# TRAINING LANGUAGE AGENTS TO LEARN FROM EXPERIENCE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Language agents can adapt from experience in interactive environments, but current reflection-based methods primarily self-correct by retrying the same 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 actor trajectories and generates system prompts intended to improve the actor's performance on future unseen tasks. We also propose an RL-based pipeline for training reflectors directly from experience, without human-provided examples. Across ALFWorld and MiniHack, our trained reflectors outperform matched untrained baselines on all six held-out task types, showing that the ability to learn from experience can itself be learned. These gains extend across two model families and to a different actor, with initial evidence of cross-benchmark transfer. Finally, we introduce MetaGym, a Python library for constructing meta-environments for self-improving language agents.

## 1 INTRODUCTION

Large language models (LLMs) increasingly act as decision-making agents in interactive environments, producing tokens that are interpreted as actions (Wang et al., 2024; Sumers et al., 2023; Yao et al., 2022). When faced with an unfamiliar environment, a general-purpose agent should learn and adapt its behaviour to improve its policy (Silver & Sutton, 2025; Sawyer et al., 2024; Ying et al., 2025; Shinn et al., 2023). Humans do this naturally: when playing chess for the first time, a person may begin with an initial strategy and refine it after each game.

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 (Yao et al., 2022; Brown et al., 2020). 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 explore self-improvement in language agents through autonomous prompt modification and self-reflection (Shinn et al., 2023; Yao et al., 2023; Feng et al., 2025; Ge et al., 2025; Jiang et al., 2026). An LLM analyses the agent's actions after an unsuccessful attempt, then generates a reflection to guide the next attempt. However, these methods primarily improve performance by retrying the same task instance. The resulting lessons are not evaluated for transfer to future tasks, forcing the agent to relearn similar strategies for each instance. Moreover, exact retries are impossible in many settings: 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



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

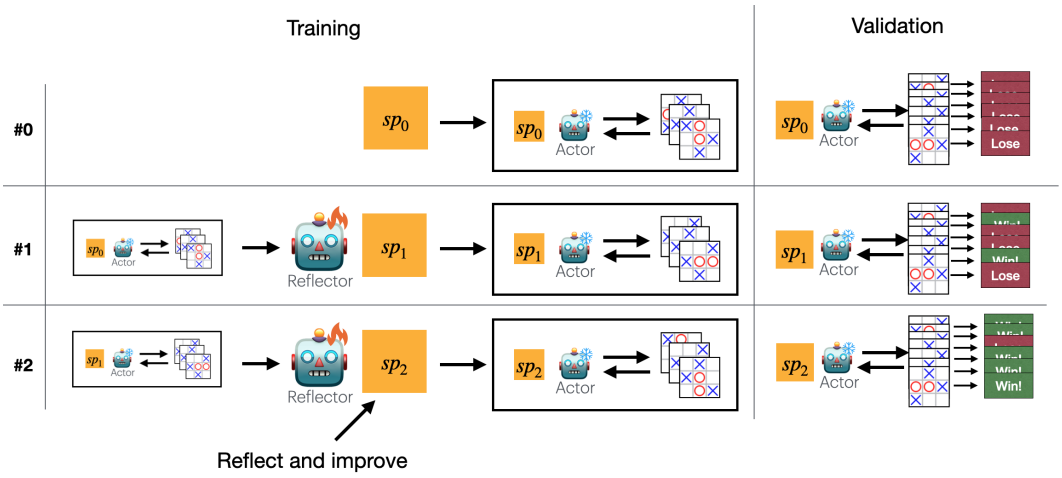



Figure 1: The In-context Training (ICT) task. At each meta-turn, the reflector uses the actor's previous trajectories to revise its system prompt, after which the actor attempts a new batch of unseen tasks. Each prompt is subsequently evaluated on a fixed held-out set of tasks. In our experiments, we fine-tune the reflector while keeping the actor model frozen.

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 question, we introduce the **In-context Training** (ICT) task, illustrated in Figure 1. ICT uses a dual-LLM design: (1) an *actor* interacts with the environment according to a system prompt, while (2) a *reflector* analyses the actor's experience and revises that prompt. The reflector must therefore extract lessons that improve performance on future instances rather than enable another attempt at the same instance. For example, in a maze-solving task, experience from previously encountered mazes may reveal strategies that help the actor solve new, unseen mazes.

