

GRACE: A Language Model Framework for Explainable Inverse Reinforcement Learning

Silvia Sapora*, Devon Hjelm, Alexander Toshev, Omar Attia, Bogdan Mazoure

Apple

Inverse Reinforcement Learning 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 **As CodE**), a method for using Large Language Models 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 validate GRACE on the BabyAl and AndroidWorld benchmarks, where it efficiently learns highly accurate rewards, even in complex, multi-task settings. 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.

Correspondence: Silvia Sapora: silvia.sapora@stats.ox.ac.uk; Bogdan Mazoure: bmazoure@apple.com

Date: October 3, 2025

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 can 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). Code is a particularly well-suited representation, because reward functions are often far simpler to express than the complex policies which maximize them (Ng & Russell, 2000; Cook, 1971; Godel, 1956). 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.

^{*}Work done while SS was an intern at Apple.

In this work, we address the question of how to efficiently infer rewards-as-code from expert demonstrations using LLMs. We propose an optimization procedure inspired by evolutionary search (Goldberg, 1989b; Eiben & Smith, 2003b; Salimans et al., 2017a; 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). We 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 or to 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). However, 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 that makes the expert's policy optimal over all alternatives (Ng & Russell, 2000). While related to our formulation, our representation (code) and our optimization strategy (evolutionary search) are fundamentally different. Subsequent works focused on learning policies directly, 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.

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:

$$J(\pi, r) = \mathbb{E}_{\tau \sim \pi, T} \left[\sum_{t=1}^{H} r(s_t) \right]$$
 (3.1)

where τ are trajectory unrolls of horizon H of the policy π in \mathcal{M} . An optimal agent can be learned by maximizing Equation (3.1) 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):

$$\min_{\pi} \max_{R} J(\pi_E, r) - J(\pi, r) \tag{3.2}$$

More recent IRL 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 (Salimans et al., 2017b; Eiben & Smith, 2003a; Goldberg, 1989a). 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 - Generating Rewards As CodE, an interpretable IRL framework that generates a reward function as executable Python code. Initially, an LLM analyzes expert and random trajectories to identify goal states (Phase 1) and generates a preliminary set of reward programs. 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). The best-performing reward function is used to train an RL agent. This agent then explores the environment, and the new trajectories it generates are used to expand the dataset, revealing new edge cases or failure modes (Phase 3). This loop continues, with the expanded dataset from Phase 3 being used in the next iteration of Phase 1 and 2, progressively improving the reward function. The overall process is illustrated in Figure 1 and detailed below and in Algorithm 1

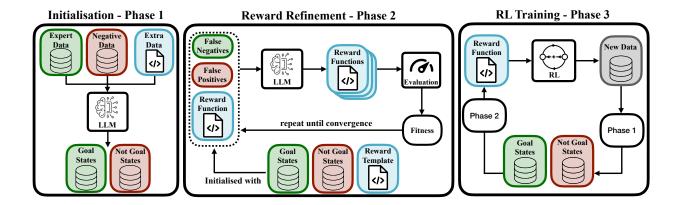


Figure 1 Overview of the GRACE framework. (a) The expert, negative and extra data is used to identify goal states. (b) The goal and non-goal states are used to generate reward functions through an evolutionary procedure. The rewards are iteratively refined by feeding the examples misclassified by the 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.

Phase 1: Goal States Identification 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 interchangeably a set of trajectories as well the set of all states from these trajectories.

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:

Goal states: The LLM analyzes the states from expert demonstrations to identify the subset of goal states $S_g \subseteq \mathcal{D}^+$ that solve the task - these are positive samples. All remaining non-goal states $S_{ng} = \{\mathcal{D}^+ \setminus S_g\} \cup \mathcal{D}^-$ are initially treated as negative samples.

