Training Language Agents to Learn from Experience

Yuval Shalev
University of Cambridge
ys716@cam.ac.uk

Zifeng Ding
University of Cambridge
zd320@cam.ac.uk

Mateja Jamnik
University of Cambridge
mj201@cam.ac.uk

Abstract

Language agents can adapt from experience in interactive environments, but current reflection-based methods can only self-correct within a single task instance. Whether such experience can be distilled into reusable lessons that improve performance on future unseen tasks remains unclear. We address this problem by introducing the In-context Training (ICT) task, a framework for evaluating cross-task self-improvement in language agents. In ICT, a reflector model observes trajectories collected by an actor model and generates system prompts intended to improve the actor's performance on future unseen tasks. We then propose an RL-based training pipeline for learning such reflections directly from experience, without human-provided examples. Across ALFWorld and MiniHack, our trained reflectors outperform an untrained baseline on most held-out task families, showing that the ability to learn from experience can itself be learned. In some cases, we observe generalisation beyond the benchmark on which the reflector was trained, to substantially different environments. Finally, we introduce MetaGym, a generic Python library for constructing meta-environments, enabling future research on self-improving language agents.

1 Introduction

In recent years, there has been a growing interest in using large language models (LLMs) as decision-making agents in interactive environments [18, 17]. In this setting, LLMs act as agents that leverage their language capabilities to interact with external environments, producing tokens that are interpreted as actions [21]. When faced with an unfamiliar environment, a general-purpose agent should be able to learn and adapt its behaviour from experience in order to find an optimal policy for action [16, 11, 23, 14]. Humans handle this task with ease. For example, when playing chess for the first time, a person may begin with an initial strategy in mind and, after each game, reflect on their experience to refine that strategy for future matches.

In LLM-based agents, the agent's policy is determined not only by the model's parameters, but also by the prompt provided to the model, which specifies how the agent should act in the environment [21, 3]. Learning and adapting may therefore be achieved by modifying the prompt provided to the model, making prompt optimisation a natural mechanism for self-learning in language agents. This approach has three main advantages: (1) it avoids the compute-heavy pipeline of fine-tuning LLMs for each specific task through gradient descent; (2) it leverages the linguistic abilities of the model to support rapid learning from only a small number of experiences; and (3) it offers a degree of transparency, since the accumulated experience is represented solely in natural language and can be directly inspected.

Recent studies have explored self-improvement in language agents through autonomous prompt modification, using self-reflection mechanisms [14, 22, 5–7]. In this setting, an LLM generates a reflection following an unsuccessful attempt at an interactive task. The reflection considers both the agent's actions and the resulting feedback from the environment, and is then used to guide the agent's next attempt by providing hints and suggesting how to avoid similar mistakes. This yields a

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

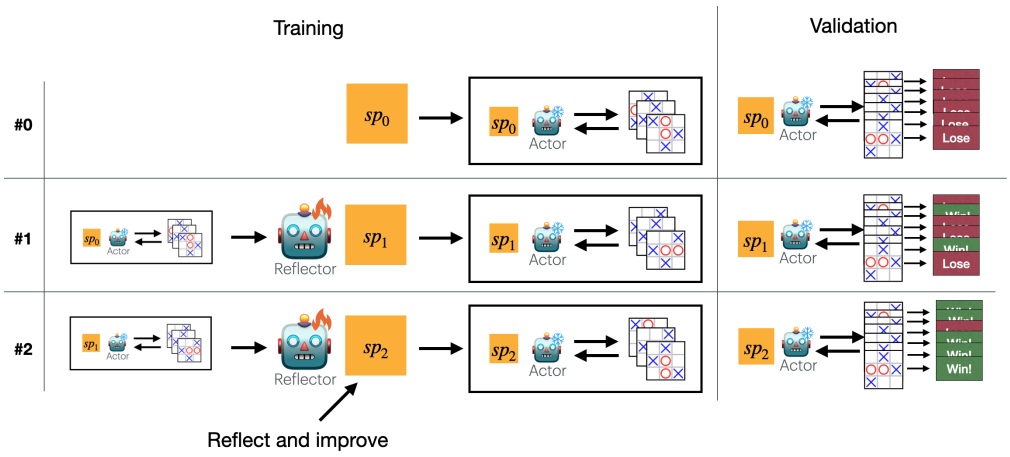


Figure 1: The In-context Training (ICT) task. In each turn, the reflector model generates a new system prompt sp_i based on the previous performance of the actor model. The actor then interacts with a batch of **unseen** tasks using the newly generated prompt. The quality of the prompt is subsequently evaluated on a fixed held-out set of tasks. In our experiments, we fine-tune the reflector while keeping the actor model frozen.

continuous online-learning mechanism, in which each task attempt provides an additional data point for improving behaviour after only a small number of trials.

However, existing work primarily focuses on self-correction by repeatedly attempting the same task until success. This introduces several important limitations. First, the agent acquires experience only within a single task instance, and the resulting knowledge is not transferred across tasks. Once a task has been solved, the agent must repeat the full reflection process from scratch when encountering a new task. This is inherently inefficient, since insights from one task could often generalise to similar tasks (e.g., *"in maze-solving tasks, avoid revisiting the same path"*). Under the current paradigm, however, the agent is forced to relearn such lessons for each task. Second, in many real-world settings, repeating the exact same task is not possible. For example, in a game of chess one cannot require an opponent to reproduce the same sequence of moves, and in a cooking scenario a mistake may irreversibly consume the available ingredients.

Although the ability to learn online from experience across tasks is highly desirable, it remains unclear whether self-reflection mechanisms can produce reflections that generalise well enough to improve performance on previously unseen tasks. We therefore ask the following research question: ***Can LLMs be trained to generate self-reflections that facilitate cross-task self-improvement, and if so, how?***.

To address this, we first define the **In-context Training** (ICT) task, a framework for evaluating the ability of LLMs to learn from experience for cross-task self-improvement (see Figure 1). The ICT task implements a simple self-learning loop through a dual-LLM design. The first model is the *actor*, which serves as the operational agent interacting with external environments according to a system prompt. Crucially, at each turn of the ICT task, the actor faces a new **unseen** batch of tasks and has only **one chance** to solve each task. The second model is the *reflector*, which evaluates the actor's performance and proposes an improved system prompt for the next turn. The goal of the reflector is to learn from the actor's collected experiences in order to generate a system prompt that generalises across tasks within a given family of tasks. For example, in a maze-solving task, the reflector should learn from previously encountered mazes in order to produce a system prompt that enables the actor to solve **any** unseen maze drawn from the same task family in a single attempt.

