **Trajectory-level comparison of ReTTT and Frozen Reflection.**

context, while Generative Agents (Park et al., 2023) and Reflexion (Shinn et al., 2023) retain episodic experience and reflection in textual memory. Voyager (Wang et al., 2023) further stores reusable executable skills, shifting memory from recalling past observations toward preserving procedural knowledge.

Recent work makes such experience more adaptive. AdaMEM (Zhang et al., 2026) dynamically constructs test-time strategy memory, and ProcMEM (Mi et al., 2026) learns and maintains reusable procedural skills from interaction, both without online parameter updates. Complementary approaches begin to parameterize experience: ParamMem (Yao et al., 2026) encodes cross-sample reflection patterns into parametric memory, while Procedural Memory Distillation (Liu et al., 2026) distills accumulated procedural memory into model weights during training. ReTTT instead targets in-episode procedural internalization: interaction experience is continually reflected, maintained as explicit procedural memory, and written into episode-local fast weights while the same task is still unfolding.

4.2 Test-Time Adaptation for Language Agents

Test-time training adapts model parameters using information available only at inference time. Early TTT methods (Sun et al., 2020) establish this paradigm for distribution adaptation, while qTTT (Bansal et al., 2026) shows that context-specific gradient updates can improve long-context utilization in LLMs. These settings primarily adapt to a fixed test input or context. More recent work extends parameter adaptation to interactive agents. Agentic TTT (aTTT) (Wang et al., 2026) continually trains on the evolving agent trajectory and reweights repeated tokens to mitigate self-training drift.

ReTTT differs in how reflection, memory, and adaptation are coupled and the currently adapted policy first converts interaction history into screened and dynamically maintained procedural experience. This experience is then internalized into fast weights shared by both action and reflection generation, so earlier updates influence not only subsequent behavior but also how later experience is interpreted and transformed into future supervision.

5 Conclusion

We introduced ReTTT, which constructs and internalizes maintained procedural experience for in-episode test-time adaptation, with shared fast weights coupling action and reflection in a closed loop. Across three benchmarks and two model families, ReTTT consistently outperforms trajectory-based agentic TTT, with larger gains on longer-horizon tasks, highlighting the value of reliable procedural supervision and adaptive reflection. Future work can extend ReTTT across episodes by selectively consolidating and transferring reusable procedural experience.

AI USE STATEMENT

In this work, we used generative AI tools solely to aid and polish the writing, including improving grammar, clarity, concision, and presentation of author-written text. We did not use generative AI for literature retrieval or discovery, research ideation or execution, generating synthetic data, or proving mathematical claims. All AI-assisted edits were reviewed and revised by the authors, who verified the technical accuracy and consistency of the final manuscript. We take full responsibility for the final content of this work.

ETHICS STATEMENT

This work does not involve human subjects, private or personally identifiable data, or the collection or release of new datasets. All experiments are conducted on existing research benchmarks in controlled evaluation settings. We are not aware of additional ethical concerns related to privacy, fairness, discrimination, legal compliance, or conflicts of interest arising from this study.

REPRODUCIBILITY STATEMENT

We provide the complete ReTTT procedure in Algorithm 1, including in-episode reflection, procedural-memory maintenance, replay, and fast-weight updates. Section 3 describes the evaluated benchmarks, models, baselines, and evaluation protocol, while the main experimental tables report the corresponding task-level results. Appendix A.2 provides the full test-time adaptation configuration and hyperparameters, including the LoRA setup, optimization settings, procedural-memory budgets, and variant-specific choices for ReTTT-QA and ReTTT-Sum. Appendix A.3 further specifies the procedural-memory implementation, including deterministic candidate screening, same-type Jaccard-based reconciliation and its thresholds, template-based revision, and the rule-based eviction policy. Additional test-time computation and efficiency analyses are reported in Appendix A.1.

