SALT: Step-level Advantage Assignment for Long-horizon Agents via Trajectory Graph

Jiazheng Li[1]*, **Yawei Wang**[2], **David Yan**[2], **Yijun Tian**[2]
Zhichao Xu[2], **Huan Song**[2], **Panpan Xu**[2]†, **Lin Lee Cheong**[2]
[1]University of Connecticut [2]Amazon
jiazheng.li@uconn.edu {yawenwan, qiaojiny, yijunt
xzhichao, huanso, xupanpan, lcheong}@amazon.com

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities, enabling language agents to excel at single-turn tasks. However, their application to complex, multi-step, and long-horizon tasks remains challenging. While reinforcement learning (RL) offers a promising avenue for addressing these challenges, mainstream approaches typically rely solely on sparse, outcome-based rewards, a limitation that becomes especially problematic for group-based RL algorithms lacking critic models, such as Group Relative Policy Optimization (GRPO). In such methods, uniformly rewarding or penalizing all actions within a trajectory can lead to training instability and suboptimal policies, because beneficial and detrimental actions are often entangled across multi-step interactions. To address this challenge, we propose **SALT**, a novel and lightweight framework that provides a finer-grained advantage assignment, derived solely from outcome rewards. We achieve this by constructing a graph from trajectories of the same prompt, which allows us to quantify the quality of each step and assign advantages accordingly. Crucially, SALT is designed as a plug-and-play module that seamlessly integrates with existing group-based RL algorithms, requiring no modifications to the rollout procedure and introducing negligible computational overhead. Extensive experiments on the WebShop, ALFWorld, and AppWorld benchmarks with various model sizes demonstrate that SALT consistently improves performance. We also conduct a thorough analysis to validate the design choices behind SALT and offer actionable insights.

1 Introduction

Recent advances in large language models (LLMs) have demonstrated their remarkable potential to function as intelligent agents, enabling a diverse array of applications, including web agents (Wu et al., 2025; Li et al., 2025b), search agents (Jin et al., 2025; Sun et al., 2025), coding agents (Luo et al., 2025; Yang et al., 2024), and embodied agents (Xi et al., 2024; Intelligence et al., 2025). While LLM-based agents excel at simple, single-step tasks, like weather forecast inquiry with API calling, they often struggle in complex, long-horizon scenarios that require sustained, multi-step interaction with external environments (Zhou et al., 2025; Chen et al., 2025a). For example, in the AppWorld benchmark (Trivedi et al., 2024), completing the task *Like all the Venmo transactions from today involving any of my roommates on my Venmo social feed* requires GPT-4o (Hurst et al., 2024) using the ReAct framework (Yao et al., 2023) to execute a sequence of 18 precisely coordinated steps, from *logging in* to *identifying transactions* and finally *liking them*, which makes them particularly challenging for LLM-based agents.

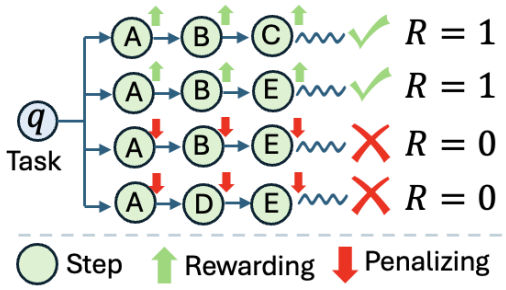

Figure 1: In group-based agentic RL, steps like A, B, or E that appear across multiple trajectories sometimes receive inconsistent advantages, being rewarded in some and penalized in others. This inconsistency stems from trajectory-level reward assignment and can lead to gradient conflicts during the policy update process.

To tackle these challenges, recent work has increasingly turned to reinforcement learning (RL, Wang et al., 2025b; Xi et al., 2025), particularly in goal-oriented settings where collecting high-

*Work done during internship at Amazon.
†Corresponding author: xupanpan@amazon.com

Findings of the Association for Computational Linguistics: EACL 2026, pages 4709–4725
March 24-29, 2026

quality expert demonstrations for supervised fine-tuning (SFT) is not only labor-intensive but also limits generalization. In contrast, RL directly optimizes for verifiable objectives, such as task success rate, and naturally fosters the "explore-and-exploit" behavior essential for navigating dynamic, interactive environments (Chen et al., 2025b; Singh et al., 2025). Among RL approaches, group-based algorithms, notably Group Relative Policy Optimization (GRPO, Shao et al., 2024), have gained widespread adoption for training LLM agents across diverse domains, including search (Jin et al., 2025), tool use (Singh et al., 2025), and more. Their appeal lies in their simplicity and scalability compared to non-group-based methods like PPO (Schulman et al., 2017), which requires maintaining a separate critic model to estimate value functions.

Group-based RL algorithms typically compute advantages by comparing the relative final rewards of multiple trajectories that attempt the same task, then uniformly broadcasting this scalar advantage to every step within each trajectory. Existing works (Feng et al., 2025b; Zhang et al., 2025; Zhou et al., 2025) have shown that this coarse-grained credit assignment is fundamentally suboptimal, especially in long-horizon settings where beneficial and detrimental actions are often entangled. Rewarding or penalizing all actions based solely on final outcomes can lead to unstable policy updates and degraded performance.

To mitigate the limitation of coarse-grained advantage assignment, we introduce SALT, a lightweight yet highly effective mechanism that delivers fine-grained, step-level advantages. SALT is motivated by a simple observation: *trajectories that solve the same task often share some steps yet diverge at others.* We believe that steps shared across all trajectories are typically neutral or non-differentiating (e.g., Step A in Figure 1), while those unique to high-reward trajectories are likely beneficial (e.g., Step C), and those exclusive to failures are likely detrimental (e.g., Step D).

Unlike prior methods, SALT explicitly identifies shared versus distinct steps through trajectory graph construction, requiring no additional supervision or reward models. It operates solely on sparse, episodic returns. Concretely, given a task, SALT first generates a batch of parallel rollouts, as in standard group-based RL. It then unifies these trajectories into a single trajectory graph using two elementary operations: *merge* and *diverge*. This structure enables quantitative step-level advantage refinement, which directly guides policy updates. Designed as a plug-and-play module, SALT integrates seamlessly into existing pipelines, inheriting their algorithmic strengths while adding negligible computational overhead.

We evaluate SALT on three challenging long-horizon benchmarks: ALFWorld (embodied reasoning), WebShop (interactive e-commerce), and AppWorld (digital personal assistance). By plugging SALT into GRPO and RLOO, we obtain consistent performance gains across model scales (1.5B, 7B and 32B parameters).

In summary, our contributions are as follows:

1. We propose SALT, a novel framework for step-level advantage assignment in long-horizon agentic RL. By constructing a trajectory graph to distinguish shared and distinct steps, SALT produces fine-grained advantages without any additional supervision or reward models.

2. SALT is a lightweight, plug-and-play module that integrates effortlessly into existing group-based RL pipelines. It consistently enhances performance while incurring negligible computational cost.

3. Through extensive experiments on ALFWorld, WebShop, and AppWorld, we demonstrate SALT's consistent superiority across various tasks and model sizes. Detailed analysis and case studies further validate the effectiveness and interpretability of our approach.

2 Preliminaries

In this section, we formalize the problem setting and review the fundamental reinforcement learning algorithms relevant to our approach.

2.1 LLM Agents

We formalize the interaction between an agent and its environment in long-horizon tasks as a Markov Decision Process (MDP). While the classical MDP assumes full observability of the environment state, in the context of LLM agents, the agent typically receives information through natural language observations rather than direct access to the underlying state. To accommodate this, we adopt a formulation that explicitly distinguishes between the environment's true state and the agent's observation. Formally, we define the process as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{F}, \mathcal{R})$, where $\mathcal{S}$ is the set of environment

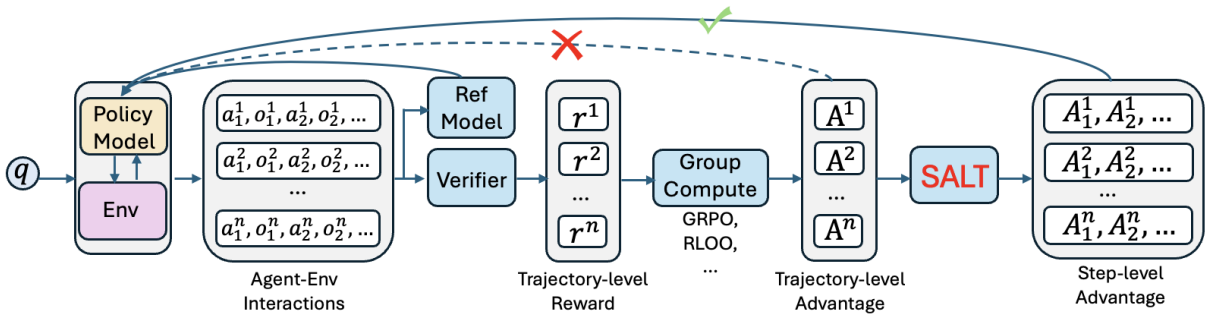

Figure 2: **Training pipeline with SALT**. After parallel rollouts and reward assignment, group-based RL computes trajectory-level advantages. SALT is then inserted and refines these into step-level advantages by leveraging cross-trajectory structure, enabling fine-grained policy updates, without altering the rollout or reward pipeline.

states, $\mathcal{A}$ is the action space, $\mathcal{O}$ is the observation space, $\mathcal{F} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition function, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. In our setting, which is tailored for LLM agents, the state, action, and observation spaces $(\mathcal{S}, \mathcal{A}, \mathcal{O})$ are all represented as natural language sequences over a finite token vocabulary.