Next, we demonstrate that the ability to learn from experience is itself a learnable skill by introducing a fine-tuning pipeline to train an LLM to generate reflections that facilitate cross-task self-improvement. We train a *reflector* to analyse a given system prompt alongside experiences collected under that prompt, and to generate an improved system prompt for future unseen tasks. Our method does not rely on human-provided examples.

We evaluate our approach on ALFWorld [15] and MiniHack [9], two benchmarks comprising diverse interactive environments. Using these benchmarks within the ICT framework, we show that a reflector trained on a set of task types can generalise and improve performance on previously unseen task types using only a small number of gathered experiences. We also show that, in some cases, the model generalises beyond the benchmark on which it was trained, transferring to substantially different environments.

Finally, we introduce MetaGym, a generic Python library for constructing gym-like meta-environments from any gym-like environment [2]. We release MetaGym as a tool to support future research on self-improvement in LLM agents within the ICT framework and beyond (`https://github.com/yuvalshalev1/MetaGym`).

Our contributions can be summarised as follows:

1. A **training pipeline** for learning reflection strategies for cross-task self-improvement without reliance on human-provided labels.

2. The **In-context Training** task, a framework for evaluating online self-improvement of LLMs that generalises across task instances.

3. **MetaGym**, a Python library for constructing meta-environments from Gym-like environments.

2 Related Work

Prompt Optimisation. Previous studies have demonstrated the potential of LLMs for prompt optimisation in static tasks [20, 4, 1]. Yang et al. [20] optimise prompts for mathematical reasoning and question answering using closed-source LLMs, and Choi et al. [4] propose a bi-level framework that jointly optimises task-specific user prompts and a cross-task system prompt. In both cases, the optimiser observes only final outputs or aggregate accuracy, not **how** its prompts shaped the model's behaviour during problem solving. In our setting, however, the optimiser observes full interaction trajectories, enabling us to study its ability to learn from the agent's behaviour. Agrawal et al. [1] leverage full reasoning trajectories to optimise prompts through iterative updates to a prompt pool. However, their scoring procedure relies on evaluating multiple candidate prompts on the same batch of task instances. In contrast, our setting assumes that each task is encountered only once and learning must occur online from non-repeatable experience. More broadly, these works optimise prompts using external procedure rather than fine-tuning the optimiser itself, leaving open whether LLMs can be trained to learn better optimisation strategies.

Prompt Optimisation for LLM-Agents. Recent work has explored prompt optimisation for LLM agents in interactive environments, while transferring experience across tasks [10, 24]. Sarukkai et al. [10] maintain an external database of self-generated successful trajectories and retrieve similar examples as in-context demonstrations for the actor. This method relies on embedding raw past trajectories directly into the prompt, rather than consolidating them into a structured representation. However, prior work shows that in-context demonstrations are not always effective, even when produced by human experts [8]. Zhao et al. [24] similarly rely on an external mechanism, extracting "insights" from pairs of past experiences and maintaining them through a voting procedure, before combining them with selected raw experiences to construct task-specific prompts during inference time. These methods therefore transfer experience through external memory and retrieval, but leave open the question of whether an LLM can be trained to learn reusable reflective strategies for cross-task self-improvement.

Reflection for Self-Correction. Shinn et al. [14] introduce reflection for LLM agents as a feedback loop in which an agent analyses its behaviour after failing a task and uses this analysis to guide subsequent attempts. Since then, several works have explored how to train LLMs to reflect more effectively in interactive environments. Yao et al. [22], Ge et al. [6] and Feng et al. [5] use dual-LLM architecture that separates actor and reflector, with the former training only the reflector and the latter training both components jointly using a gradient policy. Most recently, Jiang et al. [7] proposed a meta-RL framework in which a single LLM acts as both actor and reflector. However, all of these methods focus on self-correction through repeated attempts at the same task. They, therefore, leave open whether reflection can support cross-task improvement by extracting reusable lessons from experience.

Algorithm 1 In-context Training (ICT)

Require: LLM_{refl}, LLM_{act}, E^{train}, E^{val}, batch size k, meta-turns N, initial system prompt sp_0

1: $sp^* \leftarrow sp_0, \quad score^* \leftarrow 0$
2: Sample $\{e_j^{(0)}\}_{j=1}^k \sim E^{\text{train}}$
3: $obs_0 \leftarrow \left\{\tau_j^{(0)} \sim \text{LLM}_{\text{act}}(e_j^{(0)};\ sp_0)\right\}_{j=1}^k$
4: **for** meta-turn $i = 1, \ldots, N$ **do**
5: Generate new system prompt $sp_i \sim \text{LLM}_{\text{refl}}(sp_{i-1}, obs_{i-1})$
6: $score_i \leftarrow \sum_{e \in E^{\text{val}}} R(\text{LLM}_{\text{act}}, e, sp_i)$
7: **if** $score_i > score^*$ **then**
8: $sp^* \leftarrow sp_i, \quad score^* \leftarrow score_i$
9: **end if**
10: Sample $\{e_j^{(i)}\}_{j=1}^k \sim E^{\text{train}}$
11: $obs_i \leftarrow \left\{\tau_j^{(i)} \sim \text{LLM}_{\text{act}}(e_j^{(i)};\ sp_i)\right\}_{j=1}^k$
12: **end for**
13: **return** sp^*

3 The In-context Training Task

Here, we introduce the In-context Training (ICT) task, a framework for evaluating whether large language models can learn from experience to achieve cross-task self-improvement. ICT is designed to capture a realistic setting in which the same task cannot be repeated, and the only opportunity for improvement lies in future tasks. The framework is intentionally minimalistic, aiming to measure the model's intrinsic learning ability rather than optimise performance through external machinery.

The full process is formulated in Algorithm 1 and illustrated in Figure 1. We first define the reflector model (denoted LLM_{refl}), whose learning capabilities we aim to evaluate, the actor model (denoted LLM_{act}), whose policy we aim to optimise, and a class of tasks E on which training is performed. The task class is partitioned into a training set E^{train} and a validation set E^{val}. The objective is for the tested LLM_{refl} to analyse episodes sampled from E^{train} using LLM_{act} and generate a system prompt sp that enables the actor LLM_{act} to perform effectively across all tasks in E^{val}.