REFERENCES

Rachit Bansal, Aston Zhang, Rishabh Tiwari, Lovish Madaan, Venkata Sai Surya Subramanyam Duvvuri, Devvrit Khatri, David Brandfonbrener, David Alvarez-Melis, Prajjwal Bhargava, Mihir Kale, et al. Let's (not) just put things in context: Test-time training for long-context llms. In *International Conference on Learning Representations*, volume 2026, pp. 112130–112153, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Ye Liu, Srijan Bansal, Bo Pang, Yang Li, Zeyu Leo Liu, Yifei Ming, Zixuan Ke, Shafiq Joty, and Semih Yavuz. Procedural memory distillation: Online reflection for self-improving language models. *arXiv preprint arXiv:2607.01480*, 2026.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Procmem: Learning reusable procedural memory from experience via non-parametric ppo for llm agents. *arXiv e-prints*, pp. arXiv–2602, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Tao Ren, Weiyao Luo, Hui Yang, Rongzhi Zhu, Xiang Huang, Yuchuan Wu, Bingxue Chou, Jieping Ye, Jiafeng Liang, Yongbin Li, et al. Scaling self-evolving agents via parametric memory. *arXiv preprint arXiv:2606.04536*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International conference on machine learning*, pp. 9229–9248. PMLR, 2020.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

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

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Yanbo Wang, Jinhua Hao, Yuze Shi, Kun Yuan, and Ming Sun. No time like the present: Agentic test-time training for llm agents. *arXiv preprint arXiv:2607.03441*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

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

Tianjun Yao, Yongqiang Chen, Yujia Zheng, Pan Li, Zhiqiang Shen, and Kun Zhang. Parammem: Augmenting language agents with parametric reflective memory. *arXiv preprint arXiv:2602.23320*, 2026.

Yunxiang Zhang, Yiheng Li, Ali Payani, and Lu Wang. Adamem: Test-time adaptive memory for language agents. *arXiv preprint arXiv:2606.05684*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A Appendix

A.1 Test-Time Compute and Efficiency

ReTTT introduces additional generation cost through reflection while using fewer optimization steps than existing test-time training baselines. We characterize this trade-off using reflection tokens, loss-bearing target-token exposures, adapter gradient steps, environment interaction, and aggregate worker time. Table 5 reports per-episode statistics for the Qwen3.5-9B ALFWorld runs.

ReTTT changes the allocation of test-time computation. ReTTT-QA uses 524.78 loss-bearing target exposures and 7.14 gradient steps per episode, compared with 573.69 exposures and 12.93 gradient steps for aTTT, while improving success from 51.43% to 63.57%. ReTTT-Sum further reduces the optimization load to 257.99 target exposures and 2.51 gradient steps. Both ReTTT variants also require substantially fewer gradient steps than qTTT.

The reduced optimization load is accompanied by shorter environment interaction. As shown in Table 2, ReTTT-QA reduces the average rollout length from 33.32 steps with aTTT to 28.77 steps overall, and from 40.20 to 33.92 steps on long-horizon tasks. Table 3 further shows that ReTTT-QA reduces total executed steps from 4665 to 4028 while producing fewer malformed and inadmissible actions. Thus, ReTTT achieves higher task success with fewer gradient updates and shorter interaction trajectories.

This reduction in optimization does not make ReTTT cheaper overall, because reflection introduces additional generation cost. The aggregate worker time of ReTTT-QA and ReTTT-Sum is approximately 1.085× and 1.124× that of aTTT, respectively. ReTTT therefore shifts test-time computation away from repeated optimization and toward the construction of procedural supervision, achieving stronger performance with a modest increase in aggregate runtime.

Table 5: Test-time computation on ALFWorld with Qwen3.5-9B. *Refl. In / Out* denotes average reflection input/output tokens per episode. *Target Exp.* counts loss-bearing target-token exposures across optimization steps. *Worker Time* is summed worker runtime normalized by Base.

Method	SR	Refl. In / Out	Target Exp.	Grad. Steps	Worker Time(×)
Baselines					
Base	47.14	0 / 0	0	0	1.000
qTTT	50.00	0 / 0	4064.00	32.00	2.001
aTTT	51.43	0 / 0	573.69	12.93	2.724
ReTTT					
ReTTT-QA	**63.57**	27875.66 / 1175.30	524.78	7.14	2.955
ReTTT-Sum	60.00	29442.70 / 1001.23	**257.99**	**2.51**	3.061

A.2 Implementation Details and Hyperparameters

We use the same test-time adaptation configuration across benchmarks and model backbones. The backbone parameters remain frozen throughout each episode, and only episode-local LoRA fast weights are optimized. The adapter, optimizer state, and procedural memory are reset after each episode. Table 6 summarizes the common adaptation hyperparameters.