At each timestep t, the LLM agent π_θ generates an action a_t based on the current state s_{t-1}: $a_t \sim \pi_\theta(\cdot|s_{t-1})$. After executing the action, the agent receives the environmental feedback as the observation o_t. The interaction loop terminates when either the agent completes the task or the maximum step is reached. The final trajectory is $\boldsymbol{\tau} = (\boldsymbol{q}, a_1, o_1 ..., a_n, o_n)$, where n denotes the length of the trajectory and $\boldsymbol{q}$ denotes the task. At termination, a scalar reward $\mathcal{R}(\boldsymbol{\tau})$ is provided. The agent's objective is to learn an optimal policy π_θ that maximizes the expected cumulative reward:

$$\max_\theta \mathbb{E}_{\tau \sim \pi_\theta}[\mathcal{R}(\boldsymbol{\tau})]. \tag{1}$$

2.2 Reinforcement Learning for Agents

Proximal Policy Optimization (PPO) PPO (Schulman et al., 2017) is a widely adopted policy gradient algorithm in LLM agent training. To stabilize training, PPO restricts policy updates to remain within a proximal region of the old policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$ using the following clipped surrogate to maximize the objective:

$$\max_\theta \mathbb{E}_{\boldsymbol{q} \sim \mathcal{D}, \boldsymbol{\tau} \sim \pi_{\theta_{\text{old}}}(\cdot|\boldsymbol{q})} \Big[\min \big(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \big) \Big], \tag{2}$$

$$\text{with} \quad r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(\tau_t \mid \boldsymbol{q}, \boldsymbol{\tau}_{<t})}{\pi_{\boldsymbol{\theta}_{\text{old}}}(\tau_t \mid \boldsymbol{q}, \boldsymbol{\tau}_{<t})}.$$

Here, $\mathcal{D}$ is a dataset of tasks $\boldsymbol{q}$, $\epsilon \in \mathbb{R}$ is a clip hyperparameter usually set to 0.2, and $\hat{A}_t$ is the estimated advantage, typically computed via Generalized Advantage Estimation (GAE) (Schulman et al., 2015) using a critic network.

Note that while PPO estimates token-level advantages, it falls short compared to our method in several aspects: (1) It relies on a separate critic network, which limits scalability and efficiency. In contrast, our approach introduces only negligible computational overhead and entirely avoids the need for a critic. (2) PPO does not leverage group rollouts or collective reward computation, both of which are integral to our framework and lead to more reliable credit assignment.

Group Relative Policy Optimization (GRPO) Building on the clipped objective in eq. (2), GRPO discards the critic network by estimating advantages using the average reward within a group of sampled responses. Specifically, for each task $\boldsymbol{q}$, the LLM agent $\pi_{\boldsymbol{\theta}_{\text{old}}}$ generates a group of trajectories $\{\boldsymbol{\tau}_i\}_{i=1}^G$ with corresponding outcome rewards $\{R(\boldsymbol{\tau}_i)\}_{i=1}^G$, where $G \in \mathbb{R}$ is the group size. The estimated advantage $\hat{A}_t^i$ is then computed as:

$$\hat{A}_t^i = \frac{R(\boldsymbol{\tau}_i) - \text{mean}(\{R(\boldsymbol{\tau}_j)\}_{j=1}^G)}{\text{std}(\{R(\boldsymbol{\tau}_j)\}_{j=1}^G)} \tag{3}$$

$$\text{where } R(\boldsymbol{\tau}_i) = \begin{cases} 1.0 & \text{if } \texttt{task\_complete}, \\ 0.0 & \text{otherwise.} \end{cases}$$

In addition to this modified advantage estimation, GRPO adds an explicit KL penalty term to the clipped objective in eq. (2).

The group computation used in GRPO and other group-based RL algorithms (Liu et al., 2025; Yu et al., 2025) is highly memory-efficient and can scale effectively to large batch sizes and model sizes typical in modern LLM training, making it a practical and scalable choice.

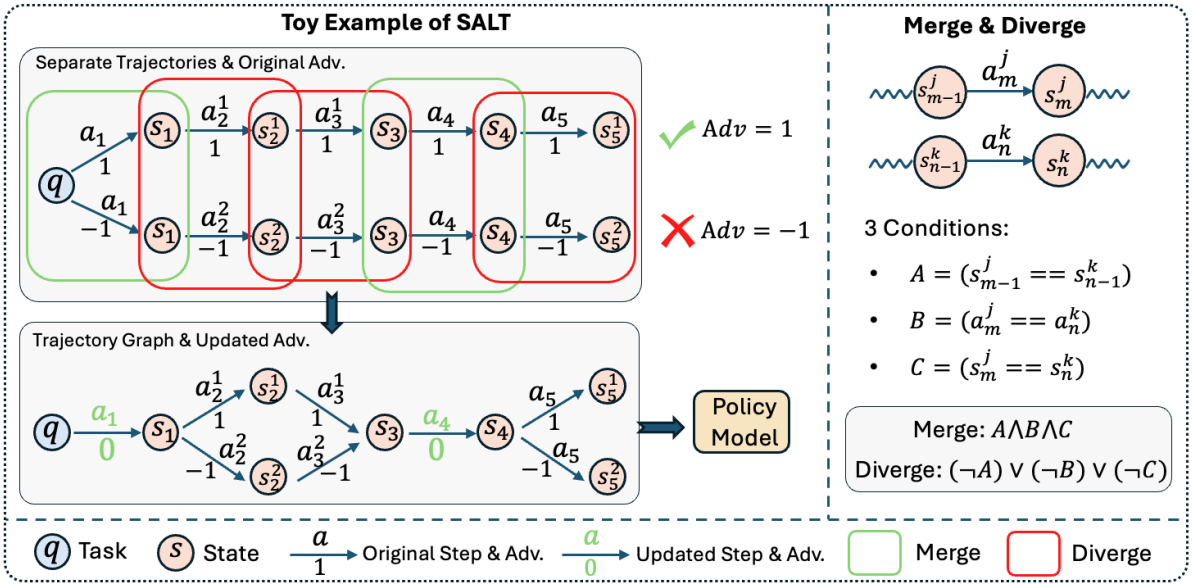

Figure 3: **Illustration of SALT on a single task with multiple rollouts.** Trajectories from the same prompt are constructed into a trajectory graph using *merge* and *diverge*. Step-level advantages are then refined by averaging advantages over merged edges while preserving original advantages for distinct edges. This yields fine-grained credit assignment using only sparse final rewards.

3 Methods

In this section, we present SALT in detail. Our framework consists of two key components: (1) **Trajectory Graph Construction**, where we build a trajectory graph from multiple rollouts of the same task using *merge* and *diverge* operations; (2) **Step-level Advantage Assignment** where we refine advantages at the step level by leveraging the graph structure and original outcome rewards.

3.1 Trajectory Graph Construction

3.1.1 Graph Initialization

As introduced in Preliminary 2.1, given a task $\boldsymbol{q}$, group-based RL algorithms generate a set of trajectories $\{\boldsymbol{\tau}_i\}_{i=1}^G$, each associated with a scalar outcome reward $R(\boldsymbol{\tau}_i)$. These rewards are used to compute group-normalized advantages $\{\hat{A}^i\}_{i=1}^G$ (Eq. 3). Within each trajectory $\boldsymbol{\tau}_i = (\boldsymbol{q}, a_1^i, o_1^i, \ldots, a_{n_i}^i, o_{n_i}^i)$, all steps $(a_1^i, \ldots, a_{n_i}^i)$ are assigned the same advantage $\hat{A}^i$ — meaning that regardless of their individual quality, all actions are uniformly rewarded or penalized.

However, as we mentioned, we observe that beneficial and detrimental actions are often entangled across multi-step interactions. To address this, SALT replaces trajectory-level advantages with fine-grained, step-level advantages $\{\hat{A}_s^i\}_{i=1}^G$.