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 (Shridhar et al., 2021) and MiniHack (Samvelyan et al., 2021), 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 (Brockman et al., 2016). We release MetaGym as a tool to support future research on self-improvement in LLM agents within the ICT framework and beyond (`https://anonymous.4open.science/r/MetaGym-575F`).

Our contributions can be summarised as follows:

1. The **In-context Training** task, a framework for evaluating cross-task self-improvement of LLMs from non-repeatable experience.
2. An **RL-based training pipeline** for learning cross-task reflection strategies without human-provided examples.
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 (Yang et al., 2023; Choi et al., 2025; Agrawal et al., 2025). Yang et al. (2023) optimise prompts for mathematical reasoning and question answering using closed-source LLMs, and Choi et al. (2025) 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. (2025) 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 methods optimise prompts through an external search procedure rather than training the optimiser itself, leaving open whether LLMs can learn better prompt-update strategies.

**Prompt Optimisation for LLM Agents.** Recent work has explored prompt optimisation for LLM agents in interactive environments, while transferring experience across tasks (Sarukkai et al., 2025; Zhao et al., 2024). Sarukkai et al. (2025) 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 (Ruoss et al., 2024). Zhao et al. (2024) 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. (2023) 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. (2023) and Ge et al. (2025) use dual-LLM architectures that train only the reflector, whereas Feng et al. (2025) trains both components jointly using policy gradients. Most recently, Jiang et al. (2026) 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.

## 3 The In-context Training Task

Here, we introduce the In-context Training (ICT) task, a framework for evaluating whether LLMs can use experience to achieve cross-task self-improvement. ICT captures a realistic setting in which task instances cannot be repeated, and improvement must transfer to future tasks. The framework is intentionally minimalistic, aiming to measure the model's intrinsic self-improvement ability rather than optimise performance through external machinery.

The full process is defined in Algorithm 1 and illustrated in Figure 1. We define the reflector model $\text{LLM}_{\text{refl}}$, whose improvement capabilities we aim to evaluate, the actor model $\text{LLM}_{\text{act}}$, whose policy we aim to optimise, and a task family $E$ over which adaptation is performed. $E$ is partitioned into an adaptation set $E^{\text{train}}$, from which the reflector observes actor trajectories, and a fixed validation set $E^{\text{val}}$, which scores the generated prompts. The objective of the reflector $\text{LLM}_{\text{refl}}$ is to analyse episodes sampled from $E^{\text{train}}$ using the actor $\text{LLM}_{\text{act}}$, and generate a system prompt $sp$ that improves the actor's performance across all tasks in $E^{\text{val}}$.

**Algorithm 1** In-context Training (ICT)

**Require:** $\text{LLM}_{\text{refl}}$, $\text{LLM}_{\text{act}}$, $E^{\text{train}}$, $E^{\text{val}}$, 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** $sp^*$

Formally, the objective of the reflector $\text{LLM}_{\text{refl}}$ 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 $\text{LLM}_{\text{act}}$ on task $e_i \in E^{val}$ when conditioned on the system prompt $sp$.

The reflector $\text{LLM}_{\text{refl}}$ 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 $E^{\text{train}}$. 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 $\text{LLM}_{\text{act}}$ and the sampled environments (line 3 of Algorithm 1). Then, the meta-loop begins: at each meta-turn $i$, the reflector $\text{LLM}_{\text{refl}}$ 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 $\text{LLM}_{\text{act}}$ on task $e_j^{(i)}$ under system prompt $sp_i$.

The proposed system prompt $sp_i$ is then evaluated on the validation tasks $E^{\text{val}}$, 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