Reflection is performed periodically during interaction using the same episode-local adapted model as action generation. Generated procedural experience is screened and maintained before being used for parameter updates. ReTTT-QA and ReTTT-Sum share the same adaptation mechanism but differ in the serialization of procedural experience, as summarized in Table 7.

A.3 Additional Details of Procedural Memory Maintenance

We provide additional implementation details of the candidate screening, memory reconciliation, revision, and eviction operations used in ReTTT. Each procedural candidate is represented as $z = (\kappa, u, E)$, where κ denotes its procedural type, u its decision-relevant content, and E its supporting

Table 6: Shared test-time adaptation hyperparameters.

Setting	Value
Backbone precision	bfloat16
Adapter precision	float32
Adapted module	Final attention `q_proj`
Parallel LoRA components	2
LoRA rank	4
LoRA α	8
Optimizer	AdamW
Learning rate	5×10^{-4}
AdamW (β_1, β_2)	$(0.9, 0.999)$
AdamW ϵ	10^{-8}
Weight decay	0
Gradient clipping	1.0
Maximum training length	1,024 tokens
Supervised tokens	Target
Memory capacity	12 entries
Memory target-token budget	768 tokens
Cross-episode persistence	None

Table 7: Variant-specific settings of ReTTT-QA and ReTTT-Sum.

Setting	ReTTT-QA	ReTTT-Sum
Representation	Procedural QA	Progress summary
Generated items / reflection	≤ 3	≤ 1
Maximum example updates	5	3
Memory update	Item-level	Latest checkpoint

trajectory evidence. All operations described below are deterministic and introduce no additional model calls beyond the reflection step described in the main method.

Candidate screening. Before memory reconciliation, each reflected candidate is passed through a deterministic screening stage. We first verify that the candidate follows the expected procedural schema, that the cited evidence refers to valid steps in the current interaction trajectory, and that the serialized training target satisfies the prescribed token budget.

Evidence consistency is checked using fixed rules and string-matching predicates over the cited trajectory observations, environment feedback, and task outcome. In particular, a candidate is not accepted merely because its evidence identifiers are valid: procedural claims that contradict the corresponding trajectory evidence are rejected. For example, claims about successful progress must be supported by the associated environment feedback rather than only by an attempted action. Screening therefore performs rule-based evidence grounding and trajectory-consistency checks, without an additional LLM judge, learned verifier, or entailment model. Candidates that fail any screening condition are discarded before entering memory or contributing to parameter updates.

Similarity-based reconciliation. Each surviving candidate is compared only with memory entries of the same procedural type. Entries with a different procedural type are not considered as reconciliation candidates. For two same-type entries z and m, we compute Jaccard similarity over their normalized decision-relevant content:

$$s(z, m) = \frac{|C(z) \cap C(m)|}{|C(z) \cup C(m)|}, \tag{15}$$

where $C(\cdot)$ denotes the normalized content representation used for memory matching. We then select

$$m^* = \underset{m \in \mathcal{M}_k : \kappa_m = \kappa_z}{\arg\max} \; s(z, m), \qquad s^* = s(z, m^*). \tag{16}$$

Table 8: **Rule-based specificity classes used for memory eviction.** Task-specific entries are retained before generic entries.

Category	Eviction class
Failure recovery	Task-specific
Multi-object	Task-specific
Task transform	Task-specific
Termination state	Task-specific
Generic container	Generic
Generic navigation	Generic
Other	Generic

If no entry with the same procedural type exists, the candidate is directly treated as a new entry.