To achieve this, we construct a directed acyclic trajectory graph $\mathcal{G} = (V, E, H)$ over the set of trajectories $\{\boldsymbol{\tau}_i\}_{i=1}^G$ for task $\boldsymbol{q}$.

- V (nodes) represents all states, including:
  - The task description $\boldsymbol{q}$ as the root node, and
  - All subsequent states across trajectories:

$$V = \{\boldsymbol{q}, \underbrace{s_1^1, s_2^1, \cdots s_{n_1}^1}_{traj.\ 1}, \cdots, \underbrace{s_1^G, s_2^G, \ldots s_{n_G}^G}_{traj.\ G}\}. \tag{4}$$

- E (edges) represents all actions in the form of tuples:

$$E = \{\underbrace{(\boldsymbol{q}, a_1^1, s_1^1), (s_1^1, a_2^1, s_2^1), \cdots, (s_{n_1-1}^1, a_{n_1}^1, s_{n_1}^1)}_{\text{traj. 1}}, \cdots, \underbrace{(\boldsymbol{q}, a_1^G, s_1^G), (s_1^G, a_2^G, s_2^G), \cdots, (s_{n_G-1}^G, a_{n_G}^G, s_{n_G}^G)}_{\text{traj. G}}\}. \tag{5}$$

- H (advantage values) stores the initial advantage for each action:

$$H = \{\underbrace{\hat{A}_1^1, \hat{A}_2^1, \cdots \hat{A}_{n_1}^1}_{traj.\ 1}, \cdots, \underbrace{\hat{A}_1^G, \hat{A}_2^G, \ldots \hat{A}_{n_G}^G}_{traj.\ G}\}. \tag{6}$$

Initially, the graph shares a common root node $\boldsymbol{q}$, with G branches diverging from it. Within each branch, all edges inherit the same trajectory-level advantage (i.e., $\hat{A}_1^i = \hat{A}_2^i = \cdots = \hat{A}_{n_i}^i$).

3.1.2 Graph Refinement

We then refine the graph using two operations — *merge* and *diverge*:

Merge Operation. Two edges $(s^j_{m-1}, a^j_m, s^j_m)$ and $(s^k_{n-1}, a^k_n, s^k_n)$ are merged if **all** of the following hold:

- $s^j_{m-1} = s^k_{n-1}$: same starting state,
- $a^j_m = a^k_n$: same action taken,
- $s^j_m = s^k_n$: same resulting state.

Diverge Operation. Two edges diverge if **any one** of the following is true:

- $s^j_{m-1} \neq s^k_{n-1}$: different starting states,
- $s^j_{m-1} = s^k_{n-1}$ but $a^j_m \neq a^k_n$: same state, different action,
- $s^j_{m-1} = s^k_{n-1}$, $a^j_m = a^k_n$, but $s^j_m \neq s^k_n$: same state and action, but different resulting state.

Note that we define the state s_t as the sequence of the most recent h observation-action pairs, i.e., $s_t = \{a_{t-h+1}, o_{t-h+1}..., a_t, o_t\}$, rather than relying solely on the immediate observation o_t. This design enables SALT to capture richer contextual dependencies across steps. The history length h serves as a tunable hyperparameter that controls the strictness of merge and diverge operations: larger h encourages stricter state matching, while smaller h allows more flexible matching.

We construct the trajectory graph $\mathcal{G}' = (V', E', H)$ in a greedy, sequential manner: for each trajectory in the group, we process its steps from start to end, incrementally integrating them into the evolving graph via *merge* and *diverge* operations. This online construction yields a refined graph that explicitly encodes which steps are shared across trajectories and which are distinct.

3.2 Step-level Advantage Assignment

Given the refined graph $\mathcal{G}'$, we now reassign advantages to better reflect each step's contribution.

Suppose there are M sets of merged edges: $\{E_1, E_2, \ldots, E_M \mid E_i \subset E\}$, where each $E_i = \{a_1, a_2, \ldots, a_{n_i}\}$ and corresponding advantages $H_i = \{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_{n_i}\}$ (we omit superscripts here for clarity). For each merged set, we update all associated advantages to their group mean:

$$\hat{A}'_1 = \hat{A}'_2 = \cdots = \hat{A}'_{n_i} = \frac{1}{n_i} \sum_{j=1}^{n_i} \hat{A}_j, \quad (7)$$

where $\hat{A}'$ denotes the updated advantages.

The intuition is simple: steps that appear identically across multiple trajectories are likely "neutral" or "required" — they should not be over-penalized or over-rewarded based on any single trajectory's outcome. Averaging their advantages reduces gradient conflict and stabilizes training.

For divergent edges, those unique to specific trajectories, we retain their original advantages. These steps are the true differentiators: they directly influence whether a trajectory succeeds or fails, and thus deserve trajectory-specific credit.

The result is an updated advantage set H', containing fine-grained, step-level signals ready to guide policy updates as in eq. (2)

Importantly, this refinement mechanism is particularly effective in long-horizon settings where trajectories may contain both suboptimal and corrective actions. By aggregating evidence across multiple rollouts, SALT can distinguish steps that are consistently beneficial from those that are incidental or recoverable errors. In the following subsection, we analyze how this property enables robust credit assignment in common "mistake-and-recover" scenarios.

3.3 Credit Assignment under Error Recovery

SALT is particularly effective in *mistake-and-recover* scenarios, where early suboptimal actions are followed by corrective steps leading to success. In a group of size G, it is statistically likely that multiple trajectories encounter the same error state s_{err}. If some trajectories fail while others recover, SALT **merges** these shared error nodes, assigning them a "soft penalty" (the averaged group mean) that is lower than the advantage of consistently optimal steps.

Conversely, the subsequent **corrective actions** remain unique to successful trajectories. As these edges **diverge** in the graph, SALT preserves their high trajectory-level advantages without dilution. This mechanism effectively isolates credit to the specific recovery behaviors while dampening the noise introduced by incidental early mistakes, guiding the policy toward cleaner, more efficient paths.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate our method on three challenging long-horizon agent benchmarks: (1) **WebShop** (Yao et al., 2022): a simulated e-