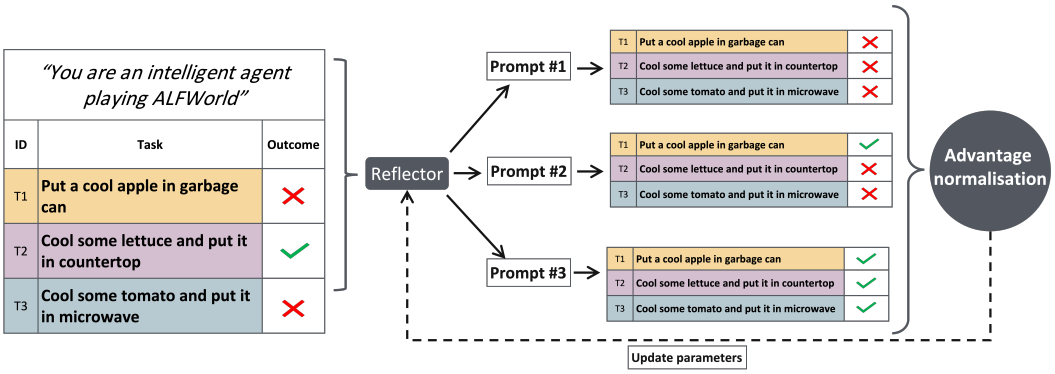



Figure 2: One reflector-training step on an *ALFWorld-Cool and Place* example. A system prompt and its episode batch are sampled from the dataset and fed into the reflector, which generates multiple candidate system prompts. Each candidate is scored by replaying the **same** tasks with the frozen actor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

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 $\text{LLM}_{\text{act}}$ 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 $\text{LLM}_{\text{act}}$ 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:

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

Although the reward signal is derived by replaying the previous batch of tasks, we explicitly prompt the reflector to generate a system prompt intended for **future** tasks, discouraging overfitting to the observed instances. Replay is used only to construct the offline training signal; during ICT evaluation, each adaptation task is encountered once.

**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 $\text{LLM}_{\text{refl}}$ using GRPO (Shao et al., 2024). 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 $\text{LLM}_{\text{act}}$ under each candidate system prompt.

# 5 EXPERIMENTS

## 5.1 EXPERIMENTAL SETUP

Unless otherwise specified, all experiments were conducted using the Qwen2.5-7B-Instruct model (Yang et al., 2024) and the *verl* framework (Sheng et al., 2024) for RL-based fine-tuning. We construct the datasets 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. 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-agent tasks. MetaGym instantiates an interactive meta-environment in which each meta-action is represented by a system prompt, 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 can be partitioned into distinct task types. Specifically, we use ALFWorld (Shridhar et al., 2021) and a subset of MiniHack (Samvelyan et al., 2021). For each benchmark, we train a dedicated reflector on the tasks in the corresponding meta-train set, and then evaluate it on tasks drawn 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. (2026), 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. For dataset creation, we execute the ICT loop 32 times for each of the four meta-train task types. 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. For dataset creation, we execute the ICT loop 32 times for each of the four meta-train task types. 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. For the ALFWorld reflector, we use *Cool and Place* and *Pick Two* task types, while for the MiniHack reflector, we use *Read*, *PutOn*, *Zap* and *Dark* task types. For each reflector and task type, we run the ICT task 30 times for 10 turns, using batch size $k = 5$ for both benchmarks. These repetitions allow us to account for variation due to model sampling and task selection, and to evaluate robustness across different ICT runs. Each run is initialised using a different random

Table 1: Average success rates of the selected prompts generated by our trained reflectors, the corresponding untrained baseline reflectors, and the initial prompt on held-out task types. ± values indicate standard deviation across random seeds. Bold indicates the highest result for each task type. Averaged over tasks, the trained reflector improves over its baseline by **+9.9%** on ALFWorld ($p < 0.001$) and **+3.9%** on MiniHack ($p < 0.001$) for Qwen2.5-7B-Instruct, and by **+17.4%** ($p < 0.001$) and **+13.5%** ($p < 0.001$) for Llama3.1-8B-Instruct (paired Wilcoxon signed-rank test).

| | | Qwen2.5-7B reflectors | | Llama3.1-8B reflectors | |
|---|---|---|---|---|---|
| **Environment** | **$sp_0$** | **Baseline** | **Trained (ours)** | **Baseline** | **Trained (ours)** |
| *ALFWorld* | | | | | |
| Cool and Place | 15.4±4.7% | 36.7±11.3% | **47.1±21.5%** | 26.4±9.3% | **42.7±6.9%** |
| Pick Two | 28.3±6.5% | 28.9±6.2% | **38.3±8.9%** | 24.2±7.0% | **42.6±8.8%** |
| *MiniHack* | | | | | |
| Read | 30.7±6.3% | 35.4±4.7% | **40.4±5.4%** | 19.7±10.6% | **36.9±5.1%** |
| PutOn | 17.1±5.7% | 26.2±5.4% | **30.6±4.9%** | 15.3±8.4% | **27.6±7.6%** |
| Zap | 7.0±4.0% | 18.3±8.1% | **23.8±6.2%** | 8.3±7.4% | **21.6±6.1%** |
| Dark | 42.9±6.3% | 51.0±5.8% | **52.0±4.8%** | 39.5±11.8% | **50.7±7.2%** |