We use fixed thresholds

$$\tau_{\text{low}} = 0.3, \qquad \tau_{\text{high}} = 0.7, \tag{17}$$

and reconcile the candidate according to

$$\text{Reconcile}(z, m^*) = \begin{cases} \text{Insert}(z), & s^* < 0.3, \\ \text{Revise}(m^*, z), & 0.3 \le s^* < 0.7, \\ \text{Replace}(m^*, z), & s^* \ge 0.7. \end{cases} \tag{18}$$

Thus, low-overlap candidates introduce new procedures, highly overlapping candidates replace an existing entry with the newly observed version, and partially overlapping candidates trigger revision.

Template-based revision. For moderate similarity, ReTTT revises the matched memory entry using a fixed deterministic template rather than an additional language-model call. The template concatenates the existing procedural content with the newly observed information into a single updated entry:

$$m' = \text{TemplateMerge}(m^*, z). \tag{19}$$

This operation is purely rule based and does not invoke the reflection model or any separate verifier. The revised entry is included in $\mathcal{N}_k$ and is therefore eligible for subsequent test-time internalization.

Memory bounds and eviction. After reconciliation, ReTTT enforces both an entry-count budget and a target-token budget:

$$|\mathcal{M}_{k+1}| \le B, \qquad \sum_{m \in \mathcal{M}_{k+1}} \tau(m) \le T, \tag{20}$$

where $\tau(m)$ denotes the number of target tokens associated with entry m. In our experiments, $B = 12$ and $T = 768$.

To determine eviction priority, we use a fixed lexical classifier over the normalized procedural content. Each memory entry is assigned to one of seven categories shown in Table 8. The first four categories capture task-specific progress, requirements, or recovery information and are therefore retained with higher priority. The remaining three categories represent more generic procedural guidance and are evicted first when the memory budget is exceeded. The classifier is rule based and does not invoke an LLM.

When either memory constraint is violated, ReTTT first considers entries in the generic classes for eviction. If multiple generic entries are available, the oldest one is removed. If no generic entry remains, the oldest task-specific entry is removed instead. Eviction is repeated until both memory constraints are satisfied.

FIFO age is determined by the original insertion time of an entry. Replacing or revising an existing entry does not reset its age for eviction purposes. Thus, reconciliation can update the content of a stored procedure without moving it to the end of the FIFO queue.

A.4 More Case Studies

We provide additional trajectory-level examples to complement the controlled case study in the main text. Across ALFWorld, ScienceWorld, and SWE-bench Lite, these examples illustrate how ReTTT converts interaction feedback into task-specific procedural guidance and how the resulting adaptation shapes subsequent execution.

Additional ALFWorld cases. Figures 5 and 6 show the same qualitative pattern as the example in the main text. In the cooling task, ReTTT identifies the missing operation—placing the object in the refrigerator does not itself cool it—and converts this feedback into the explicit action needed to complete the task. In the second case, once the target tomato becomes visible, the adapted reflector updates its guidance from continued exploration to immediate target acquisition. By contrast, Frozen Reflection continues to produce generic prerequisite-oriented guidance or unsupported rules, leading to unnecessary search and eventual timeout. Together, these cases show that adapted reflection can revise procedural guidance as new task-specific evidence becomes available.

ScienceWorld cases. Figures 7–9 illustrate how procedural adaptation supports multi-step scientific interaction. In the animal-delivery task, ReTTT identifies the required focus operation and immediately follows it with pickup and delivery. In the two comparison tasks, ReTTT gathers evidence from both candidates before committing to a final choice. In contrast, aTTT either omits a required prerequisite or selects a candidate before performing the necessary comparison. These cases highlight the importance of converting intermediate environment feedback into guidance about what evidence or operation is still required before completing the task.

SWE-bench Lite cases. Figures 10–12 show the same effect in repository-level software engineering. Across all three examples, ReTTT reaches a fully correct patch with fewer agent steps than aTTT. Its trajectories make more localized diagnoses and targeted edits: preserving the correct relationship type, restricting hashability normalization to the component that requires it, and retaining conflict arguments when adding the missing bulk-upsert branch. This reduces unnecessary edits and validation cycles while preserving task constraints. In contrast, aTTT either requires additional correction steps or, in the bulk-upsert case, converges to an incomplete fix that introduces regressions.

