GRACE: A LANGUAGE MODEL FRAMEWORK FOR EXPLAINABLE INVERSE REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Inverse Reinforcement Learning (IRL) aims to recover Reward Models from expert demonstrations, but traditional methods yield "black-box" models that are difficult to interpret and debug. In this work, we introduce GRACE (**G**enerating **R**ewards **A**s **C**od**E**), a method for using code Large Language Models (LLMs) within an evolutionary search to reverse-engineer an interpretable, code-based reward function directly from expert trajectories. The resulting reward function is executable code that can be inspected and verified. We empirically demonstrate that GRACE can efficiently learn highly accurate rewards in the multi-task setups as defined by two benchmarks, BabyAI and AndroidWorld. Further, we demonstrate that the resulting reward leads to strong policies compared to both competitive Imitation Learning and online RL approaches with groundtruth rewards. Finally, we show that GRACE is able to build complex reward APIs in multi-task setups.

1 INTRODUCTION

The performance of modern Reinforcement Learning (RL) agents is determined by, among other factors, the quality of their reward function. Traditionally, reward functions are defined manually as part of the problem specification. In many real-world settings, however, environments are readily available while reward functions are absent and must be specified. Manually designing rewards is often impractical, error-prone, and does not scale, particularly in contemporary multi-task RL scenarios (Wilson et al., 2007; Teh et al., 2017; Parisotto et al., 2016).

A natural alternative is to automate reward specification by learning a reward model from data. The dominant paradigm here is Inverse Reinforcement Learning (IRL), which attempts to infer a reward model from observations of expert behavior (Ng & Russell, 2000; Christiano et al., 2017; Ziebart et al., 2008). In the era of Deep RL, approaches such as GAIL (Ho & Ermon, 2016) represent rewards with deep neural networks. While effective, these reward functions are typically opaque black boxes, making them difficult to interpret or verify (Molnar, 2020). Moreover, IRL methods often require substantial amounts of data and often lead to inaccurate rewards (Sapora et al., 2024).

An alternative representation that has recently gained traction is using code to express reward models (Venuto et al., 2024a; Ma et al., 2023). These approaches leverage code-generating Large Language Models (LLMs) and human-provided task descriptions or goal states to generate reward programs (Venuto et al., 2024a). Subsequently, the generated rewards are verified (Venuto et al., 2024a) or improved using the performance of a trained policy as feedback (Ma et al., 2023). However, this prior work has not investigated whether it is possible to recover a reward function purely from human demonstrations in an IRL-style setting, without utilizing any explicit task description or domain-specific design assumptions.

In this work, we address the question of how to efficiently infer rewards-as-code from expert demonstrations using Large Language Models (LLMs). We propose an optimization procedure inspired by evolutionary search (Goldberg, 1989; Eiben & Smith, 2003; Salimans et al., 2017; Romera-Paredes et al., 2024a; Novikov et al., 2025b), in which code LLMs iteratively introspect over demonstrations to generate and refine programs that serve as reward models. This perspective effectively revisits the IRL paradigm in the modern context of program synthesis with LLMs.

Our contributions are threefold. We first demonstrate that code LLMs conditioned on expert demonstrations can produce highly accurate reward models. These rewards generalize well to held-out demonstrations and are well-shaped, providing informative intermediate signals rather than merely verifying final success criteria. We further show that the approach is sample-efficient: accurate rewards are obtained from relatively few demonstrations, in contrast to IRL methods based on neural networks that typically require large amounts of training data. More importantly, directly using demonstrations means no domain knowledge or human-in-the-loop guidance is manually specified during reward generation.

Second, we show that the learned rewards enable training of strong policies. We perform our evaluations in two domains: the procedurally generated navigation environment *BabyAI* (Chevalier-Boisvert et al., 2018) and the real-world device control environment *AndroidWorld* (Rawles et al., 2024) demonstrate that GRACE outperforms established IRL approaches such as GAIL (Ho & Ermon, 2016) as well as online RL with ground-truth rewards (Schulman et al., 2017). This highlights both the efficiency of GRACE in learning rewards and its promise for building capable agents across diverse domains.

Finally, by representing rewards as code, GRACE inherits additional advantages. The resulting rewards are interpretable and verifiable by humans, and, when inferred across multiple tasks, naturally form reusable reward APIs that capture common structure and enable efficient multi-task generalization. Our analysis shows that as the evolutionary search progresses, GRACE shifts from creating new functions to heavily reusing effective, high-level modules it has already discovered, demonstrating the emergence of a modular code library.

2 Related Works

LLMs for Rewards A common way to provide verification/reward signals in an automated fashion is to utilize Foundation Models. LLM-based feedback has been used directly by Zheng et al. (2023) to score a solution. Additionally, an LLM can be used to a critique examples (Zankner et al., 2024). Comparing multiple outputs in a relative manner has been also explored by Wang et al. (2023). Note that such approaches use LLM in a zero shot fashion with additional prompting and potential additional examples. Hence, they can utilize only a small number of demonstrations at best. In addition to zero shot LLM application, it is also common to train reward models, either from human feedback (Ouyang et al., 2022) or from AI feedback (Klissarov et al., 2023; 2024). Note that such approaches require training a reward model that isn't interpretable and often times require a larger number of examples.

Code as Reward As LLMs have emerged with powerful program synthesis capabilities (Chen et al., 2021; Austin et al., 2021; Li et al., 2023; Fried et al., 2022; Nijkamp et al., 2022) research has turned towards generating environments for training agents Zala et al. (2024); Faldor et al. (2025) for various domains and complexities. When it comes to rewards in particular, code-based verifiers use a language model to generate executable Python code based on a potentially private interface such as the environment's full state. Because early language models struggled to reliably generate syntactically correct code, the first code-based verifiers (Yu et al., 2023; Venuto et al., 2024b) implemented iterative re-prompting and fault-tolerance strategies. More recent approaches focus on progressively improving a syntactically correct yet suboptimal reward function, particularly by encouraging exploration (Romera-Paredes et al., 2024b; Novikov et al., 2025a). Other approaches such as Zhou et al. (2023); Dainese et al. (2024) use search in conjunction with self-reflection (Madaan et al., 2023) to provide feedback.

Inverse Reinforcement Learning (IRL) Early approaches infer a reward function by requiring the expert policy to outperform all alternatives (Ng & Russell, 2000). While related to our formulation, our representation (code) and our optimization strategy (evolutionary search) are fundamentally different. Subsequent works have focused on directly learning policies without explicit reward recovery (Abbeel & Ng, 2004), while incorporating entropy regularization (Ziebart et al., 2008) or leveraging convex formulations (Ratliff et al., 2006). In contrast, GRACE benefits from implicit regularization through its symbolic reward representation, though evolutionary search provides no optimization guarantees. More recently, Imitation Learning (IL) has achieved considerable practical success (Ross et al., 2011), often by training a discriminator to distinguish expert from non-expert trajectories (Ho & Ermon, 2016; Swamy et al., 2021). While such discriminators define implicit rewards, our approach instead operates with explicit reward representations.

3 METHOD

3.1 BACKGROUND

Reinforcement Learning We consider a finite-horizon Markov Decision Process (MDP) (Puterman, 2014) parameterized by $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, r \rangle$ where $\mathcal{S}$, $\mathcal{A}$ are the state and action spaces, $T : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the transition operator, and R is a reward function. The agent's behavior is described by the policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$. Starting from a set of initial states $\mathcal{S}_0 \subset \mathcal{S}$, the agent takes the action $a \sim \pi(s)$ at s, receives a reward $r(s)$ and transitions into state $s' \sim T(s, a)$.

The performance of the agent is measured with expected cumulative per-timestep rewards, referred to as return:

$$J(\pi, r) = \mathbb{E}_{\tau \sim \pi, T}[\sum_{t=1}^{H} r(s_t)] \tag{1}$$

where τ are trajectory unrolls of horizon H of the policy π in $\mathcal{M}$. An optimal agent can be learned by maximizing Equation (1) via gradient descent with respect to the policy, also known as policy gradient (Sutton et al., 1999; Schulman et al., 2017).

Inverse Reinforcement Learning If the reward r is not specified, it can be learned from demonstrations of an expert policy π_E. In particular, the classical IRL objective learns a reward whose optimal return is attained by the expert (Ng & Russell, 2000; Syed & Schapire, 2007):

$$\min_{\pi} \max_{R} J(\pi_E, r) - J(\pi, r) \tag{2}$$

More recent Imitation Learning (IL) approaches learn a discriminator that distinguishes between expert and non-expert demonstrations (Ho & Ermon, 2016; Swamy et al., 2021). The likelihood of the agent's data under the trained discriminator can be implicitly thought of as a reward. These approaches utilize gradient based methods to optimize their objectives.

Evolutionary search As an alternative for cases where the objective is not readily differentiable, gradient-free methods can be employed. One such method is evolutionary search, which maintains a set of candidate solutions (called a population) and applies variation operators to improve it. These operators include mutation, where a hypothesis is partially modified, and recombination, where two hypotheses are combined to produce a new one. Each variation is evaluated using a fitness function, which measures the quality of a given hypothesis. Starting with an initial population, evolutionary search repeatedly applies these variation operators, replacing hypotheses with higher-fitness alternatives.

In this work, we focus on inferring reward functions, represented as Python code, from a set of demonstrations. While this setup is related to IRL, representing rewards as code prevents us from applying gradient-based methods commonly used in IRL. For this reason, we adopt evolutionary search as our optimization method.

3.2 GRACE

We propose GRACE - **G**enerating **R**ewards **A**s **C**od**E**, an interpretable IRL framework that generates a reward function as executable Python code. Initially, an LLM analyzes expert and random trajectories to optionally identify goal states (Phase 1) and generates a preliminary set of reward programs. The step of goal identification is optional and can be skipped in favor of directly querying the LLM for a reward function which best matches the expert trajectories. This initial set is then iteratively improved through evolutionary search, where the LLM mutates the code based on misclassified examples to maximize a fitness function (Phase 2). Finally, an RL agent is trained using the refined reward, and the new trajectories it generates are used to further expand the dataset and further improve the reward function (Phase 3). The overall process is illustrated in Figure 1 and detailed below and in Algorithm 1

Phase 1: Initialization The initial reward code generation by GRACE is based on a set of demonstration trajectories $\mathcal{D}^+$ and a set of random trajectories $\mathcal{D}^-$. The former is generated using an expert policy or human demonstrations depending on the concrete setup, while the latter is produced by a random policy. Note that with a slight abuse of notation we will use $\mathcal{D}$ to denote interchangebly a set of trajectories as well the set of all states from these trajectories.

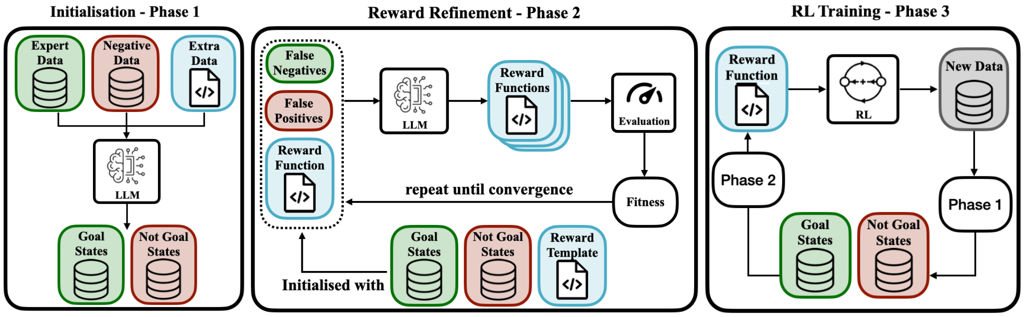

Figure 1: Overview of the GRACE framework. **(a)** The expert, negative and extra data (if any) is used to identify goal states. By default, all expert states are classified as goal states and all negative states as non-goal states **(b)** The goal and non-goal states are used to generate reward functions through an evolutionary procedure. The rewards are iteratively refined by feeding the examples misclassified by the reward. **(c)** An agent is trained with online RL using the converged reward; the data it sees during the training is classified by the LLM into $\mathcal{D}^+, \mathcal{D}^-$ and used to further improve the reward.

The language model is prompted with a random subset of $\mathcal{D}^+$ and, optionally, extra information available about the environment (e.g. its Python code or tool signature), to produce two artifacts:

Initial rewards: The LLM generates an initial set $\mathcal{R}^{\text{init}}$ of reward functions. Each function $r \in \mathcal{R}^{\text{init}}$ is represented as Python code:

```
def reward(state: string) -> float:
    <LLM produced code>
```

(Optional) Goal states: The LLM analyzes the states from expert demonstrations to identify the subset of goal states $\mathcal{S}_g \subseteq \mathcal{D}^+$ that solve the task - these are positive samples. All remaining non-goal states $\mathcal{S}_{ng} = \{\mathcal{D}^+ \setminus \mathcal{S}_g\} \cup \mathcal{D}^-$ are initially treated as negative samples.
designed to assign high values to goal states $\mathcal{S}_g$ and low values to non goal ones $\mathcal{S}_{ng}$. This set of rewards is treated as the population in the subsequent evolution phase.

Phase 2: Reward Refinement through Evolutionary Search GRACE uses Evolutionary Search to obtain rewards that best explain the current set of goal and non goal states. This is achieved by *mutating* the current reward population $\mathcal{R}$ using a code LLM and retaining rewards with high *fitness*.