**ALFWorld - Cool and Place**

**=== ANALYSIS ===**
In Episodes 1 and 2, the agent failed to complete the assigned tasks [...] The agent did not cool the items properly **despite seeing a fridge** [...] In Episode 3, the agent succeeded in cooling and placing a bowl in the shelf [...] The system prompt did not guide the agent effectively to cool the items.

**=== IMPROVED PROMPT ===**
You are an intelligent agent playing [...]
Remember to cool the item and place it in the designated receptacle [...] go to the item to take it from its initial location, then **go to the fridge** or sink to cool the item [...]

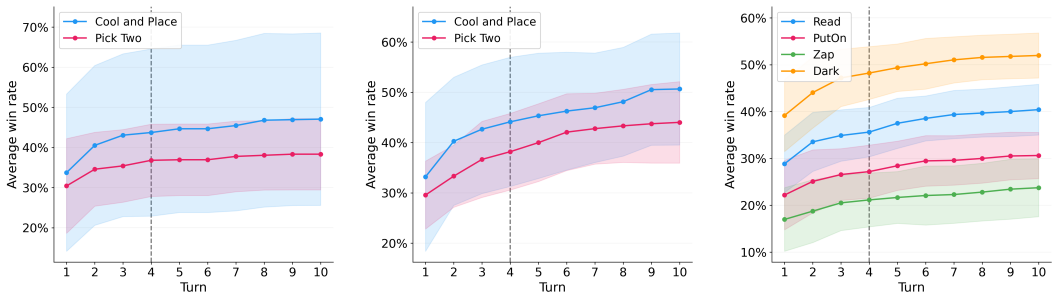
**MiniHack - Read**

**=== ANALYSIS ===**
[...] In Episode 3, the agent found a scroll but **did not read it**, which means it may not have prioritized reading scrolls effectively [...] In Episode 5, the agent **found a scroll and read it** [...] However, this episode is the only successful one [...]

**=== IMPROVED PROMPT ===**
You are an intelligent agent playing [...]
If you find a scroll, **read it**. [...]

Figure 3: Examples of outputs from trained reflectors on held-out task types. **(Top)** The reflector observes that the actor struggles to determine how to cool objects in *ALFWorld Cool and Place* and generates a system prompt that instructs it to look for the fridge. **(Bottom)** The reflector infers that success in *MiniHack-Read* requires reading a scroll and generates a system prompt that instructs the actor to do so.

seed to produce different task batches, while the validation sets remain fixed across runs. For each run, we select the best-performing prompt using the fixed validation set. Table 1 reports the mean success rate of these selected prompts across runs, together with the performance of the untrained baseline reflector and the initial prompt. For the initial prompt, we aggregate outcomes across the trained-reflector and baseline conditions ($n = 60$). We also train reflectors based on Llama 3.1-8B-Instruct (Grattafiori et al., 2024) using the same dataset and frozen Qwen2.5-7B-Instruct actor, testing whether the method transfers to a second reflector family. These results are also reported in Table 1. Examples of reflector outputs are shown in Figure 3. Below we discuss our findings.




(a) ALFWorld ($k = 5$) (b) ALFWorld ($k = 3$) (c) MiniHack ($k = 5$)

Figure 4: Average success rate of the best prompt identified up to each meta-turn, aggregated across task types for each reflector. Shaded regions indicate standard deviation across random seeds. The dashed vertical line marks meta-turn 4, the end of the training horizon. Reflectors continue to identify higher-performing prompts beyond the horizon observed during training.

**LLMs Can Be Trained to Achieve Cross-Task Self-Improvement.** Table 1 shows that our trained reflectors consistently outperform their untrained baselines across all evaluated task types, both for Qwen2.5 and Llama3.1. Reflectors trained on one set of task types therefore improve actor performance on previously unseen task types.