Formally, the objective of the reflector LLM_{refl} is to generate a system prompt sp^* that maximises the cumulative reward obtained by the actor across the validation tasks:

$$sp^* = \arg\max_{sp} \sum_{i=1}^{|E^{val}|} R(\text{LLM}_{\text{act}}, e_i, sp) \tag{1}$$

where $R(\text{LLM}_{\text{act}}, e_i, sp)$ denotes the reward obtained by the actor LLM_{act} on task $e_i \in E^{val}$ when conditioned on the system prompt sp.

The reflector LLM_{refl} is modelled as a meta-agent interacting with a meta-environment. The meta-environment is initialised with an initial system prompt sp_0 and samples a batch of k tasks from the training set E^{train}. The meta-environment then executes a batch of episodes using sp_0 and returns a meta-observation obs_0, the full sequences of interactions between the actor LLM_{act} and the sampled environments (line 3 of Algorithm 1). Then, the meta-loop begins: at each meta-turn i, the reflector LLM_{refl} is prompted to generate a new system prompt sp_i, conditioned on the previous system prompt sp_{i-1} and the last meta-observation obs_{i-1} (line 5):

$$sp_i \sim \text{LLM}_{\text{refl}}(sp_{i-1}, obs_{i-1}) \tag{2}$$

where obs_i is the meta-observation consisting of the batch of trajectories collected under sp_i:

$$obs_i = \{\tau_j^{(i)} \sim \text{LLM}_{\text{act}}(e_j^{(i)}, sp_i)\}_{j=1}^k \tag{3}$$

and $\tau_j^{(i)} = (a_0, o_0, a_1, o_1, \ldots, a_T, o_T)$ is the trajectory of actions a_t and environment observations o_t produced by LLM_{act} on task $e_j^{(i)}$ under system prompt sp_i.

The proposed system prompt sp_i is then evaluated on the validation tasks E^{val}, and sent to the meta-environment to execute the next batch of episodes (lines 10-11).

4 Training LLMs to Learn from Experience

We now introduce our pipeline for fine-tuning LLMs to generate reflections that facilitate cross-task self-improvement.

Dataset Creation. To construct the training dataset, we run the ICT loop r times for N turns each, using a base model. At the end of each turn i, we record the tuple $(sp_i, obs_i, e_0^{(i)}, \ldots, e_{k-1}^{(i)})$, consisting of the current system prompt sp_i, the meta-observation obs_i, and the batch of tasks $e_0^{(i)}, \ldots, e_{k-1}^{(i)}$ used in that turn. Repeating this process yields a fixed dataset $\mathcal{D}$ of $r \times N$ data points in total. The motivation is to capture a diverse set of system prompts arising from different stages of the ICT process, which correspond to varying degrees of required improvement. For example, if the reflector receives a system prompt that succeeds on one task in the batch but fails on the others, it should be able to analyse the factors that contributed to the successful case and generalise them to the remaining tasks. Conversely, if the reflector receives a system prompt that succeeds on all tasks in the batch, it should generate a new system prompt that remains highly similar to the previous one, potentially reproducing it exactly.

Reward Signal. As described in Equation (2), the reflector generates a system prompt sp_i conditioned on the previous system prompt sp_{i-1} and the last batch of trajectories obs_{i-1} collected by LLM_{act} under sp_{i-1}. Its immediate task is therefore to analyse how the previous system prompt led to the actor's observed performance on the previous batch $e^{(i-1)}$, and to improve the prompt in light of the issues revealed by those trajectories. Put differently, although the reflector's goal is to improve performance on future unseen tasks, its intermediate objective is to answer the following question: *"If I had the opportunity to try the same batch of tasks again, what would I do differently to improve my performance?"*.

To encourage the model to account for observed performance issues and incorporate corresponding fixes into the new system prompt, we derive the training reward directly from the previous batch of tasks. Specifically, the reward used to rank sp_i is the average reward obtained by the actor LLM_{act} on the batch $e^{(i-1)}$ when conditioned on sp_i, that is, by replaying the previous batch under the new system prompt sp_i. Formally, the reward function is defined as:

$$\mathcal{R}(sp_i) = \frac{1}{k} \sum_{j=1}^{k} R(\text{LLM}_{\text{act}},\, e_j^{(i-1)},\, sp_i). \tag{4}$$

Notably, although the reward signal is derived from the previous batch of tasks, the reflector should be explicitly prompted to analyse those performance issues and generate a system prompt intended for **future** tasks. This is essential to discourage in-context overfitting to the most recently observed batch.

Training Procedure. We follow a standard training procedure for RL-based fine-tuning, where in each training step, we sample a batch of tuples $(sp_i, obs_i, e_0^{(i)}, \ldots, e_{k-1}^{(i)})$ from the dataset $\mathcal{D}$ and update the parameters of LLM_{refl} using GRPO [12]. The procedure is illustrated in Figure 2: for each tuple, the reflector samples a group of G candidate system prompts $\{sp^{(1)}, \ldots, sp^{(G)}\} \sim \pi_\theta(\cdot \mid sp_i, obs_i)$, and each candidate is scored using $\mathcal{R}$ as defined in Equation (4). The scores are computed dynamically during training by executing the batch of tasks $e_0^{(i)}, \ldots, e_{k-1}^{(i)}$ with LLM_{act} under each candidate system prompt.

5 Experiments

5.1 Experimental Setup

All our experiments were conducted using the Qwen2.5-7B-Instruct model [19] and the *verl* framework [13] for RL-based fine-tuning. The datasets were constructed as described in Section 4, using $r = 128$ ICT loops of $N = 4$ turns each, with Qwen2.5-7B-Instruct serving as both the actor model and the base reflector model. For each benchmark, we trained two reflectors: one with batch size $k = 3$ and another with batch size $k = 5$. The reflection prompts used in our experiment are detailed in Appendix A. Training hyperparameters and compute resources are reported in Appendix B.