The *fitness* f of a reward function r measures how well this function assigns large values to goal and small values to non-goal states, akin to what would be expected from a meaningful reward:

$$f(r) = \mathbb{E}_{s\sim\mathcal{S}_g}[r(s)] - \mathbb{E}_{s\sim\mathcal{S}_{ng}}[r(s)] \tag{3}$$

~~In practice, to normalize the fitness computation, we bound the reward signal. Any reward value greater than or equal to a predefined $r(s) \geq \tau$ is treated as 1, and any value below is treated as 0 for the purpose of this calculation.~~

The *mutation* operator m of a reward, that is used to improve the current reward population, is based on an LLM that is prompted to introspect the reward code and address failures. To do so it is provided with several inputs pertaining to the source code of the reward (if available), misclassified states, and additional debugging information:

$$m(r) = \text{LLM}(\text{source}(r), \text{info}, \text{prompt}) \tag{4}$$

In more detail, $\text{source}(r)$ is the Python code for the reward. The $\text{info} = (s_g, r(s_g), s_e, \text{debug}(r, s_g))$ is intended to focus the model on failures by honing onto states misclassified by the reward. It consists of a sequence of misclassified states $s \in S$, their reward value $r(s)$, as well as a debugging info $\text{debug}(r, s)$ produced by printing intermediate values during the execution of r on the misclassified state s. The composition of this feedback is intentionally varied; each prompt contains a different number of examples, presented as either individual states or full trajectories. To help the model discriminate between true and false positives, prompts containing a false positive are augmented with an expert state $s_e \sim D^+$.

We repeatedly apply the above mutation operation to modify the reward population $\mathcal{R}$ to improve its fitness. In more detail, we repeatedly sample a reward $r \in \mathcal{R}$ with probability $\frac{\exp F(r)}{\sum_{r' \in \mathcal{R}_P^i} \exp(F(r'))}$. Subsequently, we apply the mutation and keep the new reward function only if it has a higher fitness than other already created rewards. After K mutations, we return the reward function with highest fitness $r^* = \arg\max_{r \in \mathcal{R}} \{f(r)\}$. This phase is presented as function EvoSearch in Algorithm 1.

Phase 3: Training Trajectory Expansion via Reinforcement Learning The optimal reward r^* above is obtained by inspecting existing demonstrations. In order to further improve the reward, we ought to collect further demonstrations by training a policy π_{r^*} using the current optimal reward r^*; and use this policy to collect additional data $\mathcal{D}_{r^*}$.

In more detail, we employ PPO (Schulman et al., 2017) to train a policy in the environment of interest. As this process can be expensive, we use a predefined environment interaction budget N instead of training to convergence. After obtaining these additional trajectories, we use the same process as described in Sec. (3.2, Phase 1) to identify goal $\mathcal{S}_{g^*}$ and non-goal states $\mathcal{S}_{ng^*}$. The new trajectories are likely to contain new edge cases and examples of reward hacking, if any. These are used to further refine the reward population as described in the preceeding Sec. (3.2, Phase 2.1). The process terminates when the RL agent achieves a desired level of performance. This phase is presented as function DataExpand in Algorithm 1.

The final algorithm, presented in Algorithm 1, consists of repeatedly performing Evolutionary Search over reward population $\mathcal{R}$ followed by data expansion using RL-trained policy. Each iteration is called a generation.

Additional reward shaping When the reward function offline performance on $\mathcal{D}$ doesn't translate to good online RL performance, we assume that the reward signal is poorly shaped, and additional refinement is required. In these cases, the LLM's info in Eq. 4 is augmented beyond misclassified states to include failed trajectory examples from $\mathcal{D}_{r^*}$. To achieve this, we instruct the LLM to reshape the reward function, using expert trajectories as a reference, so that it provides a signal that increases monotonically towards the goal.

Discussion The above algorithm iterates between policy optimization and reward optimization. The objective for the latter is the fitness function from Eq. 3. If one flips the reward on non-goal states of positive demonstrations or goal states in learned policy demonstrations, it is straightforward to show that GRACE optimizes the canonical IRL objective using Evolutionary Search.

Proposition 1. *Suppose $m(s) = 1$ iff $s \in \mathcal{S}_g$, else $m(s) = -1$, then GRACE optimizes, $\min_\pi \max_r J(\pi_E, m \circ r) - J(\pi, -m \circ r)$, which is a variation of Eq. (2).*

The proof can be found in Appendix A.1.

4 Experiments

We empirically evaluate GRACE with respect to its ability to generate rewards that lead to effective policy learning. Specifically, we aim to address the following questions:

Accuracy and Generalization: Can GRACE recover correct rewards, and how much supervision is required to do so?

Policy Learning Performance: How does GRACE compare to other IRL methods or to online RL trained with ground-truth rewards?

Qualitative Properties: How well-shaped are the rewards produced by GRACE?

Interpretability and Multi-Task Efficacy: Does GRACE produce reward APIs that can be shared across tasks?

Algorithm 1 GRACE: Generating Rewards As CodE

Inputs:
$\mathcal{D}^+$: expert trajectories
$\mathcal{D}^-$: random trajectories
Parameters:
P: reward population size
K: mutation steps
M number of generations
N: RL budget

procedure GRACE($\mathcal{D}^+$, $\mathcal{D}^-$)
// Phase 1: Initialization.
$\mathcal{S}_g = \{s \in D^+ \mid \text{LLM}(s, \text{goal\_prompt})\}$
$\mathcal{S}_{ng} = \mathcal{D}^+ \cup \mathcal{D}^- / \mathcal{S}_g$
$\mathcal{R} = \{\text{LLM}(S_n, S_{ng}, \text{reward\_prompt})\}$

// Reward Refinement.
for $i = 1 \dots M$ **do**
$\mathcal{R} \leftarrow \text{EvoSearch}(\mathcal{R}, \mathcal{S}_g, \mathcal{S}_{ng})$
$\mathcal{D}, \mathcal{S}_g^*, \mathcal{S}_{ng}^* \leftarrow \text{DataExpandRL}(\mathcal{R})$
$\mathcal{S}_g = \mathcal{S}_g^* \cup \mathcal{S}_g, \mathcal{S}_{ng} = \mathcal{S}_{ng}^* \cup \mathcal{S}_{ng}$
end for
return $r^* = \arg\max_{r \in \mathcal{R}} f(r)$
end procedure

// Phase 2: Refinement via Evolution.
function EvoSearch($\mathcal{R}, \mathcal{S}_g, \mathcal{S}_{ng}$)
for $k = 1 \dots K$ **do**
Sample $r \sim \exp(f(r)), r \in \mathcal{R}$
$r' \leftarrow m(r)$ // See Eq. 4
if $f(r') > \min_{r \in \mathcal{R}} f(r)$ **then**
$r'' = \arg\min_{r \in \mathcal{R}} f(r)$
$\mathcal{R} = \mathcal{R} / \{r''\} \cup \{r'\}$
end if
end for
return $\mathcal{R}$
end function

// Phase 3: Trajectory expansion via RL.
function DataExpandRL($\mathcal{R}$)
$r^* \leftarrow \arg\max_{r \in \mathcal{R}} f(r)$
Train π_{r^*} with PPO under budget N
Collect new trajectories $\mathcal{D}_{r^*}$
$\mathcal{S}_g = \{s \in \mathcal{D}_{r^*} \mid \text{LLM}(s, \text{goal\_prompt})\}$
$S_{ng} = \mathcal{D}_{r^*} / \mathcal{S}_g$
return $\mathcal{S}_g, \mathcal{S}_{ng}$
end function

4.1 Experimental Setup

To evaluate GRACE, we conduct experiments in two distinct domains: the procedurally generated maze environment *BabyAI* (Chevalier-Boisvert et al., 2018), which tests reasoning and generalization, and the Android-based UI simulator *AndroidWorld* (Rawles et al., 2024), which tests control in high-dimensional action spaces.

BabyAI Our *BabyAI* evaluation suite comprises 20 levels, including 3 custom levels designed to test zero-shot reasoning on tasks not present in public datasets, thereby mitigating concerns of data contamination. Expert demonstrations are generated using the `BabyAI-Bot` (Farama Foundation et al., 2025), which algorithmically solves BabyAI levels optimally. We extend the bot to support our custom levels as well. For each level, we gather approximately 500 expert trajectories. Another 500 negative trajectories are collected by running a randomly initialized agent in the environment. The training dataset consists of up to 16 trajectories, including both expert and negative examples. All remaining trajectories constitute the test set. For each dataset, we evolve the reward on the train trajectories and report both train and test fitness from Eq. (3).

The state is represented by a $(h, w, 3)$ array. The state is fully observable, with the first channel containing information about the object type (with each integer corresponding to a different object, such as box, key, wall, or agent), the second channel contains information about the object's color and the third any extra information (e.g. agent direction, if is the door locked).

Android To assess GRACE in a high-dimensional, real-world setting, we use the AndroidControl dataset (Rawles et al., 2023; Li et al., 2024), which provides a rich collection of complex, multi-step human interactions across standard Android applications. The state space includes both raw screen pixels and the corresponding XML view hierarchy.

From this dataset, we curate a subset of trajectories focused on the Clock application, where users successfully complete tasks such as "set an alarm for 6AM." These serve as our positive examples. Negative samples are drawn from trajectories in other applications (e.g., Calculator, Calendar, Settings). For each negative trajectory, we randomly assign an instruction from the positive set, ensuring the instruction is clock-related but the trajectory completes a task in an unrelated app. We use 80% of trajectories in the train set and the remaining for the test set.

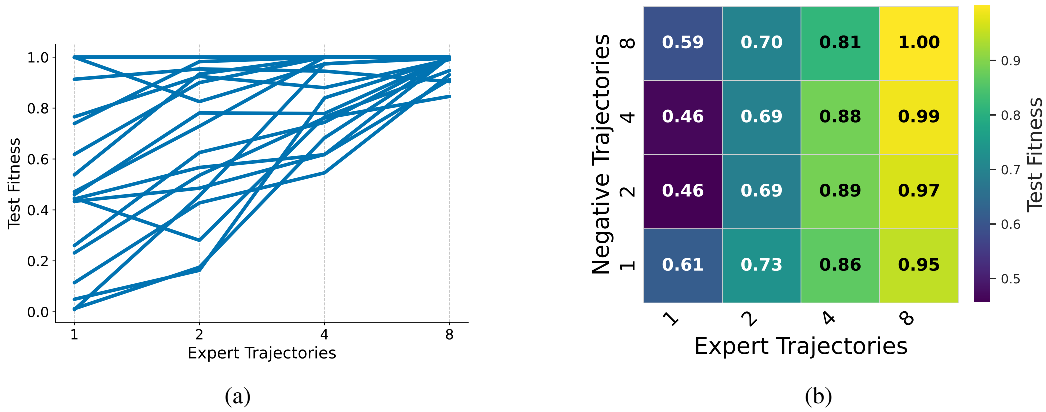

Figure 2: **Fitness vs Number of Expert Trajectories.** The fitness is computed on test dataset after obtaining maximum fitness on training data with corresponding number of expert and negative training trajectories. (a) Performance on all 20 BabyAI tasks. (b) Aggregate fitness across 20 BabyAI tasks.

MuJoCo We finally conduct additional experiments on 4 challenging tasks from the classical MuJoCo continuous control suite (Todorov et al., 2012): `Hopper`, `Walker`, `Ant`, `Humanoid`. These tasks demonstrate that GRACE also excels at reward design in continuous action and state spaces. In these experiments, we don't perform the goal identification step and simply classify all expert states as Goal states and all learner states as Non-Goal states. We run all our MuJoCo experiments using the fully differentiable physics engine Brax (Freeman et al., 2021) to speed up learning. Unlike the BabyAI and Android experiments, in MuJoCo we update the dataset 5 times ($M = 5$) with new trajectories coming from the learner policy. The reward is only updated if the fitness is low on the newly added trajectories.

GRACE Parameters All parameters of our approach used across our experiments can be found in Appendix A.6.

4.2 Analysis

GRACE recovers rewards with high accuracy. We first examine whether GRACE evolutionary search (Phase 1) can successfully recover the underlying task reward from demonstrations alone. We evaluate this in two settings using *BabyAI*: (i) a single-level setting, where the model infers a task-specific reward, and (ii) a more challenging multi-level setting, where GRACE must learn a single, general reward function conditioned on both state and a language goal.

In Figures 2 and 3, we show that the fitness consistently reaches 1.0 across all BabyAI tasks in both single- and multi-level settings, as well as on AndroidControl. A fitness of 1.0 corresponds to assigning higher values to all goal states than to non-goal states.

We further ablate two aspects of the algorithm. First, we analyze sample efficiency by varying the number of expert and negative demonstrations. Results on BabyAI (Figure 2a) show non-trivial performance even with a single demonstration, with gradual improvement and perfect scores achieved using only eight expert trajectories. The number of negative trajectories also plays a role, though to a lesser degree: for example, fitness of 0.95 is achieved with just a single negative trajectory, provided that sufficient expert trajectories are available (Figure 2b).

Finally, we assess the robustness and efficiency of the evolutionary process. As shown in Figure 3, in the multi-task setting GRACE reliably converges to a high-fitness reward function in fewer than 100 generations (i.e., evolutionary search steps), demonstrating the effectiveness of our LLM-driven refinement procedure.