Type	Method	ALFWorld							WebShop	
		Pick	Look	Clean	Heat	Cool	Pick2	All	Score	Succ.
Base: Closed-Source Models										
Prompting	GPT-4o	75.3	60.8	31.2	56.7	21.6	49.8	48.0	31.8	23.7
	Gemini-2.5-Pro	92.8	63.3	62.1	69.0	26.6	58.7	60.3	42.5	35.9
Base: Qwen2.5-1.5B-Instruct										
Prompting	Qwen2.5	5.9	5.5	3.3	9.7	4.2	0.0	4.1	23.1	5.2
	ReAct	17.4	20.5	15.7	6.2	7.7	2.0	12.8	40.1	11.3
	Reflexion	35.3	22.2	21.7	13.6	19.4	3.7	21.8	55.8	21.9
RL Training	PPO	$93.7_{\pm 1.1}$	$80.0_{\pm 9.4}$	$93.6_{\pm 3.5}$	$83.1_{\pm 6.0}$	$74.6_{\pm 7.7}$	$70.8_{\pm 6.8}$	$83.5_{\pm 1.4}$	$85.8_{\pm 2.1}$	$72.6_{\pm 3.2}$
	RLOO	$91.3_{\pm 2.9}$	$\mathbf{77.0}_{\pm 15.5}$	$79.9_{\pm 7.5}$	$79.2_{\pm 1.7}$	$73.2_{\pm 5.5}$	$\mathbf{68.8}_{\pm 2.5}$	$79.2_{\pm 3.5}$	$84.1_{\pm 1.4}$	$71.6_{\pm 2.4}$
	RLOO+SALT	$\mathbf{94.2}_{\pm 2.1}$	$74.4_{\pm 9.9}$	$\mathbf{95.0}_{\pm 1.4}$	$\mathbf{89.5}_{\pm 7.5}$	$\mathbf{79.5}_{\pm 8.6}$	$64.9_{\pm 4.6}$	$\mathbf{84.0}_{\pm 2.2}$	$\mathbf{87.9}_{\pm 0.8}$	$\mathbf{75.8}_{\pm 2.9}$
	GRPO	$92.1_{\pm 1.6}$	$64.3_{\pm 20.2}$	$89.1_{\pm 5.7}$	$\mathbf{84.1}_{\pm 4.8}$	$75.3_{\pm 7.8}$	$75.3_{\pm 9.0}$	$81.8_{\pm 2.1}$	$86.2_{\pm 2.1}$	$72.2_{\pm 4.1}$
	GRPO+SALT	$\mathbf{96.2}_{\pm 1.7}$	$\mathbf{65.2}_{\pm 10.8}$	$\mathbf{93.1}_{\pm 4.7}$	$81.8_{\pm 8.3}$	$\mathbf{85.0}_{\pm 6.9}$	$\mathbf{77.0}_{\pm 4.7}$	$\mathbf{85.2}_{\pm 2.5}$	$\mathbf{86.9}_{\pm 0.6}$	$\mathbf{74.7}_{\pm 2.4}$
Base: Qwen2.5-7B-Instruct										
Prompting	Qwen2.5	33.4	21.6	19.3	6.9	2.8	3.2	14.8	26.4	7.8
	ReAct	48.5	35.4	34.3	13.2	18.2	17.6	31.2	46.2	19.5
	Reflexion	62.0	41.6	44.9	30.9	36.3	23.8	42.7	58.1	28.8
RL Training	PPO	$97.0_{\pm 1.2}$	$66.3_{\pm 6.7}$	$93.2_{\pm 2.3}$	$95.7_{\pm 3.1}$	$72.4_{\pm 4.0}$	$75.3_{\pm 1.8}$	$85.5_{\pm 1.3}$	$77.9_{\pm 4.3}$	$71.5_{\pm 5.0}$
	RLOO	$93.1_{\pm 1.6}$	$70.6_{\pm 6.9}$	$78.3_{\pm 3.8}$	$86.9_{\pm 7.0}$	$69.7_{\pm 3.9}$	$67.5_{\pm 7.9}$	$79.3_{\pm 0.7}$	$\mathbf{83.6}_{\pm 2.6}$	$\mathbf{76.8}_{\pm 3.6}$
	RLOO+SALT	$\mathbf{93.4}_{\pm 2.1}$	$\mathbf{72.6}_{\pm 7.5}$	$\mathbf{91.5}_{\pm 3.2}$	$\mathbf{90.3}_{\pm 3.5}$	$\mathbf{78.1}_{\pm 2.7}$	$\mathbf{76.4}_{\pm 4.4}$	$\mathbf{87.3}_{\pm 4.4}$	$83.1_{\pm 3.8}$	$75.2_{\pm 5.5}$
	GRPO	$\mathbf{88.9}_{\pm 3.9}$	$\mathbf{67.0}_{\pm 8.7}$	$73.7_{\pm 11.7}$	$\mathbf{76.1}_{\pm 7.7}$	$52.7_{\pm 8.1}$	$68.2_{\pm 6.4}$	$72.5_{\pm 5.5}$	$79.8_{\pm 4.0}$	$72.4_{\pm 5.5}$
	GRPO+SALT	$87.5_{\pm 4.9}$	$58.8_{\pm 14.7}$	$\mathbf{89.3}_{\pm 3.6}$	$75.3_{\pm 4.3}$	$\mathbf{70.2}_{\pm 5.5}$	$\mathbf{68.5}_{\pm 5.8}$	$\mathbf{77.8}_{\pm 1.7}$	$\mathbf{84.7}_{\pm 1.7}$	$\mathbf{76.2}_{\pm 3.4}$

Table 1: **Performance on ALFWorld and WebShop**. The results are reported as the average and standard deviation of three independent trainings.

commerce environment with real products and crowd-sourced instructions, where agents must navigate search, results, and product pages, performing actions like querying, filtering, and selecting, to complete purchase tasks. (2) **ALFWorld** (Shridhar et al., 2021): a platform that bridges abstract text-based environments with embodied tasks from ALFRED, requiring agents to reason before executing physical actions. (3) **AppWorld** (Trivedi et al., 2024): a suite of nine simulated consumer apps (e.g., email, payments, music, shopping, phone, file system), testing agents' ability to invoke complex APIs to fulfill user requests.

Baselines. We compare SALT against three categories of strong baselines: (1) *prompting-based* (training-free) methods, (2) *supervised fine-tuning* (SFT) approaches, and (3) *reinforcement learning* (RL) algorithms. To evaluate the effectiveness of our step-level advantage assignment, we integrate SALT into two representative group-based RL methods, GRPO (Shao et al., 2024) and RLOO (Ahmadian et al., 2024), and measure the performance gain it brings as a plug-and-play enhancement.

Training Setup. We use Qwen2.5-32B-Instruct for AppWorld, Qwen2.5-1.5B-Instruct/Qwen2.5-7B-Instruct model for WebShop and ALFWorld, and set the state history length to 3 when constructing the trajectory graph. The group size for all datasets is set to 8. For AppWorld, actions and states reside in a continuous textual space. We therefore use embedding-based semantic matching to determine step equivalence; implementation details and examples are provided in Appendix B.2. All group-based RL methods (including SALT variants) share identical hyperparameters to ensure fair comparison.

Evaluation Setup. For *AppWorld*, we report Task Goal Completion (TGC) and Scenario Goal Completion (SGC), success rates per task and per scenario, on both test-normal (Test-N) and test-challenge (Test-C) splits. Results are averaged over three independent runs, and we report both the mean and standard deviation. For *ALFWorld* and *WebShop*, we report metrics averaged over the last five checkpoints to account for high variance: for *ALFWorld*, average success rates per subtask and overall; for *WebShop*, both average normalized score and success rate.

Full experimental setup and details are provided in Appendix B.

4.2 Overall Results

Summarized in Tables 1 and 2, our results show that SALT consistently enhances group-based RL

Type	Method	Test-N		Test-C	
		TGC	SGC	TGC	SGC
Base: Closed-Source Models					
Prompting	GPT-4o	48.8	32.1	30.2	13
	OpenAI o1	61.9	41.1	36.7	19.4
	Llama 3 70B	24.4	17.9	7.0	4.3
	Qwen 2.5 32B	$39.2_{\pm 3.5}$	$18.6_{\pm 2.0}$	$21.0_{\pm 1.4}$	$7.5_{\pm 1.2}$
Base: Qwen2.5-32B-Instruct					
SFT	SFT-GT	$6.2_{\pm 0.7}$	$1.8_{\pm 0.0}$	$0.8_{\pm 0.2}$	$0.1_{\pm 0.3}$
	RFT	$47.9_{\pm 3.7}$	$26.4_{\pm 2.3}$	$26.4_{\pm 1.8}$	$11.4_{\pm 2.3}$
	EI	$58.3_{\pm 2.8}$	$36.8_{\pm 6.0}$	$32.8_{\pm 0.7}$	$17.6_{\pm 1.3}$
RL Training	DPO-MCTS	$57.0_{\pm 1.5}$	$31.8_{\pm 4.2}$	$31.8_{\pm 1.3}$	$13.7_{\pm 1.5}$
	DMPO	$59.0_{\pm 1.2}$	$36.6_{\pm 4.7}$	$36.3_{\pm 1.8}$	$18.4_{\pm 2.3}$
	PPO	$50.8_{\pm 3.7}$	$28.9_{\pm 7.9}$	$26.4_{\pm 0.5}$	$10.5_{\pm 2.1}$
	RLOO	$59.8_{\pm 2.2}$	$37.5_{\pm 2.8}$	$33.8_{\pm 0.5}$	$14.4_{\pm 1.6}$
	RLOO+SALT	$\mathbf{61.3}_{\pm 2.7}$	$\mathbf{39.3}_{\pm 4.1}$	$\mathbf{37.4}_{\pm 0.9}$	$\mathbf{18.9}_{\pm 1.2}$
	GRPO	$61.5_{\pm 2.9}$	$41.4_{\pm 3.8}$	$36.3_{\pm 0.2}$	$17.0_{\pm 0.9}$
	GRPO+SALT	$\mathbf{66.2}_{\pm 2.5}$	$\mathbf{47.9}_{\pm 4.1}$	$\mathbf{36.8}_{\pm 1.5}$	$\mathbf{20.9}_{\pm 1.8}$

Table 2: **Performance on AppWorld**. The results are reported as the average and standard deviation of three independent evaluations.

algorithms, though its effectiveness depends on model scale and task structure.

On ALFWorld, SALT improves both GRPO and RLOO with the Qwen2.5-1.5B model, boosting GRPO from 81.8% → 85.2% (+3.4pp) and RLOO from 79.2% → 84.0% (+4.8pp), with large subtask gains such as *Cool* (+9.7pp) and *Heat* (+10.3pp). With the 7B model, SALT continues to help: RLOO improves from 79.3% → 87.3% (+8.0pp), and GRPO from 72.5% → 77.8% (+5.3pp). On WebShop, SALT benefits the 1.5B model (RLOO: 71.6% → 75.8%, +4.2pp), but on the 7B model, RLOO slightly declines (76.8% → 75.2%, –1.6pp) while GRPO still gains (+3.8pp), suggesting that SALT's credit assignment may interact differently with larger-model optimization dynamics.

The gains are more pronounced on the complex AppWorld benchmark. SALT consistently lifts RLOO, e.g., on Test-C, TGC improves from 33.8% → 37.4% (+3.6pp) and SGC from 14.4% → 18.9% (+4.5pp). GRPO+SALT achieves the best overall performance, with TGC on Test-N rising from 61.5% → 66.2% (+4.7pp) and on Test-C from 30.1% → 36.8% (+6.7pp), demonstrating SALT's ability to enhance generalization in challenging, novel tasks.