MetaGym. Task execution for both the ICT loop and the fine-tuning procedure is managed by MetaGym, our library for meta-environment construction for LLM-agentic tasks. MetaGym instantiates an interactive meta-environment in which each meta-action is represented by a system prompt,

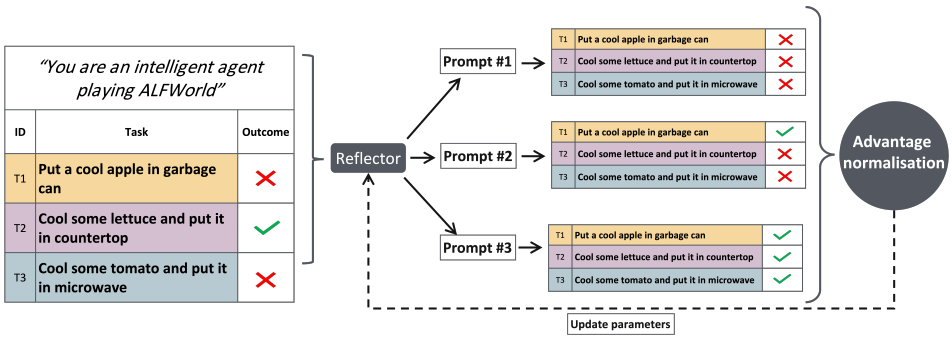


Figure 2: A single training step illustrated on an *ALFWorld-Cool and Place* example. A system prompt and its associated episode batch are sampled from the dataset and fed into the reflector, which generates multiple candidate system prompts. Each candidate is scored by replaying the **same** tasks with the frozen actor, and the resulting scores drive advantage estimation and parameter updates via GRPO.

and each returned meta-observation is a batch of rollouts generated by an actor. Internally, it spawns task instances and executes them in parallel. See Appendix C for more details and a usage example.

Task Partitioning. The objective of the proposed training approach is to train a reflector that performs effectively on the ICT task, where the goal is to generate a general system prompt that achieves strong performance across tasks within a given task type. A straightforward evaluation strategy would be to divide task instances into training and validation sets. However, this is insufficient: a reflector trained on one task type could simply memorise a high-performing prompt for that type and reproduce it at test time, thereby bypassing the in-context reasoning the model is intended to learn. To mitigate this, we partition tasks by **type** rather than by instance, yielding disjoint **meta-train** and **meta-test** sets, each comprising distinct task types. Each task type consists of a collection of similar tasks, from which the reflector is expected to extract transferable insights that generalise to other tasks within the same type. During training, the reflector is exposed exclusively to task types from the meta-train set. At evaluation time, it is required to generalise to previously unseen task types in the meta-test set, generating reflections based solely on in-context observations available at inference time.

5.1.1 Environments

To construct the meta-train and meta-test sets, we use agentic benchmark environments that could be partitioned into distinct task types. Specifically, we use ALFWorld [15] and a subset of MiniHack [9]. For each benchmark, we train a dedicated reflector using the tasks from the corresponding meta-train set. The training datasets are constructed by executing the ICT loop 32 times for each meta-train task type within the benchmark. During evaluation, we apply the ICT loop to unseen task types by sampling tasks from the meta-test set of the same benchmark.

ALFWorld. ALFWorld is a benchmark comprising six task types, all set in a partially observable, house-like textual environment. Following Jiang et al. [7], we use *Pick and Place*, *Examine in Light*, *Clean and Place*, and *Heat and Place* as the meta-train set, and *Cool and Place* and *Pick Two and Place* as the meta-test set. Further details are provided in Appendix D.

MiniHack. MiniHack is a benchmark comprising many tasks in two-dimensional grid world environments. To construct a meta-set with different task types, we use a collection of eight 5x5 grid world environments. We use *MiniHack-Room-Random-5x5-v0*, *MiniHack-Eat-v0*, *MiniHack-Wield-v0*, and *MiniHack-Wear-v0* as the meta-train set, and *MiniHack-PutOn-v0*, *MiniHack-Zap-v0*, *MiniHack-Read-v0*, and *MiniHack-Room-Dark-5x5-v0* as the meta-test set. Further details are provided in Appendix E.

5.2 Results

We evaluate the reflectors trained on ALFWorld and MiniHack on their respective held-out meta-test sets. The ALFWorld reflectors are evaluated using the *Cool and Place* and *Pick Two* task types, and the MiniHack reflectors were evaluated on the *Dark*, *Read*, *PutOn*, and *Zap* task types. Each reflector is tested using the same batch size it is trained on. For each reflector and task type, we run the ICT task 10 times for 10 turns each. This repetition is designed to control for noise introduced

Table 1: Average winning prompt success rates of our trained reflectors viz. Qwen2.5-7B-Instruct and the initial prompt (sp_0), on held-out task types for varying batch sizes k. $\pm$ values indicate standard deviation across all random seeds. Our trained reflector shows better performance on the ICT task across nearly all task types in both benchmarks.

		$k = 3$		$k = 5$	
Environment	**Initial prompt**	**Qwen2.5-7B**	**Trained (ours)**	**Qwen2.5-7B**	**Trained (ours)**
ALFWorld					
Cool and Place	13.3$\pm$4.3%	42.4$\pm$10.9%	**49.2$\pm$10.1%**	38.8$\pm$12.0%	**48.0$\pm$15.0%**
Pick Two	27.7$\pm$6.5%	34.6$\pm$5.6%	**42.5$\pm$7.2%**	28.3$\pm$6.1%	**41.2$\pm$7.6%**
MiniHack					
Dark	35.0$\pm$7.2%	48.4$\pm$5.8%	**51.9$\pm$4.0%**	42.8$\pm$9.7%	**48.1$\pm$3.5%**
Read	29.1$\pm$6.4%	34.7$\pm$4.1%	**38.4$\pm$5.4%**	32.2$\pm$5.9%	**40$\pm$5.4%**
PutOn	24.8$\pm$5.2%	26.6$\pm$9.0%	**30$\pm$4.7%**	22.8$\pm$10.0%	**34.1$\pm$2.2%**
Zap	10.4$\pm$5.8%	**18.8$\pm$5.6%**	14.1$\pm$3.5%	23.1$\pm$7.9%	**27.8$\pm$4.5%**

MiniHack - Read

=== ANALYSIS ===
In Episode 1, the agent **successfully picks up and reads** a resource. The system prompt did not specify what to do in this situation, and the agent made the correct decisions.
In Episodes 2 and 3, the agent either makes incorrect assumptions or takes actions that lead to a negative reward due to misread or misinterpreted situations.
The system prompt needs to clearly guide the agent on what to do in all situations [...]

=== IMPROVED PROMPT ===
You are an intelligent agent playing [...]
Continue exploring until you either find a resource or item, or until you have explored all possible directions. [...]
If you find a resource or item, pick it up or **read it**, depending on the situation.