GRACE outperforms other IRL and online RL: To validate the quality of the inferred reward model, we compare against two approaches. First, we employ PPO Schulman et al. (2017), as a representative algorithm for online RL, with both GRACE as a reward as well as a groundtruth

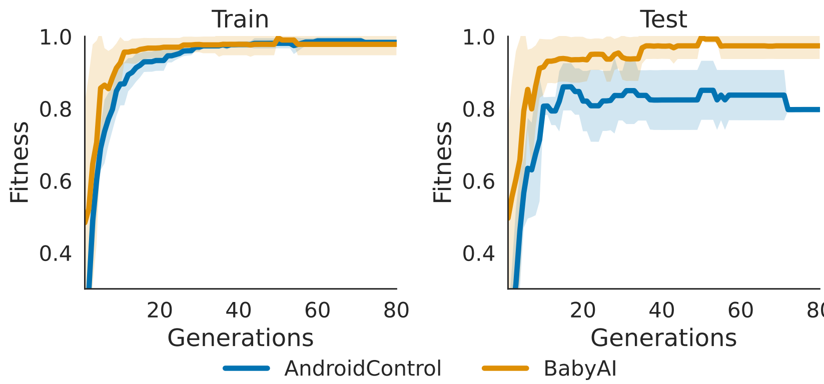

Figure 3: **Fitness vs Number of generations.** Evolution of train and test fitness across evolution generations, as defined by Algorithm 1, for BabyAI (multi-level settings) and AndroidControl (bottom) for "set alarm" task. For BabyAI, we provide 8 expert trajectories and 8 negative trajectories for each task. Shading is standard deviation across 3 seeds. For these experiments, no online data is added beyond the initial trajectories provided ($M = 1$).

sparse success reward. Clearly, the latter should serve as an oracle, while it does not benefit from dense rewards.

As an IRL baseline, we compare against GAIL (Ho & Ermon, 2016), that trains a policy whose behavior is indiscriminable from the expert data, as judged by a learned discriminator. GAIL is trained with a large dataset of $2,000$ expert trajectories per task, substantially larger than our train data of 8 expert trajectories.

As shown in Table 1 and 2, GRACE consistently matches or outperforms GAIL across all tasks with lesser training data. On several BabyAI tasks, GRACE matches Oracle PPO with ground-truth rewards, whereas GAIL completely fails. This demonstrates that the interpretable, code-based rewards from GRACE are practically effective, enabling successful downstream policy learning. To ensure a fair comparison, the agents for the GAIL baseline and GRACE are trained using the same underlying PPO implementation, agent architecture and hyperparameters as the oracle. Performance is measured by the final task success rate after 1e7 environment steps. No extra information or environment code is provided in context to GRACE.

Similarly, we use the evolved reward function on the AndroidControl dataset to finetune our agent on the Clock *AndroidWorld* tasks: ClockStopWatchPausedVerify, ClockStopWatchRunning and ClockTimerEntry. The agent obtains near perfect performance on the Stopwatch tasks zero-shot, but learning on our reward doesn't decrease performance. The training curves for all tasks are reported in Figure 4.

	PPO	**GRACE w/ GPT-4o**	**GRACE w/ Qwen3-Coder-30B**	**GAIL w/ 10 traj**	**GAIL w/ 200 traj**
Hopper	2212 ± 54	2143 ± 80	2106 ± 76	1902 ± 183	2056 ± 92
Walker	2675 ± 292	2072 ± 576	2229 ± 600	790 ± 90	1982 ± 101
Ant	6239 ± 237	5707 ± 210	6085 ± 804	3871 ± 408	5521 ± 674
Humanoid	6455 ± 302	5809 ± 106	5921 ± 301	4772 ± 251	6521 ± 337

Table 1: Average returns on 4 MuJoCo (BRAX) continuous control tasks. Average and standard deviation is reported across 5 different seeds. The total number of required LLM calls to recover a reward for each task averages at 200 for both GPT-4o and Qwen3-Coder-30B.

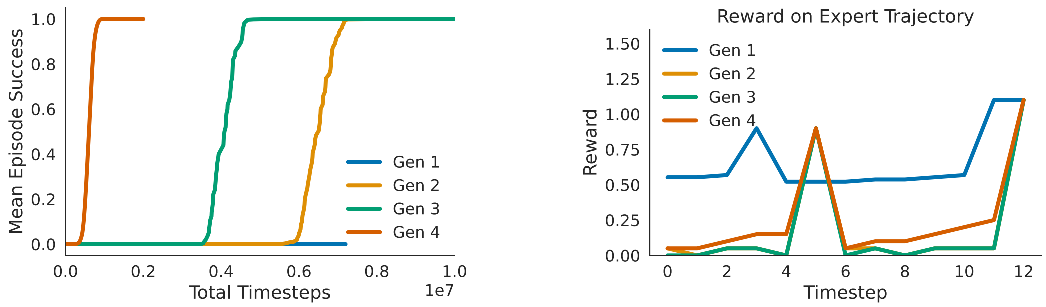

Figure 5: **Shaping** Using the default reward recovered by GRACE occasionally leads to failure in learning the correct behavior due to poor shaping. Through the targeted shaping in Phase 3, we significantly improve final performance and speed of learning.

Task	PPO	GAIL	GRACE
GoToRedBallNoDist	1.00	1.00	1.00
GoToRedBall	1.00	0.35	1.00
PickupDist	0.31	0.15	0.32
PickupLoc	0.21	0.00	0.26
GoToObj	1.00	0.92	1.00
OpenDoorColor	1.00	0.98	1.00
OpenTwoDoors	1.00	0.37	1.00
PlaceBetween (new)	0.09	0.01	0.09
OpenMatchingDoor (new)	0.79	0.20	0.35
Multi-task	0.95	0.31	0.92

Table 2: **Success rates on selected BabyAI environments.** GRACE compared against PPO and GAIL. GRACE uses 8 expert trajectories per task, while GAIL uses 2000.

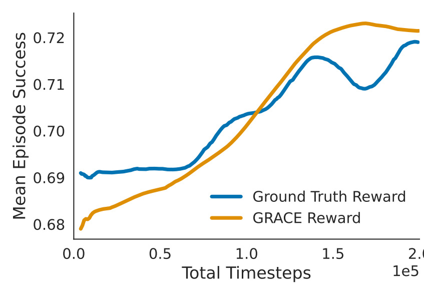

Figure 4: **Training Curves for *AndroidWorld* Clock Tasks.** Mean episode success over the 3 *AndroidWorld* clock tasks: ClockStopWatchPausedVerify, ClockStopWatchRunning, and ClockTimerEntry.

GRACE generates well shaped rewards: We demonstrate GRACE's ability to produce well-shaped rewards that accelerate learning. For challenging, long-horizon tasks like OpenTwoDoors, a correct but unshaped reward can lead to local optima where the agent gets stuck (Figure 5, "Gen 1"). By explicitly tasking the LLM to introduce shaping terms during Phase 3, GRACE refines the reward to provide a denser learning signal. As shown in Figure 5, this targeted shaping dramatically improves both the final performance and the speed of learning, allowing the agent to solve the task efficiently. This confirms that GRACE not only finds what the goal is but also learns how to guide an agent towards it.

GRACE Code Reuse: A key advantage of representing rewards as code is the natural emergence of reusable functions that collectively form a domain-specific reward library. We study this phenomenon in the multi-task *BabyAI* setting (Figure 6). In the early generations of evolutionary search, GRACE actively generates many new modules to explore alternative reward structures. After generation 10, the rate of new module creation drops sharply. At this point, GRACE shifts toward reusing the most effective, high-level modules it has already discovered.

To further illustrate this reuse, Figure 6 (right) shows call counts for a selected set of modules within the evolving reward API. For instance, the *Goal* module, which summarizes a set of goals, is initially used sparingly but becomes heavily invoked following a code refactor at generation 30. Likewise, the *agent_pos* function is reused at least five times after its introduction. These trends demonstrate that GRACE progressively builds a reward library that supports efficient multi-task generalization.

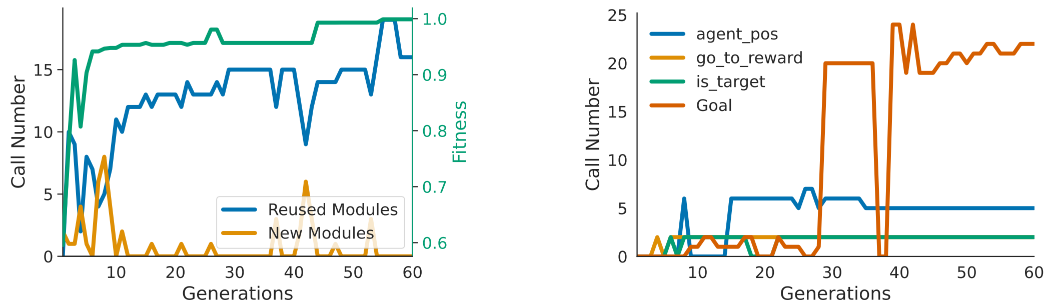

Figure 6: **Module and function reuse across generations** On the left, we show at each generation step the number of newly created modules and the number of existing and thus reused modules from prior rewards, contrasted with the fitness in the reward population. On the right, we show number of times a module are being re-used, for a select set of modules.

5 Discussion

Limitations A key limitation of GRACE is its limited scalability to high-dimensional state spaces for evolving reward functions. First, generating a reward from high-dimensional observations (such as pixels or waveform audio) directly requires the model to perform symbolic feature extraction. Second, the amount of expert and suboptimal trajectories that can be passed to the LLM is limited by its context length, which makes learning GRACE rewards from large datasets challenging.

Conclusion We introduce GRACE, a novel framework that leverages LLMs within an evolutionary search to address the critical challenge of interpretability in IRL. Our empirical results demonstrate that by representing reward functions as executable code, we can move beyond the "black-box" models of traditional IRL and produce rewards that are transparent, verifiable, and effective in RL learning. We show that GRACE successfully recovers accurate and generalizable rewards from few expert trajectories, in stark contrast to deep IRL methods like GAIL. This sample efficiency suggests that the strong priors and reasoning capabilities of LLMs provide a powerful inductive bias. Furthermore, we demonstrate the framework's practical utility by applying it to the complex AndroidWorld environment, showing that GRACE can learn rewards for a variety of tasks directly from unlabeled user interaction data with real-world applications.

6 Reproducibility Statement

To ensure the reproducibility of our research, we commit to making our code, datasets, and experimental configurations publicly available upon acceptance of this paper. We have already included extensive details within the paper itself. The appendix provides the full prompts used to interact with the LLM for goal identification, initial reward generation, evolutionary mutation, and reward shaping (Appendix A.9). Furthermore, all hyperparameters required to reproduce our results are listed in Appendix A.6.

References

Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML)*, pp. 1–8. ACM, 2004.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Michael Pavlov, Alethea Power, Lukasz Kaiser, Miljan Bavarian, Clemens Winter, Phil Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Guss, Alex Nichol, Igor Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Suyog Jain, William Saunders, Christopher Hesse, Mark Carr, Aitor Lewkowycz, David Dohan, Howard Mao, Lily Thompson, Erica Frank, Joshua Chen, Victor Butoi, David Hernandez, Liane DasSarma, Maxwell Chan, Mateusz Litwin, Scott Gray, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: A platform to study the sample efficiency of grounded language learning. *arXiv preprint arXiv:1810.08272*, 2018.

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 4299–4307, 2017.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. *Advances in Neural Information Processing Systems*, 37:60429–60474, 2024.

Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. In *International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=Y1XkzMJpPd`.

Farama Foundation, Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid: Modular customizable reinforcement learning environments. `https://github.com/Farama-Foundation/Minigrid`, 2025. Accessed: 2025-09-24.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL `http://github.com/google/brax`.

Daniel Fried, Joshua Ainslie, David Grangier, Tal Linzen, and Dani Yogatama. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations (ICLR)*, 2022.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. *arXiv preprint arXiv:2310.00166*, 2023.

Martin Klissarov, Devon Hjelm, Alexander Toshev, and Bogdan Mazoure. On the modeling capabilities of large language models for sequential decision making. *arXiv preprint arXiv:2410.05656*, 2024.

Raymond Li, Loubna Ben Allal, Yacine Jernite Zi, Denis Kocetkov, Chenxi Mou, Aleksandra Piktus, Laura Weber, Wenhao Xiao, Jihad Bibi, Stella Biderman, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on computer control agents. *arXiv e-prints*, pp. arXiv–2406, 2024.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv: Arxiv-2310.12931*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML)*, pp. 663–670. Morgan Kaufmann, 2000.

Erik Nijkamp, Richard Pang, Hiroaki Hayashi, Tian He, Baptiste Roziere, Canwen Xu, Susan Li, Dan Jurafsky, Luke Zettlemoyer, Veselin Stoyanov, and Hyung Won Chung. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations (ICLR)*, 2022.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025a.