To quantify the performance gain from training, we calculate the difference in performance between each pair of ICT runs within each benchmark (*trained* − *baseline*), matched by random seed. This compares performance on the same batches of in-context training examples. We assess statistical significance using the paired Wilcoxon signed-rank test. Overall, our Qwen2.5 reflectors improve over their baselines by an average of **9.9%** on ALFWorld ($p < 0.001$) and **3.9%** on MiniHack ($p < 0.001$). The corresponding improvements for the Llama3.1 reflectors are **17.4%** ($p < 0.001$) and **13.5%** ($p < 0.001$), respectively.

To further investigate the generalisability of our reflectors, we test whether a reflector trained to improve one frozen actor can transfer its reflection capabilities to a different actor. Specifically, we repeat the evaluation using Qwen3-4B-Instruct-2507 (Team, 2025) as the actor. The trained ALFWorld reflector improves over the baseline reflector by an average of **19.6%** ($p < 0.001$), while the trained MiniHack reflector achieves an average improvement of **4.2%** ($p < 0.001$). Detailed results are presented in Table 4 in Appendix F.

**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 a useful learning signal, as all rewards are equal. A larger batch size increases the likelihood of obtaining a positive reward signal, particularly when tasks are difficult and rewards are sparse.

To study this effect, we additionally train reflectors with a smaller batch size ($k = 3$) for each benchmark and repeat the ICT evaluation. The results are presented in Table 5 in Appendix G. Reducing the reflection batch size has different effects across the two benchmarks. In ALFWorld, the trained reflector continues to outperform the baseline, achieving an average improvement of **14.9%** ($p < 0.001$). Given the smaller number of tasks encountered per turn, this suggests that the additional episodes provided by $k = 5$ are not necessary for effective reflection in this benchmark. In MiniHack, however, the trained $k = 3$ reflector performs comparably to the baseline reflector, with an average difference of +0.2% ($p = 0.99$). These results suggest that larger reflection batches may be necessary to obtain a useful fine-tuning signal on this benchmark.

**Reflectors Continue to Improve Beyond the Training Horizon.** Figure 4 presents the average best win rate achieved up to each turn, aggregated across tasks for different experimental settings. In all three settings, reflectors continue to improve actor performance beyond turn 4, the end of the

Table 2: Cross-benchmark evaluation of MiniHack-trained reflector on ALFWorld. We report the average success rates of the selected prompts. ± values indicate standard deviation across random seeds, and bold indicates the highest result for each task type. Averaged over tasks, the trained reflector improves over the untrained Qwen2.5-7B-Instruct reflector by **+3.7%** ($p < 0.001$; paired Wilcoxon signed-rank test), demonstrating that the learned reflection strategy transfers across benchmarks.

| Environment | $\mathbf{sp}_0$ | Qwen2.5-7B | Trained (ours) |
|---|---|---|---|
| Pick and Place | 77.6±2.9% | **88.6±3.1%** | 84.9±4.3% |
| Examine in the Light | 45.1±10.3% | 66.4±9.6% | **71.3±9.5%** |
| Clean and Place | 31.5±5.3% | 50.4±13.6% | **54.3±6.6%** |
| Heat and Place | 15.9±8.6% | 21.9±7.2% | **39.2±12.5%** |
| Cool and Place | 15.6±4.8% | **36.7±11.3%** | 35.7±8.8% |
| Pick Two | 28.3±6.2% | 28.9±6.2% | **29.4±4.4%** |

horizon used to construct the training data. In the ALFWorld experiment, 28.3% of runs achieve their best performance after turn 4, increasing to 56.7% for $k = 3$. In the MiniHack experiment, 47.5% of runs achieve their best performance after turn 4. Although the model was trained only on prompts generated during the first four turns of the ICT loop, it can continue to improve actor performance over a longer test-time horizon.

**Cross-Benchmark Generalisation.** To test whether the reflection capabilities learned by our method generalise beyond the training benchmark, we evaluate a reflector trained exclusively on MiniHack within ALFWorld. Detailed results are presented in Table 2, together with the initial-prompt and untrained Qwen2.5-7B-Instruct baselines. The MiniHack-trained reflector outperforms the untrained baseline by an average of 3.7% ($p < 0.001$), achieving higher performance on four out of six task types. This result shows that the learned reflection strategy can transfer between substantially different benchmarks.