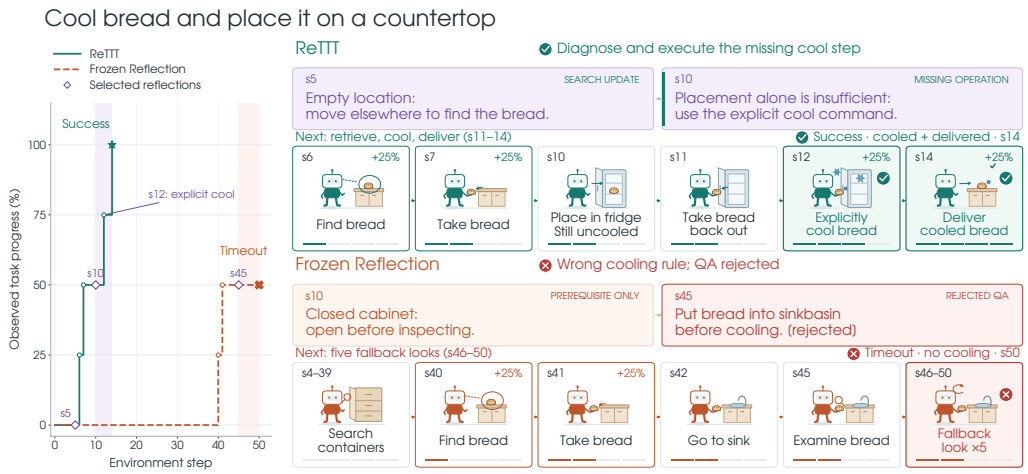


Figure 5: **ALFWorld case: identifying a missing operation.** ReTTT recognizes that placement alone does not cool the bread and executes the required cooling operation, while Frozen Reflection times out.

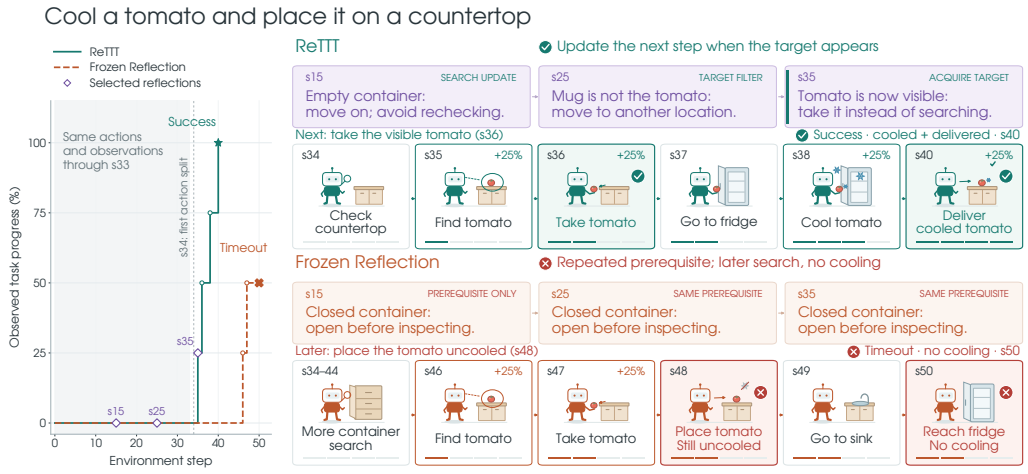


Figure 6: **ALFWorld case: updating the next action from new evidence.** ReTTT switches from exploration to target acquisition once the tomato is observed, whereas Frozen Reflection continues repetitive guidance and times out.