Alexander Novikov, NgÃćn VÅľ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025b. URL `https://arxiv.org/abs/2506.13131`.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander MÄĚdry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoochian, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, Dane Sherburn, Daniel Kappler, Daniel Levin, Daniel Levy, David Carr, David Farhi, David Mely, David Robinson, David Sasaki, Denny Jin, Dev Valladares, Dimitris Tsipras, Doug Li, Duc Phong Nguyen, Duncan Findlay, Edede Oiwoh, Edmund Wong, Ehsan Asdar, Elizabeth Proehl, Elizabeth Yang, Eric Antonow, Eric Kramer, Eric Peterson, Eric Sigler, Eric Wallace, Eugene Brevdo, Evan Mays, Farzad Khorasani, Felipe Petroski Such, Filippo Raso, Francis Zhang, Fred von Lohmann, Freddie Sulit, Gabriel Goh, Gene Oden, Geoff Salmon, Giulio Starace, Greg Brockman, Hadi Salman, Haiming Bao, Haitang Hu, Hannah Wong, Haoyu Wang, Heather Schmidt, Heather Whitney, Heewoo Jun, Hendrik Kirchner, Henrique Ponde de Oliveira Pinto, Hongyu Ren, Huiwen Chang, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian O'Connell, Ian Osband, Ian Silber, Ian Sohl, Ibrahim Okuyucu, Ikai Lan, Ilya Kostrikov, Ilya Sutskever, Ingmar Kanitscheider, Ishaan Gulrajani, Jacob Coxon, Jacob Menick, Jakub Pachocki, James Aung, James Betker, James Crooks, James Lennon, Jamie Kiros, Jan Leike, Jane Park, Jason Kwon, Jason Phang, Jason Teplitz, Jason Wei, Jason Wolfe, Jay Chen, Jeff Harris, Jenia Varavva, Jessica Gan Lee, Jessica Shieh, Ji Lin, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joanne Jang, Joaquin Quinonero Candela, Joe Beutler, Joe

Landers, Joel Parish, Johannes Heidecke, John Schulman, Jonathan Lachman, Jonathan McKay, Jonathan Uesato, Jonathan Ward, Jong Wook Kim, Joost Huizinga, Jordan Sitkin, Jos Kraaijeveld, Josh Gross, Josh Kaplan, Josh Snyder, Joshua Achiam, Joy Jiao, Joyce Lee, Juntang Zhuang, Justyn Harriman, Kai Fricke, Kai Hayashi, Karan Singhal, Katy Shi, Kavin Karthik, Kayla Wood, Kendra Rimbach, Kenny Hsu, Kenny Nguyen, Keren Gu-Lemberg, Kevin Button, Kevin Liu, Kiel Howe, Krithika Muthukumar, Kyle Luther, Lama Ahmad, Larry Kai, Lauren Itow, Lauren Workman, Leher Pathak, Leo Chen, Li Jing, Lia Guy, Liam Fedus, Liang Zhou, Lien Mamitsuka, Lilian Weng, Lindsay McCallum, Lindsey Held, Long Ouyang, Louis Feuvrier, Lu Zhang, Lukas Kondraciuk, Lukasz Kaiser, Luke Hewitt, Luke Metz, Lyric Doshi, Mada Aflak, Maddie Simens, Madelaine Boyd, Madeleine Thompson, Marat Dukhan, Mark Chen, Mark Gray, Mark Hudnall, Marvin Zhang, Marwan Aljubeh, Mateusz Litwin, Matthew Zeng, Max Johnson, Maya Shetty, Mayank Gupta, Meghan Shah, Mehmet Yatbaz, Meng Jia Yang, Mengchao Zhong, Mia Glaese, Mianna Chen, Michael Janner, Michael Lampe, Michael Petrov, Michael Wu, Michele Wang, Michelle Fradin, Michelle Pokrass, Miguel Castro, Miguel Oom Temudo de Castro, Mikhail Pavlov, Miles Brundage, Miles Wang, Minal Khan, Mira Murati, Mo Bavarian, Molly Lin, Murat Yesildal, Nacho Soto, Natalia Gimelshein, Natalie Cone, Natalie Staudacher, Natalie Summers, Natan LaFontaine, Neil Chowdhury, Nick Ryder, Nick Stathas, Nick Turley, Nik Tezak, Niko Felix, Nithanth Kudige, Nitish Keskar, Noah Deutsch, Noel Bundick, Nora Puckett, Ofir Nachum, Ola Okelola, Oleg Boiko, Oleg Murk, Oliver Jaffe, Olivia Watkins, Olivier Godement, Owen Campbell-Moore, Patrick Chao, Paul McMillan, Pavel Belov, Peng Su, Peter Bak, Peter Bakkum, Peter Deng, Peter Dolan, Peter Hoeschele, Peter Welinder, Phil Tillet, Philip Pronin, Philippe Tillet, Prafulla Dhariwal, Qiming Yuan, Rachel Dias, Rachel Lim, Rahul Arora, Rajan Troll, Randall Lin, Rapha Gontijo Lopes, Raul Puri, Reah Miyara, Reimar Leike, Renaud Gaubert, Reza Zamani, Ricky Wang, Rob Donnelly, Rob Honsby, Rocky Smith, Rohan Sahai, Rohit Ramchandani, Romain Huet, Rory Carmichael, Rowan Zellers, Roy Chen, Ruby Chen, Ruslan Nigmatullin, Ryan Cheu, Saachi Jain, Sam Altman, Sam Schoenholz, Sam Toizer, Samuel Miserendino, Sandhini Agarwal, Sara Culver, Scott Ethersmith, Scott Gray, Sean Grove, Sean Metzger, Shamez Hermani, Shantanu Jain, Shengjia Zhao, Sherwin Wu, Shino Jomoto, Shirong Wu, Shuaiqi, Xia, Sonia Phene, Spencer Papay, Srinivas Narayanan, Steve Coffey, Steve Lee, Stewart Hall, Suchir Balaji, Tal Broda, Tal Stramer, Tao Xu, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Cunninghman, Thomas Degry, Thomas Dimson, Thomas Raoux, Thomas Shadwell, Tianhao Zheng, Todd Underwood, Todor Markov, Toki Sherbakov, Tom Rubin, Tom Stasi, Tomer Kaftan, Tristan Heywood, Troy Peterson, Tyce Walters, Tyna Eloundou, Valerie Qi, Veit Moeller, Vinnie Monaco, Vishal Kuo, Vlad Fomenko, Wayne Chang, Weiyi Zheng, Wenda Zhou, Wesam Manassra, Will Sheu, Wojciech Zaremba, Yash Patil, Yilei Qian, Yongjik Kim, Youlong Cheng, Yu Zhang, Yuchen He, Yuchen Zhang, Yujia Jin, Yunxing Dai, and Yury Malkov. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2016.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Nathan Ratliff, J. Andrew Bagnell, and Martin Zinkevich. Maximum margin planning. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pp. 729–736. ACM, 2006.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for android device control, 2023. URL https://arxiv.org/abs/2307.10088.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024a. doi: 10.1038/s41586-023-06924-6.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024b.

StÃl'phane Ross, Geoffrey Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 627–635, 2011.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2017.

Silvia Sapora, Gokul Swamy, Chris Lu, Yee Whye Teh, and Jakob Nicolaus Foerster. Evil: Evolution strategies for generalisable imitation learning, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 12, pp. 1057–1063, 1999.

Gokul Swamy, Sanjiban Choudhury, J Andrew Bagnell, and Steven Wu. Of moments and matching: A game-theoretic framework for closing the imitation gap. In *International Conference on Machine Learning*, pp. 10022–10032. PMLR, 2021.

Umar Syed and Robert E Schapire. A game-theoretic approach to apprenticeship learning. *Advances in neural information processing systems*, 20, 2007.

Yee Whye Teh, Victor Bapst, Wojciech M. Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 4499–4509, 2017.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

David Venuto, Mohammad Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with VLMs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 49368–49387. PMLR, 21–27 Jul 2024a. URL `https://proceedings.mlr.press/v235/venuto24a.html`.

David Venuto, Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with vlms. *arXiv preprint arXiv:2402.04764*, 2024b.

Yuxiang Wang, Yuchen Lin, Dongfu Jiang, Bill Y. Chen, Xiang Shen, Jidong Zhao, Xiang Yu, Chen Li, Xiao Qin, and Jie Sun. Llm-blender: Ensembling large language models with pairwise ranking and generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2023.

Aaron Wilson, Alan Fern, and Prasad Tadepalli. Multi-task reinforcement learning: A hierarchical bayesian approach. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, pp. 1015–1022. ACM, 2007.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

Abhay Zala, Jaemin Cho, Han Lin, Jaehong Yoon, and Mohit Bansal. Envgen: Generating and adapting environments via llms for training embodied agents. In *Conference on Language Modeling (CoLM)*, 2024.

William Zankner, Rohan Mehta, Eric Wallace, Jack Fitzsimons, Y. Yang, Alex Mei, Daniel Levy, William S. Moses, and Joseph E. Gonzalez. Critique-out-loud reward models. 2024. URL `https://arxiv.org/abs/2408.11791`.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1433–1438. AAAI Press, 2008.

A Appendix

A.1 Relations to Inverse Reinforcement Learning

Proposition 2. *Suppose $m(s) = 1$ iff $s \in \mathcal{S}_g$ else $m(s) = -1$, then GRACE optimizes, $\min_\pi \max_r J(\pi_E, m \circ r) - J(\pi, -m \circ r)$, which is a variation of Eq. (2)*

Proof. Suppose $m(s) = 1$ iff $s \in \mathcal{S}_g$ else $m(s) = -1$ is a mask over goal states. Then, the fitness function from Eq. 3 can be re-written in terms of the policy return akin to Eq. 1:

$$\begin{aligned} f(r) &= \mathbb{E}_{s\sim\mathcal{S}_g}[r(s)] - \mathbb{E}_{s\sim\mathcal{S}_{ng}}[r(s)] && (5)\\ &= \mathbb{E}_{\tau\sim D^+, s\in\tau}[m(s)r(s)] - \mathbb{E}_{\tau\sim D^-, s\in\tau}[-m(s)r(s)] && (6)\\ &= J(\pi_E, m\circ r) - J(\pi, -m\circ r) && (7)\end{aligned}$$

where m flips the reward value either if the state is non-goal and generated by the expert or it is a goal and generated by the learned policy.

The operator m can either be defined in Phase 1 by the LLM, or it can default to $m(s) = 1$ iff $s \in \mathcal{S}_E$ (expert states) or $m(s) = -1$ iff $s \in \mathcal{S}_L$ (learner states). Phase 2, the reward refinement stage is maximizing f w.r.t the reward. Phase 3, on the other side, is maximizing the return of π, or minimizing its negative. Thus, GRACE attempts to solve:

$$\min_\pi \max_r J(\pi_E, m\circ r) - J(\pi, -m\circ r)$$

□

A.2 Goal Identification

Goal identification is the critical first step (Phase 1) of the GRACE framework, where an LLM automatically labels states from expert demonstration trajectories ($\mathcal{D}^+$) as either goal states (s_g) or non-goal states (s_{ng}). This process creates the initial dataset that the evolutionary search uses to refine the reward functions. We evaluated the effectiveness of this automated approach using gpt-4o

(OpenAI et al., 2024), with the results presented in Table 3. The findings show that providing the model with textual representations of states is highly effective, achieving 94% accuracy. In contrast, relying on image-based input alone was significantly less effective, with accuracy dropping to 49%. However, it is likely that models with more comprehensive visual pre-training would be substantially better at identifying goal states from image-only inputs. This is still much better than chance, as the trajectories average around 20 steps. The experiment also tested performance on shuffled trajectories to see if the model relied on temporal order. Accuracy with text input saw a minor drop to 88%, indicating that while the model leverages the sequence of events, it is not entirely dependent on it to identify goal states.

Table 3: Model Accuracy Comparison

	gpt-4o w/		
Metric	Text	Images	Text and Images
Accuracy	0.94 ± 0.24	0.49 ± 0.38	0.88 ± 0.34
Accuracy on Shuffled	0.88 ± 0.48	0.49 ± 0.50	0.75 ± 0.43

In the more complex AndroidControl domain, GRACE showed a remarkable ability not only to identify the goal state within a trajectory but also to refine the task's textual instruction to accurately reflect the demonstrated behavior. A few examples highlight this robustness:

- **Refining Instructions to Match Behavior**: GRACE resolves ambiguities between an instruction and the corresponding trajectory. For instance, in a trajectory where the user was instructed to "set a timer" but did not start it, GRACE updated the instruction to explicitly include a "don't start the timer" clause. Similarly, when a user was asked to "set an alarm for 9am" but also performed the extra step of naming the alarm, GRACE appended the instruction to include the naming step, ensuring the final instruction precisely matched the expert demonstration.
- **Discarding Irrelevant Trajectories**: The system correctly identifies and filters out trajectories where the user's actions are inconsistent with the instruction's domain. When a user was instructed to perform a task in the 'Clock' app but completed it in the 'ClockBuddy' app, GRACE identified the application mismatch. This allowed the trajectory to be filtered from the dataset for the intended 'Clock' app task. A similar process occurred when a user was given a nonsensical instruction like "give me directions for X in the clock app" and then used Google Maps.

A.3 Additional Online Results

Task	PPO	GAIL	GRACE
OpenRedDoor	1.00	1.00	1.00
GoToObjS4	1.00	1.00	1.00
GoToRedBlueBall	0.96	0.40	0.99
GoToRedBallGrey	0.97	0.77	0.99
Pickup	0.10	0.00	0.09
Open	0.30	0.18	0.22
OpenRedBlueDoors	1.00	0.96	0.98
OpenDoorLoc	0.39	0.40	1.00
GoToLocalS8N7	0.64	0.39	0.97
GoToDoor	0.74	0.37	0.99
SortColors (new)	0.00	0.00	0.00

Table 4: **Success rates on additional BabyAI environments**. The performance of our method, GRACE, is compared against two key baselines: PPO, trained on the ground-truth reward, and GAIL, trained using 2000 expert trajectories per task. GRACE's performance is evaluated with 8 expert trajectories per task to demonstrate its high sample efficiency. All values represent the final success rate at the end of training.

A.4 Extended Discussion and Future Work

GRACE's reliance on programmatic reward functions introduces several limitations, particularly when compared to traditional deep neural network based approaches. These limitations also point toward promising directions for future research.