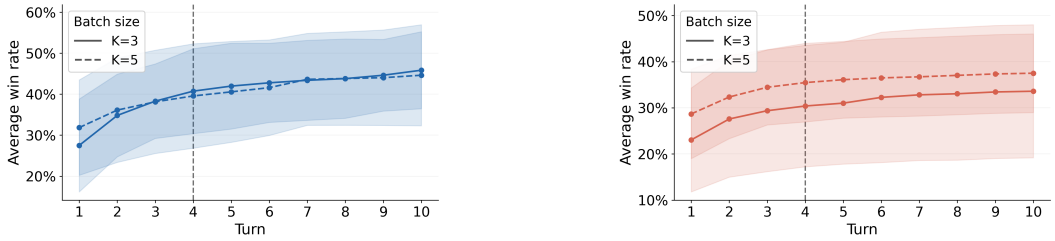
ALFWorld - Cool and Place

=== ANALYSIS ===
In all three episodes, the agent failed to correctly identify the location of a cool potato or a plate. [...]
For improvement, **the prompt should more strongly guide the agent to consider locations like fridges** and side tables when an object needs to be cooled. The prompt should ensure the agent knows to go to these locations first.

=== IMPROVED PROMPT ===
You are an intelligent agent playing [...]
If you see an object you need to cool, **consider locations such as the fridge** and side tables, and attempt to go there first to pick up the object and cool it more quickly. [...]
Remember to safely handle the cooled object and place it in the required location after cooling.

Figure 3: Examples of our reflectors' output on held-out task types. **(Top)** The reflector correctly deduces that the goal of *MiniHack-Read* environments is to find the item and read it, and generates a system prompt that guides the agent to do so. **(Bottom)** The reflector observes that the actor is struggling to understand how to cool the objects in the *ALFWorld Cool and Place* environments, and generates a system prompt that guides the agent to look for the fridge. There were no tasks requiring cooling or reading in the training data.

by model temperature and to evaluate robustness across different in-context training tasks. Each run is initialised using a different random seed to obtain different task batches, while the validation sets are kept fixed across all runs. For each run, we select the best-performing prompt, and the average success rates across the winning prompts are summarised in Table 1, along with the performance of the untrained Qwen2.5-7-Instruct model and the average result achieved using the initial prompt. For calculating the average result of the initial prompt, we use the initial outcomes from all runs across all four reflectors ($n = 40$). Figure 3 shows representative examples of reflector outputs on the *MiniHack-Read* and *ALFWorld Cool and Place* tasks. Below we discuss our findings.



(a) ALFWorld

(b) MiniHack

Figure 4: Average winning prompt success rate up to each turn, aggregated across all tasks for each reflector. Shaded regions indicate standard deviation across all random seeds. The dashed vertical line marks turn 4, the end of the training horizon. Performance continues to improve beyond the number of steps seen during training.

LLMs Can Be Trained to Achieve Cross-Task Self-Improvement. Table 1 reveals that across nearly all evaluated tasks and environments, our trained reflector consistently outperforms the untrained Qwen2.5-7B-Instruct baseline, demonstrating that the ability to learn from experience is itself a learnable capability. Reflectors trained on one set of task types successfully transfer their improved reflection ability to unseen tasks types – providing direct evidence that *learning from experience* is itself a learnable meta-skill. The one exception is the *Zap* task in MiniHack for the k=3 condition, which is the hardest task and may require larger batches to meaningfully learn from. Notably, our results indicate that untrained Qwen2.5-7B-Instruct was also able to improve the initial system prompt when conditioned on sampled episodes of the actor, suggesting that the ability to extract reusable lessons from experience already exists in the model to some extent.

More Meta-Turns Lead to Better Performance. Figure 4 presents the average best win rate achieved up to each turn, aggregated across all tasks for each experimental group. Performance improves consistently with additional meta-turns across all four groups and, crucially, continues to increase beyond turn 4, the boundary of our training data. This is an important finding: during training, our model was exposed to prompts created only by the first 4 turns of the ICT loop, yet at test time, it was able to recursively improve its own generated prompts further beyond this horizon. In the ALFWorld experiments, 55% (k=3) and 45% (k=5) of the best-performing prompts were achieved after turn 4, while in the MiniHack experiments, 32.5% (k=3) and 37.5% (k=5) emerged after turn 4. Notably, 25% of the best ALFWorld k=3 prompts were not attained until turn 10, suggesting that further meta-turns may yield additional improvements.

Effect of Batch Size. The batch size plays two important roles that can influence the performance of the reflector. First, it determines the number of episodes observed by the reflector in each turn, thereby affecting the diversity of experiences from which it can learn. Notably, when a task is difficult, a small batch size may contain no successful episodes, making it more challenging for the reflector to extract useful lessons in the absence of positive examples. Second, batch size affects the fine-tuning process of the reflector itself: during training, if none of the generated candidate prompts is successful on any of the replayed tasks, the GRPO algorithm cannot compute advantages because all rewards are zero. A larger batch size increases the likelihood of obtaining a positive reward signal, particularly when tasks are difficult and rewards are sparse.

Increasing the reflection batch size from k=3 to k=5 shows different effects across the benchmarks (see Table 1 and Figure 4). In ALFWorld, performance is similar across batch sizes, suggesting that the quality of individual reflections remains largely unchanged. Taking into account the total number of tasks experienced, k=3 shows a clear advantage, as adding more episodes appears to be redundant while costing more solving attempts. In MiniHack, the performance of the k=5 reflector was, in most cases, better than that of the k=3 reflector, suggesting that using larger batches can be more effective for fine-tuning the reflector.

The untrained Qwen2.5-7B-Instruct performs worse under the k=5 condition across almost all tasks. Notably, in the *Pick Two* task, the prompts it discovers show performance remaining close to the initial prompt. One possible explanation is that processing five episodes simultaneously presents a workload that is too high for Qwen2.5-7B to handle effectively. One interesting exception is the *Zap* task in MiniHack, where Qwen2.5-7B-Instruct performs better in the k=5 condition, which we attribute to the difficulty of the task.

Table 2: Cross-benchmark evaluation: average winning prompt success rate for MiniHack reflector evaluated on ALFWorld. $\pm$ values indicate standard deviation across all random seeds. Our trained $k = 5$ reflector was able to transfer its learning capabilities to substantially different task types.