Notably, SALT outperforms PPO, despite PPO using a critic, validating that fine-grained credit assignment from outcome rewards alone is both feasible and highly effective. Together, these results confirm SALT's robustness across model sizes and task complexities.

4.3 Compute Efficiency

SALT is designed as a lightweight, plug-and-play module that integrates seamlessly into existing group-based RL pipelines — inserted after advantage computation and before policy update. As shown in Figure 4, we break down the computational cost of each component during training on ALFWorld using GRPO with Qwen2.5-1.5B-Instruct. The dominant time-consuming operations are rollout (240s) and policy update (30s), while computing old and reference log probabilities each takes about 8s. In contrast, the original advantage computation requires only 0.05s. Remarkably, SALT's step-level advantage generation (SAG) adds just 0.15s — a negligible increase over the baseline, and less than three times the cost of standard advantage estimation. This demonstrates that SALT introduces minimal computational overhead, making it highly efficient and scalable for large-scale agent training.

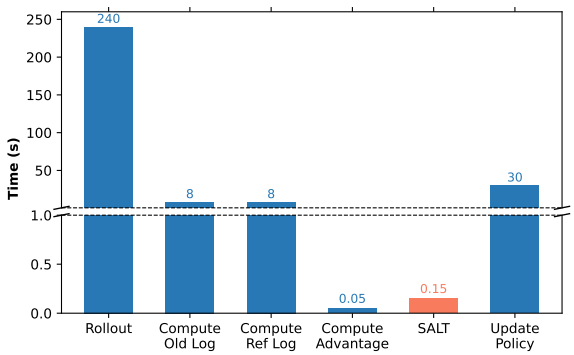

Figure 4: Time costs for different components per step.

4.4 Impact of Model Size

We examine how model size affects performance by comparing Qwen2.5-1.5B-Instruct and Qwen2.5-7B-Instruct on ALFWorld, as shown in Figure 5. Counterintuitively, the smaller 1.5B model eventually matches or exceeds the 7B variant, despite initial lag. We attribute this phenomena to the *exploration-exploitation dynamics*. First, smaller models exhibit a more pronounced *exploration bias* during early training. Due to their limited capacity, they are less able to precisely fit observed reward patterns, which counterintuitively encourages broader exploration of the action space. This often leads them to discover high-reward trajectories that larger models overlook. As training progresses and SALT provides increasingly accurate step-level advantages, these exploratory gains are rapidly con-

solidated into stable, high-performing policies. In contrast, larger models demonstrate stronger *exploitation capabilities* early on: they quickly latch onto rewarding action sequences and refine them. However, this strength can become a liability as their high capacity and precise fitting may cause them to converge prematurely to local optima.

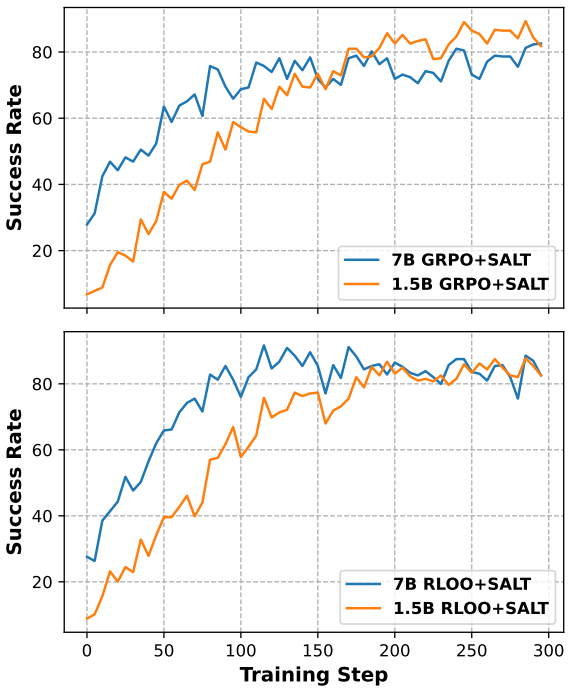

Figure 5: Success rate over training steps for 1.5B and 7B models.

4.5 Impact of State History Length

The state history length h is a key hyperparameter in SALT, determining the granularity of merge and diverge operations. We define the state as $s_t = (a_{t-h+1}, o_{t-h+1}, \ldots, a_t, o_t)$, where h controls how much recent context is considered. This design is particularly important in long-horizon tasks (e.g., over 30 turns in AppWorld), where agents may make a mistake early in the trajectory but still execute correct subsequent steps. For example, as shown in LOOP (Chen et al., 2025a), an agent might fail to retrieve a real password but successfully use a placeholder to proceed with the correct login logic. Although such trajectories ultimately fail, many of the steps after the mistake are correct and shared with successful trajectories. By using a recent history window, SALT can merge and credit these correct shared substructures even within otherwise incorrect trajectories, rather than treating the entire trajectory as uniformly negative.

We evaluate the impact of h on ALFWorld using GRPO with Qwen2.5-1.5B-Instruct, varying $h \in \{1, 2, 3, 4, 5\}$. As shown in Figure 7, $h = 1$ leads to inferior performance due to overly coarse state definitions, which cause excessive and noisy merges between semantically different situations. Increasing h to moderate values ($h = 2$ or $h = 3$) significantly improves performance by striking a balance: the state representation becomes expressive enough to distinguish truly different decision points while still allowing meaningful merging of shared substructures across trajectories. However, when h becomes too large (e.g., $h = 5$), the matching criterion becomes overly strict, resulting in very few merge operations and causing SALT to degenerate toward trajectory-level advantage assignment.

This trend is consistent with the merge rate dynamics in Figure 6: larger h yields lower merge rates, while merge rates increase over training as the policy becomes more deterministic, ensuring that SALT naturally transitions from safe baseline behavior to effective step-level credit assignment.

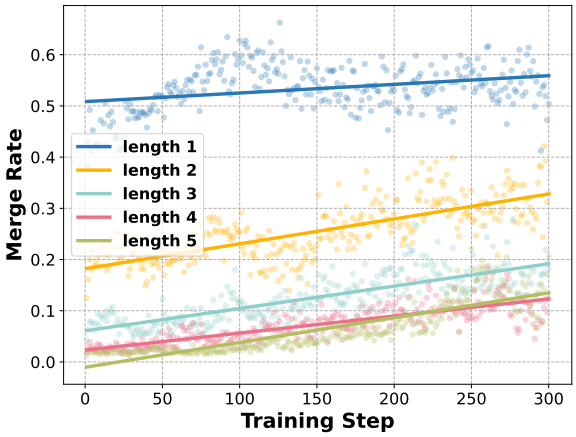

Figure 6: Merge rate across training steps for different state history lengths.

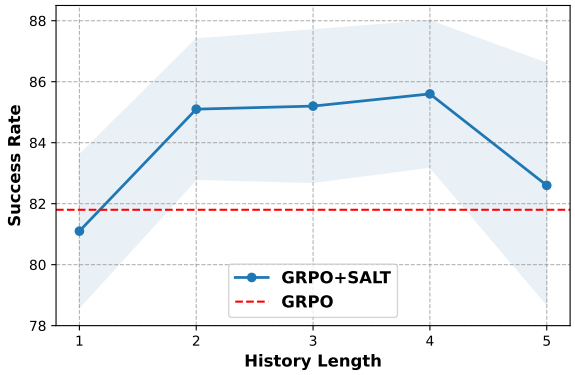

Figure 7: Success rate with varying state history lengths.

4.6 Impact of Group Size

Group size is a critical hyperparameter in group-based RL, balancing training stability against com-

putational cost. Larger groups provide more reliable reward baselines, leading to stable gradients and better generalization but at higher memory and compute. In SALT, this effect is amplified: the merge/diverge operations and step-level advantage rectification make the algorithm sensitive to group diversity. To investigate, we train GRPO+SALT on ALFWorld with Qwen2.5-1.5B-Instruct, testing group sizes $\{4, 8, 12, 16\}$. As shown in Figure 8, with a small group size of 4, GRPO slightly outperforms SALT, likely due to insufficient diversity for meaningful graph construction. However, as group size increases, SALT consistently surpasses the baseline, demonstrating its ability to leverage richer trajectory structure and confirming that larger groups enable more effective credit assignment through enhanced graph connectivity.