Input modality While generating rewards as code offers interpretability and sample efficiency, it struggles in domains where the reward depends on complex, high-dimensional perceptual inputs. Code is inherently symbolic and structured, making it less suited for interpreting raw sensory data like images or audio. For instance, creating a programmatic reward for a task like "navigate to the object that looks most fragile" is non-trivial, as "fragility" is a nuanced visual concept. NNs, in contrast, excel at learning features directly from this kind of data. Programmatic rewards can also be brittle: a small, unforeseen perturbation in the environment that violates a hard-coded assumption could cause the reward logic to fail completely, whereas NNs often degrade more gracefully.

Data Quantity GRACE demonstrates remarkable performance with very few demonstrations. This is a strength in data-scarce scenarios. However, it is a limitation when vast amounts of data are available. Deep IRL methods like GAIL are designed to scale with data and may uncover subtle, complex patterns from millions of demonstrations that would be difficult to capture in an explicit program. While GRACE's evolutionary search benefits from tight feedback on a small dataset, it is not clear how effectively it could learn from a massive dataset.

Failure Cases Although GRACE is highly sample-efficient, it is not a magic bullet. For example, in the BabyAI-OpenTwoDoors task, GRACE often proposed a reward that didn't take into account the order in which the doors were being opened. Similarly, in the new BabyAI-SortColors task, it would sometimes return a reward that only accounted for picking up and dropping both objects, without paying attention to where they were being dropped. While these errors can be easily fixed by providing a relevant negative trajectory or by treating all learner-generated states as negative trajectories, they highlight that GRACE can still misinterpret an agent's true intent based on expert demonstrations alone.

Hybrid Approaches These limitations can be substantially mitigated by extending the GRACE framework to incorporate tool use, combining the strengths of both systems. The LLM could be granted access to a library of pre-trained models (e.g., object detectors, image classifiers, or segmentation models). The LLM's task would then shift from writing low-level image processing code to writing high-level logic that calls these tools and reasons over their outputs. A final direction involves generating hybrid reward functions that are part code and part neural network. The LLM could define the overall structure, logic, and shaping bonuses in code, but instantiate a small, learnable NN module for a specific, difficult-to-program component of the reward. This module could then be fine-tuned using the available demonstrations, creating a reward function that is both largely interpretable and capable of handling perceptual nuance. By exploring these hybrid approaches, future iterations of GRACE could retain the benefits of interpretability and sample efficiency while overcoming the inherent limitations of purely programmatic solutions in complex, perception-rich environments.

A.5 New BabyAI Levels

To evaluate the generalization and reasoning capabilities of GRACE and mitigate concerns of data contamination from pre-existing benchmarks, we designed three novel BabyAI levels.

PlaceBetween The agent is placed in a single room with three distinct objects (e.g., a red ball, a green ball, and a blue ball). The instruction requires the agent to pick up a specific target object and place it on an empty cell that is strictly between the other two anchor objects. Success requires being on the same row or column as the two anchors, creating a straight line. This task moves beyond simple navigation, demanding that the agent understand the spatial relationship "between" and act upon a configuration of three separate entities.

OpenMatchingDoor This level is designed to test indirect object identification and chained inference. The environment consists of a single room containing one key and multiple doors of different colors. The instruction is to "open the door matching the key". The agent cannot solve the task by simply parsing an object and color from the instruction. Instead, it must first locate the key, visually identify its color, and then find and open the door of the corresponding color. This task assesses the agent's ability to perform a simple chain of reasoning: find object A, infer a property from it, and then use that property to identify and interact with target object B.

SortColors The environment consists of two rooms connected by a door, with a red ball in one room and a blue ball in the other. The instruction is a compound goal: "put the red ball in the right room and put the blue ball in the left room". To make the task non-trivial, the objects' initial positions are swapped relative to their goal locations. The agent must therefore execute a sequence of sub-tasks for each object: pick up the object, navigate to the other room, and drop it. This level tests the ability to decompose a complex language command and carry out a plan to satisfy multiple, distinct objectives.

A.6 Hyperparameters

Table 5: Hyperparameters for Training BabyAI with PPO

Parameter	Value
Base Model	llava-onevision-qwen2-0.5b-ov-hf
Gamma	0.999
Learning Rate	3e-5
Entropy Coef	1e-5
Num Envs	10
Num Steps	64
Episode Length	100
PPO Epochs	2
Num Minibatch	6

Table 6: Hyperparameters for Training AndroidWorld

Parameter	Value
Base Model	Qwen2.5-VL-3B-Instruct
LoRA Rank	512
LoRA Alpha	32
LoRA Dropout	0.1
Critic Hidden Size	2048
Critic Depth	4
Gamma	0.999
Learning Rate	3e-5
Entropy Coef	0.0
Num Envs	16
Num Steps	16
Episode Length	20
PPO Epochs	2
Num Minibatch	2

Table 7: Hyperparameters for GRACE Evolution

Parameter	Value
Population Size	20
Elite	4
Num Generations	100
Include expert trajectory chance	0.25
Incorrect state only chance	0.5
Expert state only chance	0.75
Model	gpt-4o

A.7 EVOLUTION EXAMPLES

```
def _parse_colour_from_text(text: Optional[str]) -> Optional[int]:
    if text is None:
        return None

    colour_words: Dict[str, int] = {
        "red": 0,
        "green": 1,
        "blue": 2,
 "yellow": 3,          "purple": 3,
  "yellow": 4,
        "orange": 5,  # keep old mapping
  "grey":  5, # alias for the observed colour code in the trajectory
  "gray":  5,
    }
     lower = text.lower()
    for word, code in colour_words.items():
        if word in text.lower(): lower:
            return code
    return None

def _parse_goal_type(text: Optional[str]) -> str:
    if text is None:
        return "key"
    txt = text.lower()
    if "ball" in txt:
        return "ball"
     if "box" in txt:
  return "box"
    return "key"
```

Figure 7: **GRACE iteratively refines the initial BabyAI reward function (iteration 0) to handle unseen entities (iteration 10).** Using execution traces, the agent fixes its color code mistake and adds a new `box` entity.

```
from __future__ import annotations

import re
from typing import Optional, Tuple

import numpy as np

COLOR2ID = {
    "red": 0,
    "green": 1,
    "blue": 2,
    "purple": 3,
    "yellow": 4,
    "grey": 5,
    "gray": 5,  # US spelling
}

OBJECT2ID = {
    "empty": 0,
    "wall": 1,
    "floor": 2,
    "door": 3,
    "key": 5,
    "ball": 6,
    "box": 8,
    "agent": 10,
}

 # Map MiniGrid direction codes (stored in the 3-rd channel of the agent cell)
 # to row/col deltas.  Empirically direction 0 points *down/south* in the
 # provided trajectories.
 DIR2VEC: dict[int, Tuple[int, int]] ={
  0:  (1, 0), # south
  1:  (0, 1), # east
```

```
    2:  (-1, 0), # north
    3:  (0, -1), # west
}

def _parse_goal(extra_info: str ) -> Tuple[int, Optional[int]]:
    """Return *(object_id, colour_id)* parsed from *extra_info*."""
    if not extra\_info:
        raise ValueError("extra_info must specify the target, e.g. 'the red ball'.")

    tokens = re.findall(r"[a-zA-Z]+", extra\_info.lower())
    obj_id: Optional[int] = None
    col_id: Optional[int] = None
    for tok in tokens:
        if obj_id is None and tok in OBJECT2ID:
        if tok in COLOR2ID and col_id is None:
            col_id = COLOR2ID[tok]
        if tok in OBJECT2ID and obj_id is None:
            obj_id = OBJECT2ID[tok]
        if col_id is None and tok in COLOR2ID:
            col_id = COLOR2ID[tok]
    if obj_id is None:
        raise ValueError(
            f"Could not parse target object from extra_info='{extra_info}'."
        )
    return obj_id, col_id  # colour may be None (wild-card)

class Reward:
    """Success when definition (single-step, dense reward):
    100.0 âĂŞ the **first** cell in front of the agent is *either*
    âĂŞ on / adjacent (according to the
            closest target object (Manhattan distance âL'd' 1), OR
    âĂŞ direction stored in the third observation channel) contains a
    matching target has disappeared from the observable grid (picked up).

    Shaping:  r = 1 / (1+d) with d the Manhattan distance to the closest
    still visible target, clipped at 0 object.
    <1.0 âĂŞ shaping reward 1/(d+1) otherwise.
    0.0 âĂŞ if either the agent or (a matching) target is out of view. not visible.

    The implementation is modular so new goal
    types can be handled by extending the OBJECT/COLOR lookup tables or by
    replacing the *success predicate*.
    """

    SUCCESS_REWARD = 100.0
    def __init__(self, extra\_info: Optional[str] str = None):
        self.tgt_obj_id, self.tgt_col_id self._target_obj_id, self._target_colour_id =
            _parse_goal(extra_info)

    def __call__(self, state:  np.ndarray) -> float:  # enable direct call
    return self.reward_fn(state)

    def reward_fn(self, state: np.ndarray) -> float:
        """state:  (H, W, 3) """
        agent_pos = self._find_agent(state)
        if agent_pos is None:
            return 0.0

        # mask of all target objects still visible
        tgt_mask = (state[:, :, 0] == self.tgt_obj_id) & (
        state[:, :, 1] == self.tgt_col_id
        )

        if not tgt_mask.any():
        # object gone -> picked up / carried
```

```
            return self.SUCCESS_REWARD

        # distance to the closest visible target
        tgt_positions = np.argwhere(tgt_mask)
        dists = np.abs(tgt_positions - agent_pos).sum(axis=1)

        target_positions = self._find_targets(state)
        if target_positions.size == 0:
         # No matching target in view -> no shaping.
         return 0.0

        # -------------------------------------------
        # Success predicate âĂŞ target must be directly in front of the agent.
        # -------------------------------------------
        if self._is_target_in_front(agent_pos, state):
         return 100.0

        # -------------------------------------------
        # Shaping:  inverse Manhattan distance (< 1.0) to the *nearest* target.
        # -------------------------------------------
        dists = np.abs(target_positions - agent_pos).sum(axis=1)
        min_dist = int(dists.min())
        if min_dist <= 1:
            return self.SUCCESS_REWARD

        return 1.0 / (1.0 + min_dist)

    @staticmethod
    def _find_agent(state: np.ndarray) -> Optional[np.ndarray]:
        """Return (row, col) of """Locate the first agent
            pixel found, in the observation (row, col) or None.""" *None* if absent."""
        locs = np.argwhere(state[:, :, 0] == OBJECT2ID["agent"])
        if locs.size == 0:
            return None
        return locs[0]

     def _find_targets(self, state:  np.ndarray) -> np.ndarray:
  """Return an (N, 2) array of row/col positions of matching targets."""
  obj_mask = state[:, :, 0] == self._target_obj_id
  if self._target_colour_id is not None:
  col_mask = state[:, :, 1] == self._target_colour_id
  mask = obj_mask & col_mask
  else:
  mask = obj_mask
  return np.argwhere(mask)

  def _is_target_in_front(self, agent_pos:  np.ndarray, state:  np.ndarray) -> bool:
  """Return *True* iff the cell directly in front of the agent matches target."""
  row, col = agent_pos
  agent_dir = int(state[row, col, 2])
  drow, dcol = DIR2VEC.get(agent_dir, (1, 0)) # default to south if unknown
  f_row, f_col = row + drow, col + dcol

  # Out of bounds âĂŠ cannot be success.
  if not (0 <= f_row < state.shape[0] and 0 <= f_col < state.shape[1]):
  return False

  # Check object id
  if state[f_row, f_col, 0] != self._target_obj_id:
  return False

  # Check colour if colour was specified.
```

```
if (
self._target_colour_id is not None
and state[f_row, f_col, 1] != self._target_colour_id
):
return False

return True
```

Figure 8: Example of code evolution across many generations.