### 5.3 Limitations

In our experiments, we train only the reflector model to isolate the contribution of prompt evolution to actor performance. However, maintaining separate actor and reflector models is computationally expensive, as it requires holding two models in memory simultaneously. Training a single model to serve both roles could reduce these resource requirements.

During the ICT loop, we used a fixed held-out task set to identify the best-performing prompt. In real-world settings, such a repeatable evaluation set may be unavailable, requiring alternative stopping criteria. One option is lifelong learning, which continuously updates the system prompt without a fixed stopping criterion. However, this raises open questions about the long-term stability of the generated prompts, which we leave for future work.

## 6 Conclusion

In this work, we investigate whether LLMs can learn in context from their own experience during online adaptation. We introduce the In-context Training (ICT) task, a framework for evaluating cross-task self-improvement in which an LLM analyses batches of unseen episodes and generates system prompts to improve performance across a task family. We also propose an RL-based fine-tuning approach that derives rewards from replayed task instances. Across held-out task types, our trained reflectors consistently outperform their untrained baselines, showing that the ability to learn from experience can itself be improved through training.

Our work provides an evaluation framework and training methodology for cross-task self-learning, a capability important for general-purpose agents operating in open-ended environments. We hope it shifts attention from reflection within individual task instances to reflection across tasks. Our approach requires no annotated examples and applies directly to agentic benchmarks. To support further research, we release MetaGym (see Appendix C), a Python library for constructing meta-environments for LLM-agent tasks.

### AI USE STATEMENT

In this work, we used generative AI tools to assist in implementing our proposed method in code, as well as in writing and debugging supporting code (e.g., experiment scripts and plotting utilities). We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results. The rest of the required disclosure tasks (formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs) are not applicable to this work, as the paper contains no formal proofs. Additionally, we used AI tools to check grammar and make minor wording refinements to text written by the authors. No content, claims, or sections were generated by AI. All AI-assisted code was reviewed, tested, and verified for correctness by the authors, and every AI-suggested text edit was manually reviewed. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

### REPRODUCIBILITY STATEMENT

We have made several efforts to ensure that our results are reproducible. The ICT task is formally defined in Section 3 (Algorithm 1), and the training pipeline is described in Section 4. The experimental setup is specified in Section 5.1, and further implementation details including training hyperparameters and the compute resources are provided in Appendix A and Appendix B. All experiments use open-weights models and publicly available benchmarks. Finally, we release source code for reflector training and ICT evaluation (`https://anonymous.4open.science/r/ICT-3449`), together with MetaGym, our library for running meta-environments (`https://anonymous.4open.science/r/MetaGym-575F`; see Appendix C).

## REFERENCES

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.

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

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.

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

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)*, pp. 112–141, 2025.

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*, pp. 16602–16621, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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.

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.

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.

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.

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.

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.

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.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

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

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

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

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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.

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.

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.

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.

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.

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.

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

We conduct all experiments on a cluster node equipped with four NVIDIA GH200 Grace Hopper Superchips, each with 96 GB of GPU memory. Each reflector is trained for approximately 20 hours. Each ICT evaluation experiment, consisting of 30 full ICT loops, took approximately 6 node hours. The training code and ICT scripts are available at `https://anonymous.4open.science/r/ICT-3449`.

Table 3 lists the hyperparameters used to train the six reflectors: four based on Qwen2.5-7B-Instruct and two based on Llama3.1-8B-Instruct. All runs use GRPO (Shao et al., 2024) as the training algorithm, and the *verl* framework (Sheng et al., 2024) 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 validation, we constructed an additional static dataset using Qwen2.5-7B-Instruct and the dataset creation procedure described in Section 4, with tasks drawn from the meta-test set task types. These tasks were excluded from the meta-test set and were never used in the ICT evaluation process. Each checkpoint is evaluated on this dataset using a single reflection step. 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.