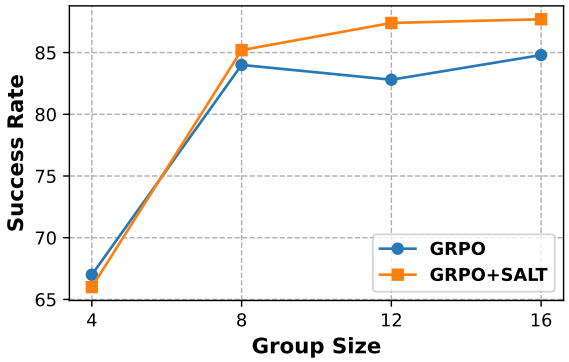

Figure 8: Success rate across different group sizes.

5 Conclusion

In this paper, we present SALT, a lightweight framework for assigning fine-grained, step-level advantages to improve group-based reinforcement learning for large language model agents. By constructing a trajectory graph that distinguishes shared and divergent steps across rollouts, SALT refines advantage assignment without requiring additional supervision, reward models, or architectural changes. As a plug-and-play module, it integrates seamlessly into existing RL pipelines such as GRPO and RLOO. Experiments on ALFWorld, WebShop, and AppWorld demonstrate consistent improvements across model sizes, highlighting SALT's effectiveness and potential for future step-level RL methods.

6 Future Work

While SALT demonstrates strong empirical performance using only outcome-based supervision, several promising extensions remain.

First, a natural direction is to explore hybrid credit assignment strategies that combine SALT's efficient step-level advantage refinement with additional external signals, such as process reward models or lightweight verifiers. Such a hybrid approach could preserve SALT's scalability and plug-and-play nature while incorporating richer semantic judgments when available, potentially further improving learning stability and sample efficiency.

Second, we believe SALT can be extended beyond interactive agent benchmarks with appropriate state-action abstractions. For mathematical reasoning tasks, actions can be defined as tool invocations (e.g., structured Python calls) or segmented reasoning steps, with semantic similarity measured using embedding models. For code generation, Abstract Syntax Trees (ASTs) can be leveraged to define structural equivalence between steps, enabling trajectory graph construction based on syntax rather than surface-level text. More broadly, SALT's applicability depends on carefully designing equivalence criteria for states and actions, suggesting a general framework for step-level credit assignment across diverse domains.

Limitations

Although SALT demonstrates strong empirical performance, several limitations remain. First, while SALT is algorithm-agnostic in design, its application may require adaptation to specific environments. For benchmarks with discrete, well-structured action and state spaces, such as ALFWorld and WebShop, SALT can be applied directly with minimal preprocessing. However, in environments like AppWorld, where actions and observations reside in a continuous textual space, additional components, such as embedding models or clustering mechanisms, are necessary to meaningfully construct the trajectory graph. Second, SALT introduces new hyperparameters, such as the history window length, which, while empirically not highly sensitive, still require tuning and may affect performance. Third, our current implementation integrates SALT only with RLOO and GRPO. Its compatibility with more advanced group-based policy optimization methods (e.g., DAPO (Yu et al., 2025), GSPO (Zheng et al., 2025)) has not yet been explored.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms. *arXiv*.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023. Fireact: Toward language agent fine-tuning. *arXiv*.

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. 2025a. Reinforcement learning for long-horizon interactive llm agents. *arXiv*.

Yongchao Chen, Yueying Liu, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, and Chuchu Fan. 2025b. R1-code-interpreter: Training llms to reason with code via supervised and reinforcement learning. *arXiv*.

Sanjiban Choudhury. 2025. Process reward models for llm agents: Practical framework and directions. *arXiv*.

Guanting Dong, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. 2025. Tool-star: Empowering llm-brained multi-tool reasoner via reinforcement learning. *arXiv*.

Lutfi Eren Erdogan, Hiroki Furuta, Sehoon Kim, Nicholas Lee, Suhong Moon, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. In *ICML*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025a. Retool: Reinforcement learning for strategic tool use in llms. *arXiv*.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025b. Group-in-group policy optimization for llm agent training. *arXiv*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv*.

Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, and 1 others. 2025. $\pi_{0.5}$: a vision-language-action model with open-world generalization. *arXiv*.

Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. 2025. treegrpo. *arXiv*.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv*.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. 2025a. Agent-oriented planning in multi-agent systems. In *ICLR*.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, and 1 others. 2025b. Websailor: Navigating super-human reasoning for web agent. *arXiv*.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025. Understanding r1-zero-like training: A critical perspective. *arXiv*.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl. Notion Blog.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *ICLR*.

Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. 2025. Agentic reasoning and tool integration for llms via reinforcement learning. *arXiv*.

Hao Sun, Zile Qiao, Jiayan Guo, Xuanbo Fan, Yingyan Hou, Yong Jiang, Pengjun Xie, Fei Huang, and Yan Zhang. 2025. Zerosearch: Incentivize the search capability of llms without searching. *arXiv*.

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

Hanlin Wang, Chak Tou Leong, Jiashuo Wang, Jian Wang, and Wenjie Li. 2025a. Spa-rl: Reinforcing llm agents via stepwise progress attribution. *arXiv*.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, and 1 others. 2025b. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv*.

Hongru Wang, Cheng Qian, Wanjun Zhong, Xiusi Chen, Jiahao Qiu, Shijue Huang, Bowen Jin, Mengdi Wang, Kam-Fai Wong, and Heng Ji. 2025c. Acting less is reasoning more! teaching model to act efficiently. *arXiv*.

Jiawei Wang, Jiacai Liu, Yuqian Fu, Yingru Li, Xintao Wang, Yuan Lin, Yu Yue, Lin Zhang, Yang Wang, and Ke Wang. 2025d. Harnessing uncertainty: Entropy-modulated policy gradients for long-horizon llm agents. *arXiv*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Jialong Wu, Baixuan Li, Runnan Fang, Wenbiao Yin, Liwen Zhang, Zhengwei Tao, Dingchu Zhang, Zekun Xi, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. 2025. Webdancer: Towards autonomous information seeking agency. *arXiv*.

Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, and 1 others. 2024. Agentgym: Evolving large language model-based agents across diverse environments. *arXiv*.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, Baodai Huang, Honglin Guo, Jiaqi Liu, Rui Zheng, Junjie Ye, Jiazheng Zhang, Wenxiang Chen, and 1 others. 2025. Agentgym-rl: Training llm agents for long-horizon decision making through multi-turn reinforcement learning. *arXiv*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: agent-computer interfaces enable automated software engineering. In *NeurIPS*.

Zhicheng Yang, Zhijiang Guo, Yinya Huang, Xiaodan Liang, Yiwei Wang, and Jing Tang. 2025. treerpo. *arXiv*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *NeurIPS*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *ICLR*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv*.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2024. Agenttuning: Enabling generalized agent abilities for llms. In *ACL*.

Zijing Zhang, Ziyang Chen, Mingxiao Li, Zhaopeng Tu, and Xiaolong Li. 2025. Rlvmr: Reinforcement learning with verifiable meta-reasoning rewards for robust long-horizon agents. *arXiv*.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, and 1 others. 2025. Group sequence policy optimization. *arXiv*.

Yifei Zhou, Song Jiang, Yuandong Tian, Jason Weston, Sergey Levine, Sainbayar Sukhbaatar, and Xian Li. 2025. Sweet-rl: Training multi-turn llm agents on collaborative reasoning tasks. *arXiv*.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. 2025. Knowagent: Knowledge-augmented planning for llm-based agents. In *NACCL*.

A Related Work

LLM Agents. The paradigm of LLMs has evolved beyond simple text generation to the development of LLM agents, which are autonomous systems designed to reason and perform complex, multi-step tasks. An agent's core functionality is centered on a central LLM that acts as its brain, enabling it to interpret a user's high-level goal, formulate a strategic plan, and execute that plan by interacting with its environment. This process typically involves a reasoning phase (Yao et al., 2023; Shinn et al., 2023), where the agent breaks down a task into actionable sub-goals; a planning phase (Erdogan et al., 2025; Li et al., 2025a), where it determines the sequence of operations; and an action phase, where it leverages external tools, such as web search APIs (Jin et al., 2025; Sun et al., 2025), code interpreters (Wang et al., 2024; Feng et al., 2025a), or knowledge bases (Zhu et al., 2025), to gather information or manipulate data. By iteratively observing the results of its actions and adjusting its plan, an LLM agent demonstrates a significant step toward more generalized and autonomous problem-solving.

Agent Training. Early approaches for building agents primarily leveraged sophisticated prompting strategies and external tools to enhance performance on complex tasks, as exemplified by methods like ReAct (Yao et al., 2023). However, models with smaller parameter counts often lack the requisite foundational capabilities for such complex reasoning. To address this limitation, some studies employ supervised fine-tuning (SFT) to enhance the models' decision-making abilities (Schick et al., 2023; Zeng et al., 2024; Chen et al., 2023). More recently, there has been a growing focus on end-to end reinforcement learning for training agents (Dong et al., 2025; Singh et al., 2025; Chen et al., 2025b), which learn through direct, adaptive online interaction with an environment, thereby obviating the need for complex data preparation. Unlike supervised fine-tuning, RL naturally aligns with the objective of maximizing cumulative rewards through agent-environment interactions. This makes RL particularly well-suited for agentic tasks. Therefore, in this work, we focus on RL-based approaches to further enhance the capabilities of LLM agents.