A.8 GENERATED REWARDS

```
# ------------------------------------------------------------
#                         IMPORTS
# ------------------------------------------------------------
import json
import math
import re
from typing import Callable, List, Optional, Set, Tuple

# ------------------------------------------------------------
#                GENERIC & NORMALISATION HELPERS
# ------------------------------------------------------------


def _contains_any(text: str, keywords) -> bool:
    text_l = text.lower()
    return any(k.lower() in text_l for k in keywords)


def _has_stopwatch(text: str) -> bool:
    t = text.lower()
    return any(p in t for p in ("stopwatch", "stop watch", "stop-watch"))


# ---------------- Tab-selection helpers ---------------------


def _tab_selected(state: str, label: str) -> bool:
    pattern = (
        rf'"(content_description|text)"\s*:\s*"{label}"[^\n]*?"is_selected"\s*:\s*true'
    )
    return bool(re.search(pattern, state, re.I))


def _alarm_tab_selected(state: str) -> bool:
    return _tab_selected(state, "Alarm") or _tab_selected(state, "Alarms")


def _timer_tab_selected(state: str) -> bool:
    return _tab_selected(state, "Timer")


def _stopwatch_tab_selected(state: str) -> bool:
    return _tab_selected(state, "Stopwatch")


def _clock_tab_selected(state: str) -> bool:
    return _tab_selected(state, "Clock")


# ---------------- Text normalisation helper -----------------


def _normalize_time_text(txt: str) -> str:
    txt2 = txt.replace(";", ":")
    txt2 = re.sub(r"\b([ap])\s*(?:\.m\.|\.m|m)\b", r"\1m", txt2, flags=re.I)
    return txt2


# ------------------------------------------------------------
#                  TIMER / DURATION PARSING
# ------------------------------------------------------------


def _parse_requested_time(text: str) -> int:
    text = text.replace("-", " ")
    hours = minutes = seconds = 0
    for patt, mult in (
        (r"(\d+)\s*hour", 3600),
        (r"(\d+)\s*minute", 60),
        (r"(\d+)\s*second", 1),
    ):
        m = re.search(patt, text, re.I)
        if m:
            val = int(m.group(1)) * mult
            if mult == 3600:
                hours = val // 3600
            elif mult == 60:
                minutes = val // 60
            else:
```

```python
                seconds = val
    if hours == minutes == seconds == 0:
        m = re.search(r"(\d+)\s*-?\s*min", text, re.I)
        if m:
            minutes = int(m.group(1))
        else:
            m = re.search(r"(\d+)", text)
            if m:
                minutes = int(m.group(1))
    total = hours * 3600 + minutes * 60 + seconds
    return total if total > 0 else 60


# ------------------------------------------------------------
#               ADDITIONAL HELPERS
# ------------------------------------------------------------


def _parse_adjust_timer_amount(instr: str) -> Optional[int]:
    instr_l = instr.lower()
    verb = r"(?:add|increase|extend|plus|up|extra|more|additional)"
    unit = r"(hours?|minutes?|seconds?)"
    pat1 = re.compile(rf"{verb}\s+(\d+)\s*(?:more\s+)?{unit}")
    pat2 = re.compile(rf"by\s+(\d+)\s*{unit}")
    seconds: List[int] = []
    for pat in (pat1, pat2):
        for m in pat.finditer(instr_l):
            num = int(m.group(1))
            u = m.group(2)
            if u.startswith("hour"):
                seconds.append(num * 3600)
            elif u.startswith("minute"):
                seconds.append(num * 60)
            else:
                seconds.append(num)
    if seconds:
        return max(1, min(seconds))
    return None


def _parse_alarm_time(instr: str) -> Tuple[int, int, Optional[str]]:
    instr_n = _normalize_time_text(instr)
    instr_l = instr_n.lower()
    m = re.search(r"(\d{1,2})\s*[:.]\s*(\d{2})\s*(am|pm)?", instr_l)
    if m:
        h, minute, ap = int(m.group(1)), int(m.group(2)), m.group(3)
    else:
        m = re.search(r"\b(\d{1,2})\s*(am|pm)\b", instr_l)
        if m:
            h, minute, ap = int(m.group(1)), 0, m.group(2)
        else:
            return 7, 0, "am"
    if ap:
        ap = ap.lower()
        if ap == "pm" and h != 12:
            h += 12
        if ap == "am" and h == 12:
            h = 0
    return h % 24, minute, ap


def _extract_timer_components(state: str) -> Optional[Tuple[int, int, int]]:
    m = re.search(r"(\d+)\s*minutes?\s*(\d+)\s*seconds", state, re.IGNORECASE)
    if m:
        minutes = int(m.group(1))
        seconds = int(m.group(2))
        return (0, minutes, seconds)

    m = re.search(r"(\d+)h\s*(\d+)m\s*(\d+)s", state, re.IGNORECASE)
    if m:
        hours = int(m.group(1))
        minutes = int(m.group(2))
        seconds = int(m.group(3))
        return (hours, minutes, seconds)

    # Case 3: "MM:SS" format, ensuring it's not part of a timestamp (like 12:30 PM)
    for mm_match in re.finditer(r"(\d{1,2}):(\d{2})(?!\s*[AaPp][Mm])", state):
        mm, ss = int(mm_match.group(1)), int(mm_match.group(2))
        if not (0 <= ss < 60):
            continue
        context = state[mm_match.end() : mm_match.end() + 80].lower()
```

```
        if "minute" in context or "timer" in context or "remaining" in context:
            return (0, mm, ss)

    if not _timer_tab_selected(state):
        return None

    tokens = re.findall(r'"text"\s*:\s*"([^"]+)"', state)
    tokens = [t.strip() for t in tokens]

    for i in range(len(tokens) - 4):
        if (
            re.fullmatch(r"\d{1,2}", tokens[i])
            and tokens[i + 1] == ":"
            and re.fullmatch(r"\d{2}", tokens[i + 2])
            and tokens[i + 3] == ":"
            and re.fullmatch(r"\d{2}", tokens[i + 4])
        ):
            h = int(tokens[i])
            m_val = int(tokens[i + 2])
            s = int(tokens[i + 4])
            if 0 <= m_val < 60 and 0 <= s < 60:
                return (h, m_val, s)

    for i in range(len(tokens) - 2):
        if (
            re.fullmatch(r"\d{1,2}", tokens[i])
            and tokens[i + 1] == ":"
            and re.fullmatch(r"\d{2}", tokens[i + 2])
        ):
            m_val = int(tokens[i])
            s_val = int(tokens[i + 2])
            if 0 <= s_val < 60:
                return (0, m_val, s_val)

    return None

def _extract_timer_value(state: str) -> int:
    timer_components = _extract_timer_components(state)
    if timer_components:
        hh, mm, ss = timer_components
        return int(hh) * 3600 + int(mm) * 60 + int(ss)
    else:
        return None

# --- UI helpers ------------------------------------------------------


def _button_visible(state: str, label: str) -> bool:
    return bool(
        re.search(rf'"(content_description|text)"\s*:\s*"{label}"', state, re.I)
    )


def _timer_screen_visible(state: str) -> bool:
    if _timer_tab_selected(state):
        return True
    s = state.lower()
    return "remaining" in s or "minutes timer" in s


def _is_timer_running(state: str) -> bool:
    return _button_visible(state, "Pause")


def _timer_keypad_mode(state: str) -> bool:
    return bool(re.search(r"\b\d{1,2}h\s*\d{1,2}m\s*\d{1,2}s\b", state))


def _is_timer_paused(state: str) -> bool:
    if _timer_keypad_mode(state):
        return False
    if _button_visible(state, "Start") and not _button_visible(state, "Pause"):
        return True
    if not _timer_screen_visible(state):
        return False
    s = state.lower()
    return "timer paused" in s or ("paused" in s and "timer" in s)


def _timer_keypad_zero(state: str) -> bool:
    if not all(
```

```python
        re.search(rf'"text"\s*:\s*"{lbl}"', state, re.I)
        for lbl in ("hour", "min", "sec")
    ):
        return False
    return len(re.findall(r'"text"\s*:\s*"0{2}"', state)) >= 3


def _timer_deleted(state: str) -> bool:
    s = state.lower()
    if "no timers" in s:
        return True
    val = _extract_timer_value(state)
    if val == 0 and not _is_timer_running(state):
        return True
    return _timer_keypad_zero(state)


def _stopwatch_running(state: str) -> bool:
    return (
        _button_visible(state, "Pause")
        or _button_visible(state, "Stop")
        or "stopwatch running" in state.lower()
    )


def _stopwatch_time_zero(state: str) -> bool:
    if re.search(r"\b0{1,2}(?::0{2}){1,3}\b(?!:\d{2})", state):
        return True
    nums = re.findall(r'"text"\s*:\s*"(\d{2})"', state)
    return bool(nums) and all(n == "00" for n in nums)


def _timer_paused_notification(state: str) -> bool:
    return bool(
        re.search(r"the\s+clock\s+notification:\s*timer", state, re.I)
        or re.search(r"timer\s+paused", state, re.I)
    )


def _alarm_context_present(state: str) -> bool:
    return _alarm_tab_selected(state) or bool(re.search(r"\balarm\b", state, re.I))


def _parse_new_timer_label(instr_l: str) -> str:
    for kw in (" as ", " named ", " called ", " name "):
        if kw in instr_l:
            part = instr_l.split(kw, 1)[1]
            part = re.split(r"[.,;]|\bfor\b|\btimer\b", part, flags=re.I)[0]
            return part.strip()
    return ""


def _timer_label_present(state: str, label: str) -> bool:
    if not label:
        return False
    return bool(
        re.search(
            rf'"(text|content_description)"\s*:\s*"{re.escape(label)}"', state, re.I
        )
    )


def _safe_json_dumps(obj) -> str:
    try:
        return json.dumps(obj, ensure_ascii=False)
    except Exception:
        return json.dumps({"error": "debug-serialization failed"})


def _any_alarm_present(state: str) -> bool:
    sl = state.lower()
    if "alarm set" in sl:
        return True
    if _alarm_tab_selected(state) and re.search(r"\b\d{1,2}:\d{2}\s*(?:am|pm)\b", sl):
        return True
    return False


def _is_alarm_deleted(state: str) -> bool:
    s = state.lower()
    return any(
```

```
        re.search(p, s)
        for p in (
            r"alarm (deleted|removed|dismissed)",
            r"\bno (active )?alarms?\b",
            r"tap here to create an alarm",
            r"alarm deleted",
        )
    )


def _snooze_completed(state: str) -> bool:
    s_low = state.lower()
    if "alarm snoozed" in s_low:
        return True
    if re.search(r"snoozed\s+for\s+\d+", s_low):
        return True
    if re.search(r"\bsnooz(ing|ed)\b", s_low):
        return True
    if "select snooze duration" in s_low:
        return True
    return False


def _rename_dialog_open(state: str) -> bool:
    s = state.lower()
    if "enter timer name" in s:
        return True
    has_buttons = re.search(r'"text"\s*:\s*"(ok|cancel)"', state, re.I)
    has_edit = re.search(r'"is_editable"\s*:\s*true', state, re.I)
    return bool(has_buttons and has_edit)


def _detect_alarm_time(state: str) -> bool:
    return bool(re.search(r"\b\d{1,2}\s*:\s*\d{2}(?:\s*[ap]m)?\b", state, re.I))


def _selected_weekdays(state: str) -> Set[str]:
    selected = set()
    for key, full, abbrev in (
        ("sunday", "Sunday", "S"),
        ("monday", "Monday", "M"),
        ("tuesday", "Tuesday", "T"),
        ("wednesday", "Wednesday", "W"),
        ("thursday", "Thursday", "T"),
        ("friday", "Friday", "F"),
        ("saturday", "Saturday", "S"),
    ):
        patt = rf'("content_description"|"text")\s*:\s*"(?:{full}|{abbrev})"[^\n]*?("
         is_selected"|"is_checked")\s*:\s*true'
        if re.search(patt, state, re.I):
            selected.add(key)
    return selected


def _alarm_time_present(state: str, hour24: int, minute: int, ap: Optional[str]):
    s = state.lower().replace("\u200a", "")
    h12 = hour24 % 12 or 12
    patterns = [rf"\b0*{h12}:{minute:02d}\s*(?:am|pm)?\b"]
    if minute == 0:
        patterns.append(rf"\b0*{h12}\s*(?:am|pm)\b")
    patterns.append(rf"\b0*{hour24}:{minute:02d}\b")
    for p in patterns:
        if re.search(p, s):
            if ap and not re.search(rf"{ap}\b", s):
                continue
            return True
    return False


# --------------- NEW HELPER ----------------------------------------


def _day_toggle_buttons_visible(state: str) -> bool:
    """Detect if the row of weekday toggle buttons is visible."""
    matches = re.findall(r'"text"\s*:\s*"(S|M|T|W|F)"', state)
    unique = set(matches)
    return len(matches) >= 5 and len(unique.intersection({"M", "T", "W", "F"})) >= 3


# -------------------------------------------------------------
#                          REWARD CLASS
```

```
# ------------------------------------------------------------


class Reward:
    """Dense reward function for Google Clock tasks."""

    _SHAPING_INC = 0.3
    _ADJ_INC_THRESHOLD = 10

    # --------------------------------------------------------
    #                       INIT
    # --------------------------------------------------------
    def __init__(self, extra_info: Optional[str] = None):
        self.raw_instr: str = extra_info or ""
        self.instruction: str = self.raw_instr.lower()
        self.instruction_norm_full = _normalize_time_text(self.raw_instr)
        self.instruction_norm = self.instruction_norm_full.lower()

        # Task detection
        self.task_type = self._infer_task()

        # Stopwatch flags
        self.restart_mode = False
        self._reset_seen = False

        # Goal parsing / bookkeeping
        self.goal_seconds = 0
        self.goal_label = ""
        self.goal_hour24 = 0
        self.goal_minute = 0
        self.goal_hms = (0, 0, 0)
        self.goal_ap: Optional[str] = None
        self.city_keyword = ""
        self.city_keywords: List[str] = []
        self.recurrence_days: Set[str] = set()
        self.alarm_any_time = False

        # Timer-adjust bookkeeping
        self.initial_timer_val: Optional[int] = None
        self.prev_timer_val: Optional[int] = None
        self.max_timer_val: Optional[int] = None
        self.increments = 0
        self.needed_increments = 0
        self._countdown_seen = False

        # Alarm creation flag
        self._alarm_creation_seen = False

        # delete-alarm bookkeeping
        self._alarm_present_ever = False

        # adjust-alarm bookkeeping
        self.orig_hour24 = 0
        self.orig_minute = 0
        self._orig_seen = False

        # pause-timer stability tracking
        self._prev_timer_val_for_pause: Optional[int] = None
        self._same_val_steps: int = 0

        # snooze-specific
        self._snooze_dialog_seen = False

        # Generic bookkeeping
        self.goal_achieved = False
        self._best_level = 0
        self._t = 0
        self._confirm_goal_seen = False

        # Map tasks to progress-functions
        self._progress_fns: dict[str, Callable[[str], int]] = {
            "reset_stopwatch": self._pl_reset_stopwatch,
            "restart_stopwatch": self._pl_restart_stopwatch,
            "start_stopwatch": self._pl_start_stopwatch,
            "pause_stopwatch": self._pl_pause_stopwatch,
            "pause_timer": self._pl_pause_timer,
            "delete_timer": self._pl_delete_timer,
            "delete_alarm": self._pl_delete_alarm,
            "add_city": self._pl_add_city,
            "set_alarm": self._pl_set_alarm,
            "adjust_alarm": self._pl_adjust_alarm,
```

```
            "rename_timer": self._pl_rename_timer,
        }

        # Goal-specific parsing / bookkeeping
        if self.task_type == "set_timer" or self.task_type == "run_timer":
            self.goal_seconds = _parse_requested_time(self.instruction)
            h = self.goal_seconds // 3600
            rem = self.goal_seconds % 3600
            m = rem // 60
            s = rem % 60
            self.goal_hms = (h, m, s)
        if self.task_type == "adjust_timer":
            inc_secs = _parse_adjust_timer_amount(
                self.instruction_norm_full
            ) or _parse_requested_time(self.instruction)
            self.goal_seconds = max(1, inc_secs)
            self.needed_increments = max(1, math.ceil(self.goal_seconds / 60))
        if self.task_type == "rename_timer":
            self.goal_seconds = _parse_requested_time(self.instruction)
            self.goal_label = _parse_new_timer_label(self.instruction)
        if self.task_type == "set_alarm":
            explicit = re.search(
                r"\d{1,2}(:\d{2})?\s*(am|pm)", self.instruction_norm_full, re.I
            )
            if explicit:
                self.alarm_any_time = False
                self._parse_alarm_goal_time()
            else:
                self.alarm_any_time = True
            self.recurrence_days = self._parse_recurrence_days(self.instruction_norm)
        if self.task_type == "adjust_alarm":
            self.goal_hour24, self.goal_minute = self._parse_adjusted_alarm()
            self.goal_ap = None
            self.orig_hour24, self.orig_minute, _ = _parse_alarm_time(
                self.instruction_norm_full
            )
        if self.task_type == "add_city":
            self.city_keyword = self._parse_city_name(self.instruction) or "italy"
            self.city_keywords = [self.city_keyword]
            first = self.city_keyword.split()[0] if self.city_keyword else ""
            if first and first not in self.city_keywords:
                self.city_keywords.append(first)
        if self.task_type == "reset_stopwatch":
            if re.search(r"\brestart\b", self.instruction) or re.search(
                r"start\s+(?:over|again)", self.instruction
            ):
                self.restart_mode = True

    # --------------------------------------------------------
    #                PUBLIC API
    # --------------------------------------------------------
    def reward_fn(self, state: str) -> float:
        self._t += 1
        if self.task_type == "set_alarm":
            self._update_alarm_creation_seen(state)
        if self.goal_achieved:
            return 100.0
        if self.task_type in self._progress_fns:
            return self._reward_from_progress(self._progress_fns[self.task_type], state)
        if self.task_type == "set_timer" or self.task_type == "run_timer":
            return self._reward_timer(state, self.task_type == "set_timer")
        if self.task_type == "adjust_timer":
            return self._reward_adjust_timer(state)
        if self.task_type == "snooze_alarm":
            return self._reward_snooze(state)
        return 0.0

    def debug_fn(self, state: str) -> str:
        dbg = {
            "step": self._t,
            "task_type": self.task_type,
            "goal_achieved": self.goal_achieved,
            "best_level": self._best_level,
        }
        if self.task_type in {"set_timer", "run_timer", "adjust_timer"}:
            dbg.update(
                {
                    "goal_seconds": self.goal_seconds,
                    "increments": self.increments,
                    "countdown_seen": self._countdown_seen,
                }
```

```
            )
        if self.task_type == "rename_timer":
            dbg["goal_label"] = self.goal_label
        if self.task_type == "snooze_alarm":
            dbg["dialog_seen"] = self._snooze_dialog_seen
        return _safe_json_dumps(dbg)

    # --------------------------------------------------------
    #               TASK INFERENCE
    # --------------------------------------------------------
    def _infer_task(self) -> str:
        instr = self.instruction
        has_sw = _has_stopwatch(instr)

        if has_sw and _contains_any(instr, ["pause", "stop"]):
            return "pause_stopwatch"
        elif has_sw and _contains_any(
            instr, ["restart", "start over", "start again", "begin again"]
        ):
            return "restart_stopwatch"
        if has_sw and _contains_any(instr, ["reset", "zero", "set to zero", "clear"]):
            return "reset_stopwatch"