		$k = 3$		$k = 5$	
Environment	**sp$_0$**	**Qwen2.5-7B**	**Trained (ours)**	**Qwen2.5-7B**	**Trained (ours)**
Pick and Place	78.1$\pm$3.4%	**89.1$\pm$4.7%**	86.6$\pm$4.6%	**87.8$\pm$2.6%**	83.4$\pm$4.2%
Examine in the Light	45.2$\pm$8.3%	**66.2$\pm$9.2%**	64.6$\pm$12.0%	61.5$\pm$8.4%	**73.1$\pm$8.6%**
Clean and Place	34.4$\pm$5.7%	**56.7$\pm$8.6%**	39.6$\pm$6.6%	50.7$\pm$11.1%	**57.8$\pm$5.8%**
Heat and Place	16.6$\pm$8.1%	**25.0$\pm$8.8%**	19.4$\pm$7.1%	21.9$\pm$8.0%	**40.6$\pm$11.9%**
Cool and Place	13.3$\pm$4.1%	**42.4$\pm$10.9%**	27.6$\pm$6.6%	**38.8$\pm$12.0%**	34.0$\pm$6.0%
Pick Two	28.2$\pm$5.2%	**34.6$\pm$5.6%**	25.4$\pm$5.1%	28.3$\pm$6.1%	**35.0$\pm$5.0%**

Cross-Benchmark Generalisation. To test whether the meta-skill learned by our method generalises beyond the training benchmark, we evaluate reflectors trained only on MiniHack within ALFWorld. Results are shown in Table 2, alongside the initial prompt and Qwen2.5-7B-Instruct baselines. The reflector trained on MiniHack with a batch size of k=5 outperforms the untrained Qwen2.5-7B-Instruct baseline on 4 out of 6 tasks, while achieving comparable performance on the remaining two. Notably, on both the *Examine in the Light* and *Heat and Place* tasks, it outperforms Qwen2.5-7B-Instruct even under the k=3 condition, which we previously observed in Table 1 to be easier within ALFWorld. This is a significant finding, as it demonstrates transferability between fundamentally different benchmarks. By contrast, the k=3 reflector does not achieve improved performance and, in some cases, performs worse than the untrained model. We attribute this, as discussed earlier, to more difficult training at smaller batch sizes, where the reward signal is sparser.

5.3 Limitations

In our experiments, we trained only the reflector model in order to isolate the contribution of prompt evolution to actor performance. However, maintaining separate actor and reflector models is computationally expensive in practice, as it requires holding two models in memory simultaneously. Training a single model to serve both roles could yield further performance gains. Additionally, during the ICT loop we used a fixed held-out task set to identify the best-performing prompt at the end of each loop. In real-world settings, such a repeatable evaluation set may not be available, necessitating alternative mechanisms for determining when to stop adaptation. One such alternative is lifelong learning – continuously updating the system prompt without a fixed stopping criterion – though this raises open questions about the long-term stability of the generated prompts, which we leave for future work.

Our next steps are to test the robustness of our approach further: collecting more samples to reduce variance, experimenting with different model sizes and baselines, and conducting additional ablation and qualitative analyses of our proposed training pipeline.

6 Conclusion

In this work, we investigated whether LLMs can learn from their own experience in-context in the realistic setting of online learning. We introduced the In-context Training (ICT) task as an framework for evaluating cross-task self-improvement, in which an LLMs repeatedly analyses batches of unseen episodes and generates system prompts intended to improve performance across a task family. To train models for this setting, we proposed an RL-based fine-tuning approach that derives reward signals from replayed input tasks. Across held-out task types, our trained reflectors consistently outperform an untrained baseline, showing that leaning from experience is itself a learnable capability.

We believe this has important implications for language agents. Our work provides both an evaluation framework and a training methodology for cross-task self-learning, a capability likely to matter for general-purpose agents operating in open-ended environments. We hope it shifts attention from reflection within a single task instance to reflection across tasks. Because our approach requires no annotated examples and applies directly to agentic benchmarks, it is readily extensible. To support further research, we release MetaGym (see Appendix C), a Python library for constructing meta-environments for LLM agentic tasks.

Acknowledgments and Disclosure of Funding

YS is supported by the Reuben Cambridge Postgraduate Scholarship (Cambridge Trust / Reuben Foundation). ZD receives support from the DARPA program SciFy.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Yumin Choi, Jinheon Baek, and Sung Ju Hwang. System prompt optimization with meta-learning. *arXiv preprint arXiv:2505.09666*, 2025.

[5] Xueyang Feng, Bo Lan, Quanyu Dai, Lei Wang, Jiakai Tang, Xu Chen, Zhenhua Dong, and Ji-Rong Wen. Improving retrospective language agents via joint policy gradient optimization. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 112–141, 2025.

[6] Yubin Ge, Salvatore Romeo, Jason Cai, Monica Sunkara, and Yi Zhang. Samule: Self-learning agents enhanced by multi-level reflection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 16602–16621, 2025.

[7] Yulun Jiang, Liangze Jiang, Damien Teney, Michael Moor, and Maria Brbic. Meta-rl induces exploration in language agents. In *International Conference on Learning Representations*, 2026.

[8] Anian Ruoss, Fabio Pardo, Harris Chan, Bonnie Li, Volodymyr Mnih, and Tim Genewein. Lmact: A benchmark for in-context imitation learning with long multimodal demonstrations. *arXiv preprint arXiv:2412.01441*, 2024.

[9] Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL `https://openreview.net/forum?id=skFwlyefkWJ`.

[10] Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-generated in-context examples improve llm agents for sequential decision-making tasks. *arXiv preprint arXiv:2505.00234*, 2025.

[11] Danny P Sawyer, Nan Rosemary Ke, Hubert Soyer, Martin Engelcke, David P Reichert, Drew A Hudson, John Reid, Alexander Lerchner, Danilo Jimenez Rezende, Timothy P Lillicrap, et al. Can foundation models actively gather information in interactive environments to test hypotheses? *arXiv preprint arXiv:2412.06438*, 2024.

[12] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[13] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

[14] 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.

[15] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. URL `https://arxiv.org/abs/2010.03768`.

[16] David Silver and Richard S Sutton. Welcome to the era of experience. *Google AI*, 1, 2025.

[17] Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2023.

[18] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[19] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[20] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

[21] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[22] Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, et al. Retroformer: Retrospective large language agents with policy gradient optimization. *arXiv preprint arXiv:2308.02151*, 2023.

[23] Lance Ying, Katherine M Collins, Prafull Sharma, Cedric Colas, Kaiya Ivy Zhao, Adrian Weller, Zenna Tavares, Phillip Isola, Samuel J Gershman, Jacob D Andreas, et al. Assessing adaptive world models in machines with novel games. *arXiv preprint arXiv:2507.12821*, 2025.