Step-level Supervision in RL. In multi-step agent training, LLM agents must interact with the environment across multiple steps before receiving a final reward at the end of the trajectory (Wang et al., 2025c; Chen et al., 2025b), making it challenging to assess the quality of intermediate actions. This lack of intermediate feedback hinders RL training and often requires extensive interactions with the environment to evaluate the trajectory. Incorporating step-level supervision or intermediate signals has been verified as effective in greatly enhancing the effectiveness and efficiency of LLM agent training (Zhou et al., 2025; Wang et al., 2025a; Feng et al., 2025b; Choudhury, 2025; Wang et al., 2025d). However, existing methods either introduce additional information to guide the generation of step-level rewards (Zhou et al., 2025; Feng et al., 2025b; Wang et al., 2025d) or use another explicit process reward model to assign intermediate rewards (Wang et al., 2025a; Choudhury, 2025) which require deliberate designs and extra compute overhead. As a result, in this paper, we propose a plug-and-play module which can be seamlessly integrated into existing RL algorithms to generate step-level feedbacks for multi-step training, without introducing any extra information or model.

Tree-based Policy Optimization. Recent works have explored tree-structured rollouts to improve credit assignment and exploration in reinforcement learning for LLMs. Tree-GRPO (Ji et al., 2025) and TreeRPO (Yang et al., 2025) construct branching rollout trees during inference, where intermediate nodes represent partial trajectories and leaves receive outcome rewards that are propagated through the tree structure. These methods explicitly modify the rollout procedure and require maintaining a search tree, introducing additional computational and engineering complexity. In contrast, SALT does not alter the rollout or inference process. It operates purely as a post-hoc advantage refinement module on sampled trajectories, leveraging shared substructures across completed rollouts via trajectory graph merging. This design makes SALT lightweight and plug-and-play, and fundamentally different from tree-based methods that rely on explicit branching, search, or reward backpropagation over tree nodes. Overall, while tree-based approaches aim to improve exploration and reward propagation by restructuring rollouts into trees, SALT focuses on efficient step-level credit assignment using only sparse outcome rewards, without introducing additional rollouts, search procedures, or critic models.

B Experiment Details

B.1 Baselines

For WebShop and ALFWorld, we adopt the results for prompting-based methods from Feng et al. (2025b). Performance of PPO, RLOO, and GRPO is evaluated using our own implementations to ensure fair and consistent comparisons under identical training conditions.

For AppWorld, we adopt the results for prompting-based methods and SFT-based methods from Chen et al. (2025a). Performance of RLOO and GRPO is evaluated using our own implementations to ensure fair and consistent comparisons under identical training conditions.

B.2 Equivalence Matching in AppWorld

Unlike environments with discrete and well-structured state-action spaces (e.g., ALFWorld), AppWorld operates in a continuous textual action space. Agent actions are expressed as free-form code snippets or natural language instructions, where semantically equivalent behaviors can differ substantially in surface form. As a result, exact string matching is insufficient for identifying shared steps across trajectories when constructing the trajectory graph in SALT.

To address this challenge, we employ embedding-based semantic matching for both states and actions in AppWorld. Concretely, each state-action pair is encoded using a sentence embedding model, and two steps are considered equivalent if their cosine similarity exceeds a predefined threshold. This allows SALT to correctly merge steps that implement the same underlying operation despite syntactic variations.

Figure 9 presents a representative example from AppWorld. Both responses aim to retrieve the Venmo account password by invoking the same supervisor API, yet they differ in variable naming, control flow structure, and coding style. While exact matching would treat these steps as distinct, embedding-based matching correctly identifies them as semantically equivalent, enabling effective merge operations during trajectory graph construction.

B.3 Hyperparameters

All models (versions with SALT and without SALT) are trained using the same hyperparameter configuration to isolate the effect of our proposed advantage refinement. Detailed settings are provided in Table 4.

Method	Test-N TGC			Test-N SGC		
	D1	D2	D3	D1	D2	D3
RLOO	$81.6_{\pm 1.9}$	$63.0_{\pm 6.8}$	$37.7_{\pm 3.4}$	$59.2_{\pm 6.8}$	$31.2_{\pm 7.6}$	$\mathbf{22.6}_{\pm 7.0}$
RLOO+SALT	$\mathbf{81.7}_{\pm 3.9}$	$\mathbf{67.9}_{\pm 4.6}$	$37.7_{\pm 4.2}$	$\mathbf{67.3}_{\pm 5.1}$	$\mathbf{38.7}_{\pm 6.1}$	$14.3_{\pm 4.2}$
GRPO	$\mathbf{85.3}_{\pm 4.4}$	$67.5_{\pm 3.8}$	$35.5_{\pm 2.5}$	$69.5_{\pm 6.1}$	$38.7_{\pm 7.3}$	$18.1_{\pm 3.5}$
GRPO+SALT	$84.9_{\pm 3.0}$	$\mathbf{76.2}_{\pm 5.3}$	$\mathbf{41.6}_{\pm 2.3}$	$\mathbf{72.6}_{\pm 5.1}$	$\mathbf{50.0}_{\pm 8.8}$	$\mathbf{23.8}_{\pm 5.2}$
Method	**Test-C TGC**			**Test-C SGC**		
	D1	D2	D3	D1	D2	D3
RLOO	$69.4_{\pm 1.6}$	$28.1_{\pm 1.6}$	$26.5_{\pm 2.2}$	$45.8_{\pm 5.1}$	$9.5_{\pm 3.5}$	$7.7_{\pm 2.1}$
RLOO+SALT	$\mathbf{74.4}_{\pm 3.0}$	$\mathbf{30.9}_{\pm 3.3}$	$\mathbf{28.5}_{\pm 2.3}$	$\mathbf{56.6}_{\pm 4.9}$	$\mathbf{10.4}_{\pm 1.5}$	$\mathbf{11.8}_{\pm 3.0}$
GRPO	$\mathbf{75.4}_{\pm 1.3}$	$30.6_{\pm 0.5}$	$26.1_{\pm 1.1}$	$\mathbf{58.3}_{\pm 3.3}$	$8.0_{\pm 0.0}$	$8.7_{\pm 0.7}$
GRPO+SALT	$70.3_{\pm 1.7}$	$\mathbf{34.6}_{\pm 0.5}$	$26.1_{\pm 3.2}$	$48.6_{\pm 3.9}$	$\mathbf{18.6}_{\pm 2.4}$	$\mathbf{12.3}_{\pm 4.3}$

Table 3: Detailed performance on AppWorld with respect to the task difficulty.

B.4 More Results

A closer inspection of the per-difficulty results in Table 3 reveals that SALT consistently enhances performance on medium- and high-difficulty tasks (D2 and D3), while maintaining competitive results on easy tasks (D1). For example, on Test-N, GRPO+SALT improves TGC by +8.7pp on D2 (67.5% $\rightarrow$ 76.2%) and +6.1pp on D3 (35.5% $\rightarrow$ 41.6%), with even more substantial SGC gains—+11.3pp on D2 (38.7% $\rightarrow$ 50.0%) and +5.7pp on D3 (18.1% $\rightarrow$ 23.8%). On the more challenging Test-C, RLOO+SALT increases SGC on D3 from 7.7% to 11.8% (+4.1pp), and GRPO+SALT achieves a striking +10.6pp SGC improvement on D2 (8.0% $\rightarrow$ 18.6%). These trends confirm that SALT's advantage-based credit assignment is particularly effective in complex, high-difficulty scenarios where precise action sequencing and long-horizon reasoning are required. Minor fluctuations on D1 (e.g., GRPO+SALT TGC on Test-C: 75.4% $\rightarrow$ 70.3%) are outweighed by consistent gains on harder tasks, underscoring SALT's value in pushing the boundary of agent capabilities in realistic, challenging environments.

C SALT Pseudocode

This appendix provides the pseudocode for SALT to facilitate reproducibility and implementation. Algorithm 1 describes the construction of the trajectory graph and the refinement of trajectory-level advantages into step-level advantages via merge-based averaging. Algorithm 2 illustrates how SALT can be seamlessly integrated into a representative group-based reinforcement learning algorithm (GRPO) without modifying the rollout or reward computation. Together, these algorithms high-

Example: Semantic-equivalent actions in AppWorld (continuous action space)

Response A (list comprehension style):

```
account_credentials = apis.supervisor.show_account_credentials()
venmo_password = [account for account in account_credentials
   if account["account_name"] == "venmo"][0]
print(venmo_password)
```

Response B (loop style):

```
creds = apis.supervisor.show_account_credentials()
for entry in creds:
   if entry["account_name"] == "venmo":
      password = entry["password"]
      break
print(password)
```

Figure 9: Two semantically equivalent AppWorld actions with different surface forms. Both responses invoke `show_account_credentials`, identify the Venmo account entry, and retrieve the password. Embedding-based semantic matching enables SALT to merge them during trajectory graph construction in continuous action spaces.