        if has_sw:
            return "start_stopwatch"

        if (
            (re.search(r"\btime\b", instr) or "clock" in instr)
            and re.search(r"\bin\s+\w+", instr)
            and not _contains_any(instr, ["timer", "alarm"])
        ):
            return "add_city"

        if "timer" in instr:
            if _contains_any(instr, ["delete", "remove", "clear"]):
                return "delete_timer"
            if _contains_any(instr, ["pause", "stop", "cancel"]):
                return "pause_timer"
            if _contains_any(instr, ["rename", "name", "called", "label"]):
                return "rename_timer"
            if re.search(
                r"\badd\b[^\n]*?\b\d+\s*(?:hour|minute|second)s?\s+timer", instr
            ):
                dont_start_req = bool(
                    re.search(
                        r"(?:\b(?:don’?t|do\s+not)\s+(?:start|run)\b)"
                        r"|(?:\bwithout\s+starting\b)"
                        r"|(?:\b(?:but|and)\s+don’?t\s+start\b)"
                        r"|(?:\bleave\s+it\s+paused\b)"
                        r"|(?:\bkeep\s+it\s+paused\b)",
                        instr,
                    )
                )
                if dont_start_req:
                    return "set_timer"
                else:
                    return "run_timer"
            if _contains_any(instr, ["increase", "extend", "more", "up"]):
                return "adjust_timer"
            if re.search(
                r"\badd\b[^\n]*?\b(minutes?|hours?|seconds?)\b[^\n]*?\bto\b[^\n]*?\btimer\b",
                instr,
            ):
                return "adjust_timer"
            return "run_timer"

        if "snooze" in instr:
            return "snooze_alarm"
        if _contains_any(instr, ["delete", "remove"]) and "alarm" in instr:
            return "delete_alarm"
        if "alarm" in instr and _contains_any(
            instr,
            [
                "delay",
                "resched",
                "push",
                "move",
                "change",
                "shift",
                "defer",
                "later",
                "increase",
```

```
            ],
        ):
            return "adjust_alarm"
        if "alarm" in instr:
            return "set_alarm"

        if _contains_any(
            instr, ["add", "timezone", "time zone", "city", "world clock"]
        ):
            return "add_city"
        return "none"

    def _update_alarm_creation_seen(self, state: str):
        s = state.lower()
        if any(kw in s for kw in ("add alarm", "alarm time", "select time")):
            self._alarm_creation_seen = True

    # ---------------------------------------------------------
    #           GENERIC reward helpers
    # ---------------------------------------------------------
    def _reward_from_progress(self, fn: Callable[[str], int], state: str) -> float:
        lvl = fn(state)
        if self.task_type == "set_alarm":
            if lvl >= 3:
                if self._alarm_creation_seen:
                    self.goal_achieved = True
                    return 100.0
                if self._confirm_goal_seen or self._best_level >= 2:
                    self.goal_achieved = True
                    return 100.0
                self._confirm_goal_seen = True
                self._best_level = max(self._best_level, 2)
                return 0.99
            self._confirm_goal_seen = False
        if lvl >= 3:
            self.goal_achieved = True
            return 100.0
        if lvl > self._best_level:
            inc = (lvl - self._best_level) * self._SHAPING_INC
            self._best_level = lvl
            return min(inc, 0.99)
        return 0.0

    # ---------------------------------------------------------
    #                  TIMER-specific dense reward
    # ---------------------------------------------------------
    def _reward_timer(self, state: str, start_req: bool) -> float:
        reward = 0.0
        if _timer_tab_selected(state):
            reward += 0.2
        current_val = _extract_timer_components(state)
        if current_val is None:
            return min(reward, 0.99)
        cur_hh, cur_mm, cur_ss = current_val
        current_digit_string = f"{cur_hh:02d}{cur_mm:02d}{cur_ss:02d}".lstrip("0")
        if current_digit_string == "":
            current_digit_string = "0"
        goal_digit_string = f"{self.goal_hms[0]:02d}{self.goal_hms[1]:02d}{self.goal_hms[2]:02d}".lstrip("0")
        if goal_digit_string == "":
            goal_digit_string = "0"
        running = _is_timer_running(state)
        if current_digit_string == goal_digit_string and running:
            if start_req and running:
                self.goal_achieved = True
                return 100.0
            if not start_req and not running:
                self.goal_achieved = True
                return 100.0
        matching_digits = 0
        for i in range(0, min(len(current_digit_string), len(goal_digit_string))):
            if goal_digit_string[i] == current_digit_string[i]:
                matching_digits += 1
            else:
                # Stop counting as soon as a mismatch occurs
                break
        reward += (matching_digits / len(goal_digit_string)) * 0.7
        return min(reward, 0.99)

    # ---------------------------------------------------------
    #           Other dense rewards (adjust_timer, snooze)
```

```
    # ---------------------------------------------------------
    def _reward_adjust_timer(self, state: str) -> float:
        reward = 0.0
        if _timer_screen_visible(state):
            reward += 0.2
        current_val = _extract_timer_value(state)
        if current_val is None:
            return min(reward, 0.99)
        if self.initial_timer_val is None:
            self.initial_timer_val = self.prev_timer_val = self.max_timer_val = (
                current_val
            )
            return min(reward, 0.99)
        if current_val > (self.max_timer_val or 0):
            self.max_timer_val = current_val
        diff_step = current_val - (self.prev_timer_val or current_val)
        if diff_step > self._ADJ_INC_THRESHOLD:
            self.increments += max(1, int(round(diff_step / 60.0)))
        elif diff_step < -1:
            self._countdown_seen = True
        self.prev_timer_val = current_val
        net_increase_max = (self.max_timer_val or current_val) - self.initial_timer_val
        fraction_by_inc = self.increments / max(1, self.needed_increments)
        fraction_by_delta = net_increase_max / max(1, self.goal_seconds)
        progress_fraction = min(1.0, max(fraction_by_inc, fraction_by_delta))
        reward += 0.8 * progress_fraction
        tol = max(2, int(self.goal_seconds * 0.05))
        goal_reached_primary = (
            self.increments >= self.needed_increments
            or net_increase_max >= self.goal_seconds - tol
        )
        committed = (
            _is_timer_running(state) or _is_timer_paused(state) or self._countdown_seen
        )
        keypad = _timer_keypad_mode(state)
        secondary_success = (
            not goal_reached_primary
            and net_increase_max >= 0.4 * self.goal_seconds
            and self.increments >= 1
            and self._countdown_seen
            and committed
            and not keypad
        )
        if (goal_reached_primary or secondary_success) and committed and not keypad:
            self.goal_achieved = True
            return 100.0
        return min(reward, 0.99)

    def _reward_snooze(self, state: str) -> float:
        s_low = state.lower()
        if "select snooze duration" in s_low:
            self._snooze_dialog_seen = True
        classic_done = (
            "alarm snoozed" in s_low
            or bool(re.search(r"snoozed\s+for\s+\d+", s_low))
            or bool(re.search(r"\bsnooz(ing|ed)\b", s_low))
        )
        row_done = (
            self._snooze_dialog_seen
            and "select snooze duration" not in s_low
            and "snooze" in s_low
            and bool(re.search(r"\b\d+\s+minutes?\b", s_low))
        )
        if classic_done or row_done:
            self.goal_achieved = True
            return 100.0
        reward = 0.0
        if _alarm_tab_selected(state):
            reward += 0.2
        if re.search(r'"(content_description|text)"\s*:\s*"snooze"', state, re.I):
            reward += 0.3
        if "select snooze duration" in s_low:
            reward += 0.2
        return min(reward, 0.99)

    # ---------------------------------------------------------
    #       Progress-level helpers (stopwatch/timer/alarm)
    # ---------------------------------------------------------
    def _pl_reset_stopwatch(self, state: str) -> int:
        if self.restart_mode:
            if _stopwatch_running(state) and self._reset_seen:
```

```
                return 3
            if _stopwatch_time_zero(state):
                self._reset_seen = True
                return 2
            if _button_visible(state, "Reset") and (
                _stopwatch_tab_selected(state) or "stopwatch" in state.lower()
            ):
                return 1
            return 0
        if _stopwatch_time_zero(state):
            return 3
        if _button_visible(state, "Reset") and (
            _stopwatch_tab_selected(state) or "stopwatch" in state.lower()
        ):
            return 2
        if _stopwatch_tab_selected(state):
            return 1
        return 0

    def _pl_pause_stopwatch(self, state: str) -> int:
        if not _stopwatch_running(state):
            return 3
        if _stopwatch_tab_selected(state):
            return 1
        return 0

    def _pl_restart_stopwatch(self, state: str) -> int:
        running = _stopwatch_running(state)
        at_zero = _stopwatch_time_zero(state)
        if running and self._reset_seen:
            return 3
        if at_zero:
            self._reset_seen = True
            return 2
        if _stopwatch_tab_selected(state):
            return 1
        return 0

    def _pl_start_stopwatch(self, state: str) -> int:
        if _stopwatch_running(state):
            return 3
        if "stopwatch" in state.lower() or _stopwatch_tab_selected(state):
            return 2
        if _contains_any(state.lower(), ["the clock", '"clock"', "alarms", "timer"]):
            return 1
        return 0

    def _pl_pause_timer(self, state: str) -> int:
        if _is_timer_paused(state):
            return 3
        current_val = _extract_timer_value(state)
        if current_val is not None:
            if self._prev_timer_val_for_pause == current_val:
                self._same_val_steps += 1
            else:
                self._same_val_steps = 0
            self._prev_timer_val_for_pause = current_val
        else:
            self._same_val_steps = 0
        stable_and_visible = (
            _timer_tab_selected(state)
            and current_val is not None
            and self._same_val_steps >= 1
            and not _is_timer_running(state)
        )
        if stable_and_visible:
            return 3
        if _timer_paused_notification(state) and _timer_tab_selected(state):
            return 3
        if _timer_paused_notification(state):
            return 2
        if _is_timer_running(state):
            return 2
        if _timer_tab_selected(state):
            return 1
        return 0

    def _pl_delete_timer(self, state: str) -> int:
        if _timer_deleted(state):
            return 3
        if _contains_any(
```

```
            state.lower(), ["delete", "remove", "clear", "âŇń", "backspace", "cancel"]
        ):
            return 2
        if _timer_tab_selected(state):
            return 1
        return 0

    def _pl_delete_alarm(self, state: str) -> int:
        s_low = state.lower()
        had_alarm_before = self._alarm_present_ever
        alarm_now = _any_alarm_present(state) or _detect_alarm_time(state)
        if alarm_now:
            self._alarm_present_ever = True
        if _is_alarm_deleted(state) and had_alarm_before:
            return 3
        if " delete" in s_low or "ð§ŮŚ" in s_low or re.search(r"trash|remove", s_low):
            return 2
        if alarm_now:
            return 1
        return 0

    def _pl_add_city(self, state: str) -> int:
        city_seen = self.city_keywords and any(
            re.search(rf"\b{re.escape(kw)}\b", state, re.I) for kw in self.city_keywords
        )
        in_search = (
            re.search(r"search for a city", state, re.I)
            or "select time zone" in state.lower()
        )
        if city_seen and _clock_tab_selected(state) and not in_search:
            return 3
        if city_seen:
            return 2
        if _clock_tab_selected(state):
            return 1
        return 0

    def _pl_set_alarm(self, state: str) -> int:
        if self._alarm_goal_met(state):
            return 3
        if "select time" in state.lower() or "alarm set for" in state.lower():
            return 2
        if _alarm_tab_selected(state):
            return 1
        return 0

    def _pl_adjust_alarm(self, state: str) -> int:
        if not self._orig_seen and _alarm_time_present(
            state, self.orig_hour24, self.orig_minute, None
        ):
            self._orig_seen = True
        if (
            _alarm_time_present(state, self.goal_hour24, self.goal_minute, None)
            and self._orig_seen
        ):
            return 3
        if "select time" in state.lower() or "alarm set for" in state.lower():
            return 2
        if _alarm_tab_selected(state) or self._orig_seen:
            return 1
        return 0

    def _pl_rename_timer(self, state: str) -> int:
        dialog_open = _rename_dialog_open(state)
        label_seen = _timer_label_present(state, self.goal_label)
        if label_seen and not dialog_open:
            return 3
        if dialog_open:
            return 2
        if _timer_tab_selected(state):
            return 1
        return 0

    # ---------------------------------------------------------
    #  Additional parsing / goal-checking helpers
    # ---------------------------------------------------------
    def _parse_recurrence_days(self, instr_l: str) -> Set[str]:
        days = {
            "sunday",
            "monday",
```

```
            "tuesday",
            "wednesday",
            "thursday",
            "friday",
            "saturday",
            "weekdays",
            "weekday",
            "week day",
            "week days",
            "weekends",
            "every day",
            "everyday",
        }
        found: Set[str] = set()
        for d in days:
            if d in instr_l:
                if d in {
                    "weekdays",
                    "weekday",
                    "week day",
                    "week days",
                    "every day",
                    "everyday",
                }:
                    found.update(
                        {"monday", "tuesday", "wednesday", "thursday", "friday"}
                    )
                elif d == "weekends":
                    found.update({"saturday", "sunday"})
                else:
                    found.add(d)
        return found

    def _alarm_goal_met(self, state: str) -> bool:
        # time & presence
        if self.alarm_any_time:
            time_ok = _any_alarm_present(state)
        else:
            time_ok = _alarm_time_present(
                state, self.goal_hour24, self.goal_minute, self.goal_ap
            )
        if not time_ok or not _alarm_context_present(state):
            return False

        # recurrence handling
        if not self.recurrence_days:
            return True

        # exact match
        if self.recurrence_days.issubset(_selected_weekdays(state)):
            return True

        # lenient weekday rule
        weekdays_set = {"monday", "tuesday", "wednesday", "thursday", "friday"}
        if self.recurrence_days == weekdays_set and _day_toggle_buttons_visible(state):
            if "not scheduled" not in state.lower():  # ensure days have been picked
                return True
        return False

    def _parse_alarm_goal_time(self):
        times = self._extract_times(self.instruction_norm_full)
        if not times:
            self.goal_hour24, self.goal_minute, self.goal_ap = _parse_alarm_time(
                self.instruction_norm_full
            )
            return
        alarm_pos = self.instruction_norm.rfind("alarm")
        chosen = next((t[:3] for t in times if t[3] > alarm_pos), times[0][:3])
        self.goal_hour24, self.goal_minute, self.goal_ap = chosen

    def _parse_adjusted_alarm(self) -> Tuple[int, int]:
        base_h, base_m, _ = _parse_alarm_time(self.instruction_norm_full)
        m = re.search(
            r"\bby\s+(\d+)\s*(hour|hours|minute|minutes)\b", self.instruction_norm
        )
        if m:
            num = int(m.group(1))
            unit = m.group(2)
            delta = num * (60 if "hour" in unit else 1)
            total = (base_h * 60 + base_m + delta) % (24 * 60)
            return total // 60, total % 60
```

```
        time_tokens: List[Tuple[int, int]] = []
        pat = re.compile(r"(\d{1,2})(?:[:.]\s*(\d{2}))?\s*(am|pm)", re.I)
        for mt in pat.finditer(self.instruction_norm):
            h, mnt, ap = int(mt.group(1)), int(mt.group(2) or 0), mt.group(3).lower()
            if ap == "pm" and h != 12:
                h += 12
            if ap == "am" and h == 12:
                h = 0
            time_tokens.append((h % 24, mnt))
        if len(time_tokens) >= 2:
            return time_tokens[1]
        return base_h, base_m

    @staticmethod
    def _parse_city_name(instr_l: str) -> str:
        parts = instr_l.split("add", 1)
        if len(parts) >= 2:
            tokens = parts[1].strip().split()
            city = []
            for w in tokens:
                if w in {"the", "a", "an"}:
                    continue
                if w in {
                    "time",
                    "timezone",
                    "zone",
                    "city",
                    "in",
                    "to",
                    "for",
                    "app",
                    "on",
                    "world",
                    "country",
                }:
                    break
                city.append(w)
            if city:
                return " ".join(city).strip()
        if " in " in instr_l:
            _, after = instr_l.split(" in ", 1)
            tokens = after.strip().split()
            city = []
            for w in tokens:
                if w in {"the", "a", "an"}:
                    continue
                wd = w.rstrip(".,;!")
                if wd in {
                    "time",
                    "timezone",
                    "zone",
                    "city",
                    "for",
                    "app",
                    "on",
                    "world",
                    "country",
                }:
                    break
                city.append(wd)
            return " ".join(city).strip()
        return ""

    @staticmethod
    def _extract_times(instr: str) -> List[Tuple[int, int, str, int]]:
        instr_n = _normalize_time_text(instr)
        pat = re.compile(r"(\d{1,2})(?:[:.]\s*(\d{2}))?\s*(am|pm)", re.I)
        res = []
        for m in pat.finditer(instr_n):
            h, minute, ap = int(m.group(1)), int(m.group(2) or 0), m.group(3).lower()
            h24 = h % 12 + (12 if ap == "pm" else 0)
            res.append((h24 % 24, minute, ap, m.start()))
        return res
```