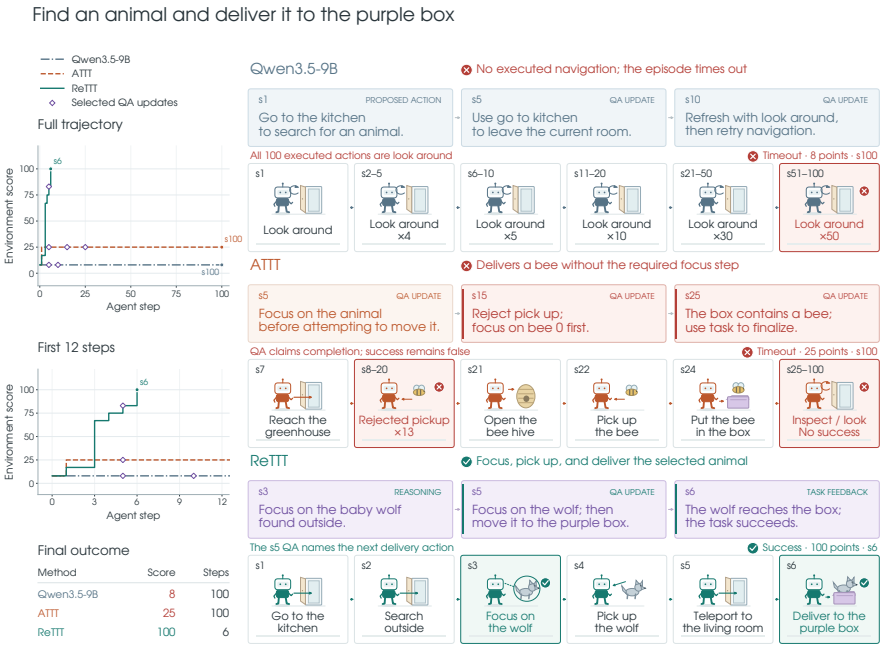


Figure 7: **ScienceWorld case: satisfying a required interaction prerequisite.** ReTTT focuses on the selected animal before pickup and delivery, completing the task in six steps, while the baselines fail to execute the required sequence.

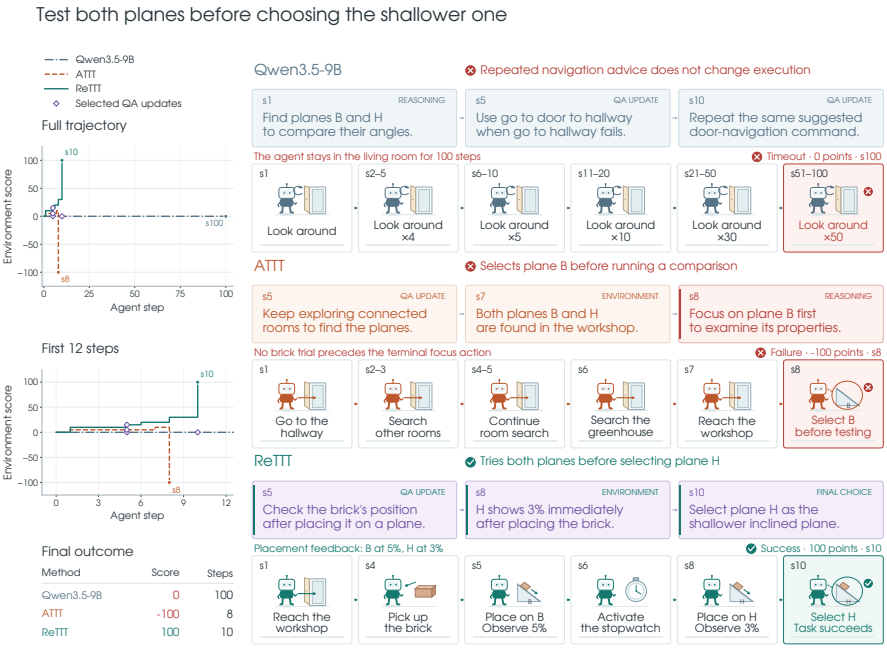


Figure 8: **ScienceWorld case: gathering evidence before selection.** ReTTT tests both planes and uses the observed responses to select the shallower one, whereas aTTT commits before performing the required comparison.