[24] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

A Reflector Prompt

The reflector prompt is shared across both environments. The reflector receives the following system prompt:

Reflector system prompt

```
You are an expert at analysing game-playing trajectories and improving agent
instructions.
You will be given one or more episodes in which an agent attempts to complete
a task guided by a system prompt.  Your goal is to learn from these episodes
to improve the agent's performance on future tasks.
Your job is to:
1.  Identify what went well and what went wrong in the agent's behaviour.
2.  Diagnose how the current system prompt contributed to those outcomes.
3.  Write an improved system prompt that addresses the identified weaknesses
to improve the agent's performance in future episodes.
Respond in EXACTLY this format -- no additional text outside these two
sections:
ANALYSIS:
[Describe what succeeded, what failed, how the system prompt contributed to
those outcomes, and what specific changes the improved prompt should make.]
IMPROVED PROMPT:
[The full improved system prompt, written to be used directly without
modification.]
```

The user message sent to the reflector follows this template:

Reflector user prompt template

```
## System Prompt Used
{previous_prompt}
## Episode Trajectories
=== Episode 1 ===
Success:  Yes/No

-- Step 1 --
Observation:  {observation}
Action:  {action}

-- Step 2 --
Observation:  {observation}
Action:  {action}

...

Step limit reached.  (only if unsuccessful)
Total reward:  {reward}


=== Episode 2 ===
Success:  Yes/No

-- Step 1 --
Observation:  {observation}
Action:  {action}

-- Step 2 --
Observation:  {observation}
Action:  {action}

...
```

```
Step limit reached.  (only if unsuccessful)
Total reward:  {reward}


Analyse the trajectories and provide an improved system prompt.
```

B Compute and Training Details

All experiments were conducted on a cluster node equipped with four NVIDIA GH200 Grace Hopper Superchips, each with 96 GB of GPU memory. Each reflector was trained for approximately 20 hours. Each ICT evaluation experiment, consisting of 10 full ICT loops, took approximately 2 node hours. In total, training and evaluation across all experiments required approximately 168 node hours. The training code and ICT scripts are available at `https://github.com/yuvalshalev1/In-Context-Training`.

Table 3 lists the hyperparameters used to train the four reflectors. All runs use Qwen2.5-7B-Instruct as the base model, GRPO [12] as the training algorithm, and the *verl* framework [13] for RL-based fine-tuning. Parameters that differ across runs reflect the longer in-context histories produced by different batch sizes k and different observation lengths in different benchmarks.

For checkpoint selection, we constructed an additional static dataset using the base model and the dataset creation procedure described in Section 4, with tasks drawn from the train split of the **meta-test set**. These are the tasks that we provide to the model as input during testing to generate reflections (rather than the tasks that we keep held out and use to report performance). Each checkpoint was evaluated on this dataset using a single reflection step, and the best-performing checkpoints were retained for further consideration. This protocol was designed to assess whether the trained models generalise to unseen task types, rather than overfit to the task types observed in the meta-train set. Among the retained candidates, the final checkpoint was selected based on its performance on the ICT task.

Table 3: Hyperparameters for reflector training across all four settings.

Hyperparameter	**ALFWorld** k=3	**ALFWorld** k=5	**MiniHack** k=3	**MiniHack** k=5
Base model	Qwen2.5-7B-Instruct			
Algorithm	GRPO			
Learning rate	3×10^{-6}			
Train batch size	16			
PPO mini-batch size	16			
PPO epochs	2			
Rollout samples (n)	8			
KL loss coefficient	0.001			
Max prompt length	8192	8192	9216	10240
Max response length	2048	2048	2048	1024
Rollout max model len	10240	10240	11264	11264
Checkpoint used	160	120	70	60

C MetaGym

MetaGym is a Python library designed to support prompt optimisation and self-improvement research for LLM agents. It natively implements the ICT task from the perspective of the reflector as a meta-agent. Each meta-action is a system prompt, and each returned meta-observation is a batch of rollouts generated by the actor conditioned on that prompt. Internally, the meta-environment spawns a batch of task instances and executes them in parallel using a chosen actor, providing native infrastructure for running the ICT loop described in section 3. MetaGym wraps a Gym-compatible environment [2] inside a Gym-like meta-environment and handles the parallel execution and trajectory collection required by the ICT framework. In our experiments, all task execution across ALFWorld and MiniHack was managed through MetaGym. The library ships with built-in support for both

benchmarks and provides instructions for integrating new environments, with the aim of supporting future research on prompt optimisation and self-improvement in LLM agents. MetaGym is released publicly alongside this work at `https://github.com/yuvalshalev1/MetaGym`.

The following snippet illustrates the MetaGym API using ALFWorld:

MetaGym usage example

```
from metagym import Actor
from metagym.envs.alfworld import ALFWorldMetaEnv
from metagym.reflector import Reflector

# Actor auto-launches a vLLM server for the model and shuts it down on exit
actor = Actor("Qwen/Qwen2.5-7B-Instruct")

# Initialise the meta-environment
env = ALFWorldMetaEnv(num_envs=3, actor=actor, split="train", seed=42)
trajectories, info = env.reset()

# Reflector:  analyses trajectories and rewrites the system prompt
reflector = Reflector("Qwen/Qwen2.5-7B-Instruct", base_url=actor.base_url)

current_prompt = "You are a helpful agent..."
for step in range(4):
    trajectories, reward, done, info = env.step(current_prompt)
    print(f"Step {step} win rate:  {reward:.0%}")
    reflection = reflector.reflect(trajectories, current_prompt)
    print(reflection.analysis)
    current_prompt = reflection.improved_prompt

# Inspect a trajectory
trajectories[0].pretty_print()
```

D ALFWorld Details

For the ICT loop, we use the built-in training and evaluation splits provided by the ALFWorld benchmark. The reflector receives task batches sampled from the training split, while the full validation split is used for evaluation. Each episode is limited to 25 steps.