Listing 1: Android Control Generated Reward.

A.9 Prompts

Goal Identification Prompt

```
Given this reward code: {reward_code}

Trajectory:
{trajectory}

Please analyze the state sequence and the agent's instruction.
Identify the index of the goal state. The state indices are 1-based.

OUTPUT FORMAT:
Answer in a json format as follows:
'reasoning': Explain your reasoning for choosing the goal state(s).
'goal_state_indexes': A list of integers representing the 1-based
index of the goal state(s), or -1 if no goal state is present.
```

Prompt 1: The prompt for identifying the goal state(s) within a trajectory using a given reward function.

LLM Initial Reward Generation

You are an ML engineer writing reward functions for RL training. Given a trajectory with marked goal states, create a Python reward function that can reproduce this behavior.

Requirements:

- Write self-contained Python 3.9 code
- Always return rewards >= 0
- Make the function generic enough to handle variations (different positions, orientations, etc.)
- Design for modularity - you might extend this reward later to handle multiple goal types
- Give 100.0 for the goal state and less than 1.0 (modulated for shaping) for all other states

Environment Details:
{env_code}, {import_instructions}, {state_description}

Trajectories
{expert_trajectories}

Key Instructions:

1. Analyze the trajectory to understand what constitutes success
2. Identify intermediate progress that should be rewarded
3. Create utility functions for reusable reward components

The code will be written to a file and then imported.
OUTPUT FORMAT:
Answer in a json format as follows:
'reasoning': Given the reason for your answer
'reward_class_code': Code for the Reward function class in the format:

```
# imports
<imports_here>
# utils functions
<utils functions here>
# reward function
class Reward:
   def __init__(self, extra_info=None):
     <code_here>

   def reward_fn(self, state):
     <code_here>

   def debug_fn(self, state):
     <code_here>
```

The Reward class will be initialized with the extra_info argument. Describe in the comments of the class the behaviour you are trying to reproduce.
reward_fn and debug_fn receive only state as argument. The debug_fn should return a string that will be printed and shown to you after calling reward_fn on each state. You can print internal class properties to help you debug the function. Extract any needed information from the state or store it in the class. The Reward class will be re-initialised at the beginning of each episode.

Prompt 2: Prompt to generate the initial set of rewards

Evolution Mutation Prompt

You are an ML engineer writing reward functions for RL training. Given a trajectory with marked goal states, create a Python reward function that can reproduce this behavior.

Requirements:

- Write self-contained Python 3.9 code
- Always return rewards >= 0
- Make the function generic enough to handle variations (different positions, orientations, etc.)
- Design for modularity - you might extend this reward later to handle multiple goal types
- Give 100.0 for the goal state and less than 1.0 (modulated for shaping) for all other states

Original Reward Code:
{{code}}

{{import_message}}
{{state_description}}

--
CRITICAL: Incorrect Trajectories
The reward function above FAILED on the following trajectories. It either assigned a high reward to a failed trajectory or failed to assign the highest reward to the correct goal state. The predicted rewards for each step are shown.
Change the reward function to fix these errors. The goal is to make the reward function correctly identify the goal state (or lack thereof) in these examples.

Key Instructions:

1. Analyze the trajectory to understand what constitutes success
2. Identify intermediate progress that should be rewarded
3. Create utility functions for reusable reward components
4. Implement goal switching logic using extra_info to determine which reward function to use
5. Reuse existing utilities where possible
6. Make sure the logic you write generalises to variations in `extra_info`

{incorrect_trajectories}

{expert_traj_str}
--

Now, provide the mutated version of the reward function that addresses these errors.

OUTPUT FORMAT:
Answer in a json format as follows:
'reasoning': Briefly explain the corrective change you made.
{REWARD_OUTPUT_FORMAT}
{REWARD_EXTRA_INFO}

Prompt 3: The prompt used for evolutionary mutation, providing feedback on incorrect trajectories.

Evolution Shaping Prompt

```
You are an ML engineer writing reward functions for RL training.
Given a trajectory with marked goal states, create a Python reward
function that can reproduce this behavior.  Requirements:
  • Write self-contained Python 3.9 code
  • Always return rewards >= 0
  • Make the function generic enough to handle variations
    (different positions, orientations, etc.)
  • Design for modularity - you might extend this reward later to
    handle multiple goal types
  • Give 100.0 for the goal state and less than 1.0 (modulated for
    shaping) for all other states
Original Reward Code:
{env_code}

{import_message}
{state_description}

--
CRITICAL: Incorrectly Shaped Trajectories
The reward function above is not shaped optimally on the following
trajectories.  This is an expert trajectory, so the reward should
monotonically increase from one state to the next.  The predicted
rewards for each step are shown.
Change the reward function to fix these errors.

{incorrect_expert_trajectories}
--

Now, provide the mutated version of the reward function that
addresses these errors.

OUTPUT FORMAT:
Answer in a json format as follows:
'reasoning':  Briefly explain the corrective change you made.
{REWARD_OUTPUT_FORMAT}
{REWARD_EXTRA_INFO}
```

Prompt 4: The prompt used for refining reward shaping based on expert trajectories.

A.10 LLM Usage Statement

We wish to disclose the role of LLMs in the preparation of this work to ensure transparency.

Manuscript Writing We employed LLMs to assist in the writing process. This included rephrasing sentences and paragraphs to enhance clarity and flow, and checking for grammatical errors and stylistic consistency. While LLMs helped refine the presentation of our ideas, all core arguments, scientific claims, and the overall structure of the paper were developed by the human authors.

Code Development and Debugging In the software development process, LLMs were used as a coding assistant. This involved generating specific utility functions based on detailed prompts, providing explanations for complex error messages, and suggesting alternative implementations for performance or readability improvements. The overall software architecture and core algorithms were designed and implemented by the human authors, who verified and tested all LLM-assisted code.