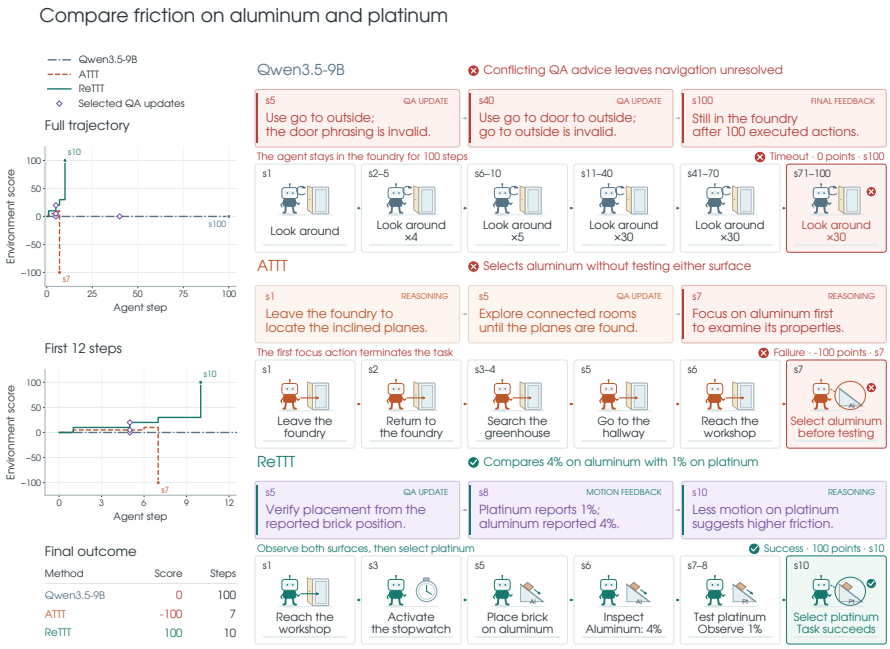


Figure 9: **ScienceWorld case: comparing candidates before committing.** ReTTT tests both surfaces and selects platinum from the observed motion feedback, while aTTT makes a premature selection without performing the comparison.

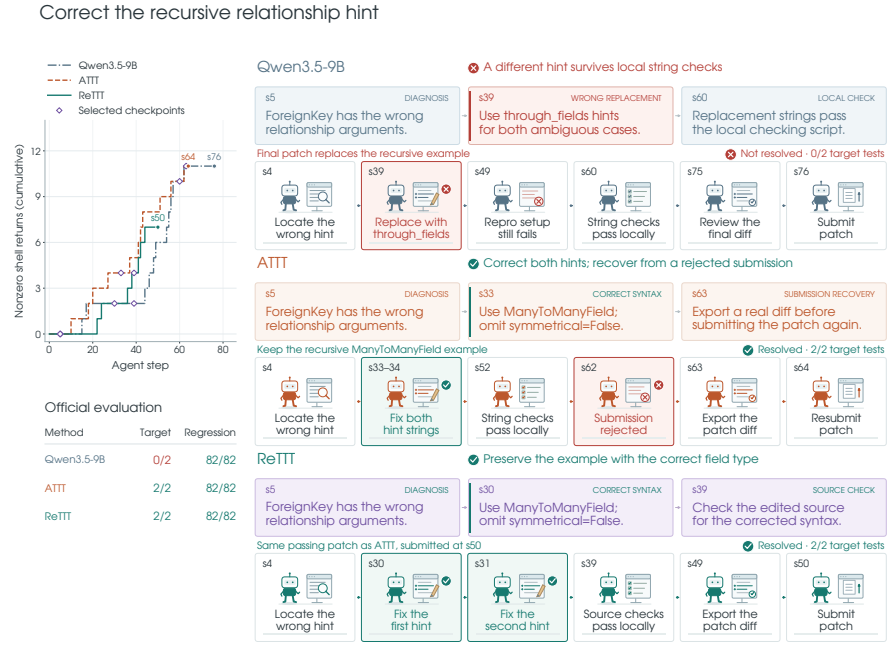


Figure 10: **SWE-bench Lite case: correcting a recursive relationship hint.** ReTTT preserves the correct field type and reaches the fully passing patch in fewer agent steps than aTTT.