D.1 Actor prompt

The actor is implemented as a ReAct-style agent [21]. The initial system prompt sp_0 is shared across all ALFWorld experiments. It is as follows:

ALFWorld actor system prompt

```
You are an intelligent agent playing ALFWorld, a text-based game where you
control a robot in a household environment.
Your goal is to complete tasks by interacting with objects in the
environment.
You must use the ReACT (Reasoning and Acting) approach:
1.  THINK about what you observe and what you should do next
2.  Take an ACTION from the available actions
3.  Observe the result and repeat
Format your responses EXACTLY as follows:
Thought:  [your reasoning about the current situation and what to do]
Action:  [exact action from the available actions list]
```

Each turn, the actor receives the following user message, appended to the conversation history of all previous steps in the episode:

ALFWorld actor user prompt template

```
{observation}
Available actions:
{available_actions}
What is your next thought and action?
```

D.2 Trajectory example

ALFWorld trajectory excerpt (pick_two_obj_and_place)

```
Observation 0:
You are in the middle of a room. Looking quickly around you, you see a
cabinet 4, a cabinet 3, a cabinet 2, a cabinet 1, a coffeemachine 1, a
countertop 1, a drawer 8, a drawer 7, a drawer 6, a drawer 5, a drawer 4,
a drawer 3, a drawer 2, a drawer 1, a fridge 1, a garbagecan 1, a microwave 1,
a sidetable 1, a sinkbasin 1, a stoveburner 4, a stoveburner 3, a stoveburner
2, a stoveburner 1, and a toaster 1.
Your task is to: put two spoon in sidetable.

Available actions: examine cabinet 1, go to cabinet 2, go to cabinet 3,
go to cabinet 4, go to coffeemachine 1, go to countertop 1, go to drawer 1,
go to drawer 2, go to drawer 3, go to drawer 4, go to drawer 5, go to drawer
6, go to drawer 7, go to drawer 8, go to fridge 1, go to garbagecan 1, go to
microwave 1, go to sidetable 1, go to sinkbasin 1, go to stoveburner 1, go
to stoveburner 2, go to stoveburner 3, go to stoveburner 4, go to toaster 1,
help, inventory, look, open cabinet 1
Thought 1: The task is to put two spoons in the sidetable. Spoons are
likely to be found in cabinets or drawers. I will start by checking the
cabinets.
Action 1: go to cabinet 1

Observation 1:
You arrive at cabinet 1. The cabinet 1 is closed.

Available actions: close cabinet 1, examine cabinet 1, go to cabinet 2,
go to cabinet 3, go to cabinet 4, go to coffeemachine 1, go to countertop 1,
go to drawer 1, go to drawer 2, go to drawer 3, go to drawer 4, go to drawer
5, go to drawer 6, go to drawer 7, go to drawer 8, go to fridge 1, go to
garbagecan 1, go to microwave 1, go to sidetable 1, go to sinkbasin 1, go to
stoveburner 1, go to stoveburner 2, go to stoveburner 3, go to stoveburner
4, go to toaster 1, help, inventory, look, take mug 1 from cabinet 1, take
peppershaker 1 from cabinet 1
Thought 2: Since the cabinet is closed, I need to open it to see if there
are any spoons inside.
Action 2: open cabinet 1

Observation 2:
You open the cabinet 1. The cabinet 1 is open. In it, you see a mug 1, and
a peppershaker 1.

Available actions: examine cabinet 2, go to cabinet 1, go to cabinet 3,
go to cabinet 4, go to coffeemachine 1, go to countertop 1, go to drawer 1,
go to drawer 2, go to drawer 3, go to drawer 4, go to drawer 5, go to drawer
6, go to drawer 7, go to drawer 8, go to fridge 1, go to garbagecan 1, go to
microwave 1, go to sidetable 1, go to sinkbasin 1, go to stoveburner 1, go
to stoveburner 2, go to stoveburner 3, go to stoveburner 4, go to toaster 1,
help, inventory, look, open cabinet 2
Thought 3: The cabinet did not contain any spoons. I should check the other
cabinets and drawers for spoons.
Action 3: go to cabinet 2
```

E MiniHack Details

MiniHack environments are procedurally generated, so task instances are produced by sampling random seeds. The reflector receives task batches sampled from a designated range of training seeds, while a fixed set of 32 seeds is held out and used consistently as the validation set for the ICT loop. Each episode is limited to 25 steps, with the exception of the k=5 training run, where this limit was reduced to 18 steps to keep episode trajectories within the model's context length.

MiniHack natively exposes a large number of actions, many of which are irrelevant to the tasks in our benchmark. To reduce ambiguity and keep the context concise, we always provide the actor with a curated list of available actions that are suitable for our task set, rather than the full action space.

E.1 Actor prompt

The actor is implemented as a ReAct-style agent [21]. The initial system prompt sp_0 is shared across all MiniHack experiments. It is as follows:

MiniHack actor system prompt

```
You are an intelligent agent playing MiniHack, a text-based dungeon
exploration game.  You are placed in an ASCII-based grid world in which your
character is represented by the ‘‘@’’ symbol.  Explore the grid world and find
your goal.
You must use the ReACT (Reasoning and Acting) approach:
1.  THINK about what you observe and what you should do next
2.  Take an ACTION from the available actions
3.  Observe the result and repeat
Format your responses EXACTLY as follows:
Thought:  [your reasoning about the current situation and what to do]
Action:  [exact action from the available actions list]
```

Each turn, the actor receives the following user message, appended to the conversation history of all previous steps in the episode:

MiniHack actor user prompt template

```
Task:  play {task}
{observation}
Available actions:
{available_actions}
What is your next thought and action?
```

E.2 Trajectory example

MiniHack trajectory excerpt (MiniHack-PutOn-v0)

```
Observation 0:
Hello Agent, welcome to NetHack!  You are a neutral human Caveman.

    ."...
    .....
    .....
    ..@..
    .....

Available actions:  step n, step s, step e, step w, step ne, step nw, step
se, step sw, run n, run s, run e, run w, run ne, run nw, run se, run sw, read,
eat, puton, zap, wield, wear, pickup
Thought 1:  The game has just started.  I will start by moving north to
explore.
```

```
Action 1:  step n

Observation 1:
    ."...
    .....
    ..@..
    ..<..
    .....

Available actions:  step n, step s, step e, step w, step ne, step nw, step
se, step sw, run n, run s, run e, run w, run ne, run nw, run se, run sw, read,
eat, puton, zap, wield, wear, pickup
Thought 2:  I can see a "<" symbol.  I will move east to investigate further.
Action 2:  step e

Observation 2:
    ."...
    .....
    ...@.
    ..<..
    .....

Available actions:  step n, step s, step e, step w, step ne, step nw, step
se, step sw, run n, run s, run e, run w, run ne, run nw, run se, run sw, read,
eat, puton, zap, wield, wear, pickup
Thought 3:  Moving south to continue exploring.
Action 3:  step s
```