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

| | **Qwen2.5-7B-Instruct** | | | | **Llama3.1-8B-Instruct** | |
|---|---|---|---|---|---|---|
| **Hyperparameter** | **ALFWorld** $k$=3 | **ALFWorld** $k$=5 | **MiniHack** $k$=3 | **MiniHack** $k$=5 | **ALFWorld** $k$=5 | **MiniHack** $k$=5 |
| Algorithm | GRPO | | | | | |
| Learning rate | $3 \times 10^{-6}$ | | | | | |
| 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 | 8192 | 10240 |
| Max response length | 2048 | 2048 | 2048 | 1024 | 2048 | 1024 |
| Rollout max model len | 10240 | 10240 | 11264 | 11264 | 10240 | 11264 |
| Checkpoint used | 160 | 120 | 70 | 60 | 80 | 40 |

## 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 (Brockman et al., 2016) 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://anonymous.4open.science/r/MetaGym-575F`.

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 (Yao et al., 2022). 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

For the ICT loop, we hold out a fixed set of 32 task instances, which is used consistently as the validation set across all runs. The reflector receives task batches sampled from the remaining training instances, which are guaranteed to be disjoint from the validation set. 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 (Yao et al., 2022). 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
```

## F ACTOR TRANSFERABILITY RESULTS

Table 4 reports results for reflectors trained with Qwen2.5-7B-Instruct as the actor and evaluated with Qwen3-4B-Instruct-2507 as the actor. In ALFWorld, the trained reflector improves over the baseline by an average of **19.6%** ($p < 0.001$), while in MiniHack it achieves an average improvement of **4.2%** ($p < 0.001$).

## G RESULTS FOR SMALLER BATCH SIZE

Table 5 reports results for reflectors trained and evaluated with batch size $k = 3$. In ALFWorld, the trained reflector improves over the baseline by an average of **14.9%** ($p < 0.001$), while in MiniHack the reflector performs comparably to the baseline reflector, with an average difference of +0.2% ($p = 0.99$).

Table 4: Average winning prompt success rates with Qwen3 as the actor model, alongside the initial prompt ($sp_0$). $\pm$ values indicate standard deviation across all random seeds. Bold indicates the highest result for each task type. Averaged over tasks, the trained reflector improves over Qwen2.5-7B-Instruct by **+19.6%** on ALFWorld ($p < 0.001$) and **+4.2%** on MiniHack ($p < 0.001$) (paired Wilcoxon signed-rank test).

| Environment | $\mathbf{sp}_0$ | Qwen2.5-7B | Trained (ours) |
|---|---|---|---|
| *ALFWorld* | | | |
| Cool and Place | 15.7±4.9% | 26.5±12.3% | **40.7±21.3%** |
| Pick Two | 29.0±6.8% | 24.7±11.0% | **49.9±8.7%** |
| *MiniHack* | | | |
| Read | 39.7±7.3% | 44.6±10.7% | **51.8±5.3%** |
| PutOn | 17.4±4.8% | 24.9±6.8% | **33.9±6.2%** |
| Zap | **34.2±7.2%** | 21.7±8.8% | 25.2±5.3% |
| Dark | 50.1±6.4% | **52.3±7.2%** | 49.6±4.8% |

Table 5: Average success rates of our trained reflectors, the untrained Qwen2.5-7B-Instruct reflector, and the initial prompt ($sp_0$), on held-out task types (batch size $k = 3$). $\pm$ values indicate standard deviation across all random seeds. Bold indicates the highest result for each task type. Averaged over tasks, the trained reflector improves over the untrained Qwen2.5-7B-Instruct reflector by **+14.9%** on ALFWorld ($p < 0.001$) and **+0.2%** on MiniHack ($p = 0.99$) (paired Wilcoxon signed-rank test).

| Environment | Initial prompt | Qwen2.5-7B | Trained (ours) |
|---|---|---|---|
| *ALFWorld* | | | |
| Cool and Place | 14.3±3.7% | 35.9±10.5% | **50.7±11.1%** |
| Pick Two | 29.3±6.3% | 29.0±7.6% | **44.0±8.1%** |
| *MiniHack* | | | |
| Read | 31.5±5.7% | 34.0±5.9% | **35.4±5.6%** |
| PutOn | 16.6±5.6% | 26.0±7.1% | **29.3±6.4%** |
| Zap | 6.7±4.6% | **15.3±7.1%** | 11.4±3.1% |
| Dark | 43.2±7.2% | **51.5±5.1%** | 51.4±6.1% |