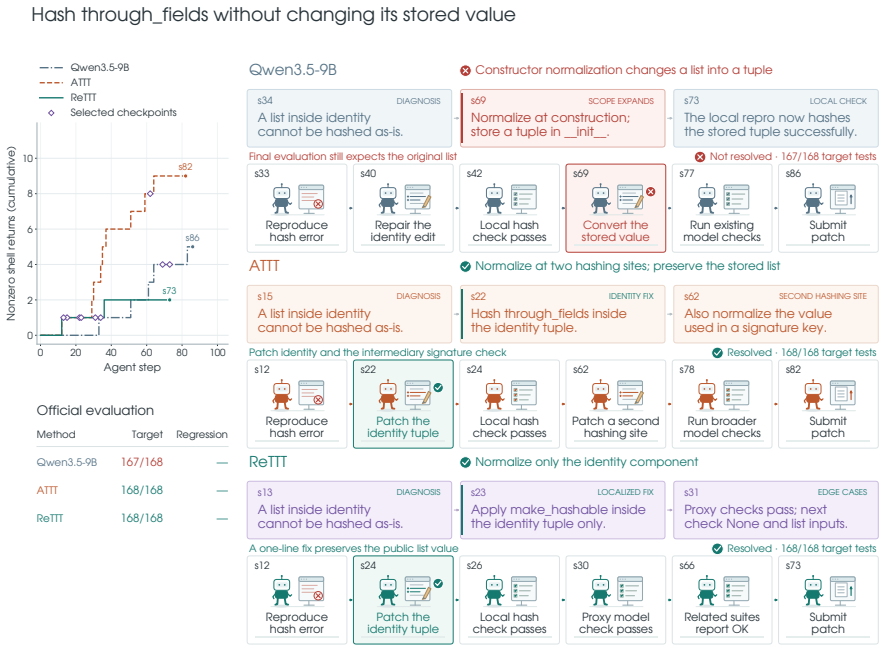


Figure 11: **SWE-bench Lite case: localizing a hashability fix.** ReTTT applies the fix only where hashability is required, preserving the stored public value and reaching a fully passing solution.

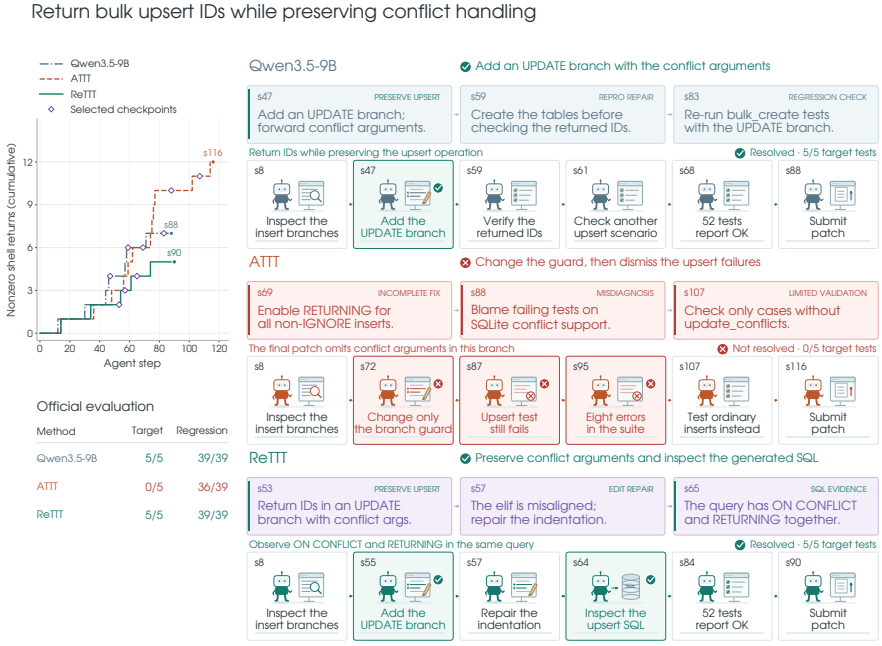


Figure 12: **SWE-bench Lite case: preserving bulk-upsert semantics.** ReTTT preserves the conflict arguments and passes all target and regression tests, while aTTT converges to an incomplete fix.