Hyperparameter	WebShop		ALFWorld	
	Qwen2.5-1.5B-Instruct	Qwen2.5-7B-Instruct	Qwen2.5-1.5B-Instruct	Qwen2.5-7B-Instruct
Mini-batch size	64	32	256	128
Max interaction steps	15		50	
Max prompt length	4096		2048	
State history length	3			
Rollout temperature	1.0			
Evaluation temperature	0.4			
Group Size	8			
Learning rate	1e-6			
Max response length	512			
KL loss coefficient	0.01			
Training steps	300			
Clip ratio	0.2			

Table 4: Hyperparameter settings used across all experiments for WebShop and ALFWorld.

light SALT as a lightweight, plug-and-play post-processing module for step-level credit assignment.

D Case Study

To illustrate how SALT refines step-level advantages, we visualize four representative cases from ALFWorld. Each example shows the contextual state history, the current action step, and the transformation from original (trajectory-level) to updated (SALT-adjusted) advantage. These cases highlight SALT's ability to distinguish between *shared neutral steps* and *distinct decisive steps* through trajectory graph analysis.

Hyperparameter	AppWorld `Qwen2.5-32B-Instruct`
Mini-batch size	32
Max interaction steps	40
Max prompt length	28048
State history length	3
Rollout temperature	1.0
Evaluation temperature	1.0
Group Size	8
Learning rate	1e-6
Max response length	1500
KL loss coefficient	0
Training steps	100
Clip ratio	0.2

Table 5: Hyperparameter settings used across all experiments for AppWorld.

Algorithm 1 SALT: Step-level Advantage Refinement via Trajectory Graph

Require: Task $\boldsymbol{q}$; group rollouts $\{\boldsymbol{\tau}_i\}_{i=1}^{G}$ where $\boldsymbol{\tau}_i = (\boldsymbol{q}, (a_1^i, o_1^i), \ldots, (a_{n_i}^i, o_{n_i}^i))$; trajectory-level advantages $\{\hat{A}^i\}_{i=1}^{G}$; history length h

Ensure: Step-level advantages $\{\hat{A}_t'^i\}$ for all steps (i, t)

Initialize an empty directed graph $\mathcal{G} = (V, E)$ with root node $\boldsymbol{q}$
Initialize an index map $\mathcal{I}$ from transition key κ to a list of occurrences ▷ κ canonically identifies a mergeable transition
for $i \leftarrow 1$ to G **do**
  for $t \leftarrow 1$ to n_i **do**
    Construct $s_{t-1}^i \leftarrow \textsc{State}(\boldsymbol{\tau}_i, t-1, h)$ and $s_t^i \leftarrow \textsc{State}(\boldsymbol{\tau}_i, t, h)$
    Define transition key $\kappa \leftarrow (s_{t-1}^i,\ a_t^i,\ s_t^i)$
    Add nodes s_{t-1}^i, s_t^i and edge $(s_{t-1}^i \xrightarrow{a_t^i} s_t^i)$ into $\mathcal{G}$
    Append occurrence $(i, t, \hat{A}^i)$ into $\mathcal{I}[\kappa]$
  end for
end for
Initialize $\hat{A}_t'^i \leftarrow \hat{A}^i$ for all (i, t)
for all keys κ with $|\mathcal{I}[\kappa]| > 1$ **do**
  Compute $\bar{A} \leftarrow \frac{1}{|\mathcal{I}[\kappa]|} \sum_{(i,t,A)\in\mathcal{I}[\kappa]} A$
  for all $(i, t, A) \in \mathcal{I}[\kappa]$ **do**
    Set $\hat{A}_t'^i \leftarrow \bar{A}$ ▷ average advantage over shared steps
  end for
end for
return $\{\hat{A}_t'^i\}$

Algorithm 2 Integrating SALT into Group-based RL (Example: GRPO)

Require: Policy $\pi_{\theta_{\text{old}}}$; task prompts $\{\boldsymbol{q}\}$; group size G; history length h

for all task $\boldsymbol{q}$ **do**
  Sample $\{\boldsymbol{\tau}_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid \boldsymbol{q})$
  Compute outcome rewards $\{R(\boldsymbol{\tau}_i)\}_{i=1}^{G}$
  Compute trajectory-level advantages $\hat{A}^i \leftarrow \dfrac{R(\boldsymbol{\tau}_i) - \textsc{Mean}(\{R(\boldsymbol{\tau}_j)\}_{j=1}^{G})}{\textsc{Std}(\{R(\boldsymbol{\tau}_j)\}_{j=1}^{G})}$ for $i = 1..G$
  Refine $\{\hat{A}^i\} \rightarrow \{\hat{A}_t'^i\}$ via $\text{SALT}(\boldsymbol{q}, \{\boldsymbol{\tau}_i\}, \{\hat{A}^i\}, h)$
  Update θ using the GRPO objective with per-step advantages $\hat{A}_t'^i$
end for

Example 1: Shared Step → Advantage Averaged

History state:
-= Welcome to TextWorld, ALFRED! =-
You are in the middle of a room. Looking quickly around you, you see a cabinet 15, a cabinet 14, a cabinet 13, a cabinet 12, a cabinet 11, a cabinet 10, a cabinet 9, a cabinet 8, a cabinet 7, a cabinet 6, a cabinet 5, a cabinet 4, a cabinet 3, a cabinet 2, a cabinet 1, a coffeemachine 1, a countertop 4, a countertop 3, a countertop 2, a countertop 1, a drawer 4, a drawer 3, a drawer 2, a drawer 1, a fridge 1, a garbagecan 1, a microwave 1, a sinkbasin 1, a stoveburner 4, a stoveburner 3, a stoveburner 2, a stoveburner 1, and a toaster 1.
Your task is to: heat some cup and put it in cabinet.
History step:
go to countertop 1
History state:
You arrive at countertop 1. On the countertop 1, you see a butterknife 1, a papertowelroll 1, and a tomato 1.,

Step:
go to stoveburner 1

Original advantage: 1.007
Updated advantage: 0.008

Figure 10: SALT reduces the advantage of this step because it frequently appears across both successful and failed rollouts. As a commonly executed transition (not task-determinative), its credit is averaged to avoid over-rewarding incidental actions.

Example 2: Shared Step in Mixed Outcomes → Advantage Neutralized

History state:
You arrive at countertop 1. On the countertop 1, you see a fork 2, a ladle 1, a pan 1, a pot 1, a soapbottle 2, a spatula 1, a tomato 1, a window 3, and a window 1.
History step:
go to fridge 1
History state:
You arrive at fridge 1. The fridge 1 is closed.
History step:
open fridge 1
History state:
You open the fridge 1. The fridge 1 is open. In it, you see a potato 1.

Step:
go to countertop 2

Original advantage: -1.854
Updated advantage: 0.138

Figure 11: Although this step originally received a large penalty (from a failed trajectory), SALT revises its advantage upward because it also appears in successful rollouts. This averaging prevents punishing actions that are contextually neutral or necessary.

Example 3: Distinct Step in Successful Trajectories → Advantage Preserved

History state:
You arrive at microwave 1. The microwave 1 is closed.
History step:
heat cup 2 with microwave 1
History state:
You heat the cup 2 using the microwave 1.
History step:
go to cabinet 1
History state:
You arrive at cabinet 1. The cabinet 1 is closed.

Step:
open cabinet 1

Original advantage: 1.007
Updated advantage: 1.007

Figure 12: SALT preserves the high advantage of this step because it occurs predominantly in successful trajectories and represents a task-critical action. No averaging is applied, as it is identified as a decisive, non-shared step.

Example 4: Distinct Step in Failed Trajectories → Advantage Preserved

History state:
You open the drawer 1. The drawer 1 is open. In it, you see a knife 2.
History step:
go to countertop 1
History state:
You arrive at countertop 1. On the countertop 1, you see a fork 2, a ladle 1, a pan 1, a pot 1, a soapbottle 2, a spatula 1, a tomato 1, a window 3, and a window 1.
History step:
go to fridge 1
History state:
You arrive at fridge 1. The fridge 1 is open. In it, you see a potato 1.

Step:
go to countertop 1

Original advantage: -1.854
Updated advantage: -1.854

Figure 13: SALT retains the negative advantage because this step appears primarily in failed trajectories and reflects inefficient backtracking. As a behaviorally distinct and detrimental pattern, it is not averaged with successful paths.