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>
```

designed to assign high values to goal states S_g and low values to non goal ones 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:

$$f(r) = \mathbb{E}_{s \sim \mathcal{S}_a}[r(s)] - \mathbb{E}_{s \sim \mathcal{S}_{na}}[r(s)] \tag{3.3}$$

In practice, to normalize the fitness computation, we bound the reward signal. Any reward value greater than or equal to a predefined $r(s) \ge \tau$ 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:

$$m(r) = \text{LLM}(\text{source}(r), \text{info}, \text{prompt})$$
 (3.4)

In more detail, source(r) is the Python code for the reward. The 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 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: Active Data Collection via Reinforcement Learning The optimal reward r^* above is obtained by inspecting existing demonstrations. 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 S_{g^*} and non-goal states 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. 3.4 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. 3.3. 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 $s \in S_g$, 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. (3.2).

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:

Algorithm 1 GRACE: Generating Rewards As CodE

```
// Phase 2: Refinement via Evolution.
Inputs:
   \mathcal{D}^+: expert trajectories
                                                                                         function EvoSearch(\mathcal{R}, \mathcal{S}_a, \mathcal{S}_{na})
   \mathcal{D}^-: random trajectories
                                                                                               for k = 1 \dots K do
Parameters:
                                                                                                    Sample r \sim \exp(f(r)), r \in \mathcal{R}
                                                                                                    r' \leftarrow m(r) // See Eq. 3.4
   P: reward population size
   K: mutation steps
                                                                                                    if f(r') > \min_{r \in \mathcal{R}} f(r) then
   M number of generations
                                                                                                         r'' = \arg\min_{r \in \mathcal{R}} f(r)
                                                                                                          \mathcal{R} = \mathcal{R}/\{r''\} \cup \{r'\}
   N: RL budget
                                                                                                    end if
procedure GRACE(\mathcal{D}^+, \mathcal{D}^-)
                                                                                               end for
     // Phase 1: Initialization.
                                                                                               return \mathcal{R}
     S_g = \{ s \in D^+ \mid \text{LLM}(s, \text{goal\_prompt}) \}
                                                                                         end function
     S_{ng}^{\circ} = \mathcal{D}^+ \cup \mathcal{D}^- / \mathcal{S}_g
     \mathcal{R} = \{ \text{LLM}(S_n, S_{nq}, \text{reward prompt}) \}
                                                                                         // Phase 3: Trajectory expansion via RL.
                                                                                         function DataExpandRL(\mathcal{R})
     // Reward Refinement.
                                                                                               r^* \leftarrow \arg\max_{r \in \mathcal{R}} f(r)
     for i = 1 \dots M do
                                                                                               Train \pi_{r^*} with PPO under budget N
           \mathcal{R} \leftarrow \text{EvoSearch}(\mathcal{R}, \mathcal{S}_g, \mathcal{S}_{ng})
                                                                                               Collect new trajectories \mathcal{D}_{r^*}
          \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}
                                                                                               S_q = \{ s \in \mathcal{D}_{r^*} \mid \text{LLM}(s, \text{goal\_prompt}) \}
                                                                                               \vec{S}_{ng} = \mathcal{D}_{r^*}/\mathcal{S}_g
                                                                                               return S_q, S_{ng}
     end for
     return r^* = \arg\max_{r \in \mathcal{R}} f(r)
                                                                                         end function
end procedure
```

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?

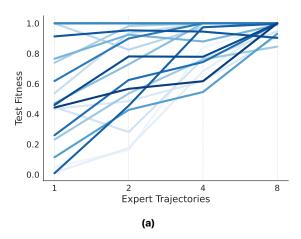
Interpretability and Multi-Task Efficacy: Does GRACE produce reward APIs that can be shared across tasks?

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. (3.3).

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).



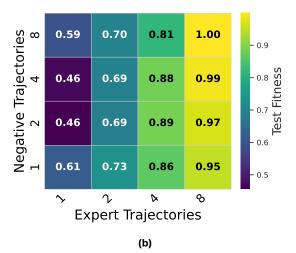


Figure 2 Fitness vs Number of Expert Trajectories. The fitness is computed on test dataset after obtaining maximum fitness on training data with corresponding number of expert and negative training trajectories. (a) Performance on all 20 BabyAI tasks. (b) Aggregate fitness across 20 BabyAI tasks.

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.

GRACE Parameters All parameters 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 singleand 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.

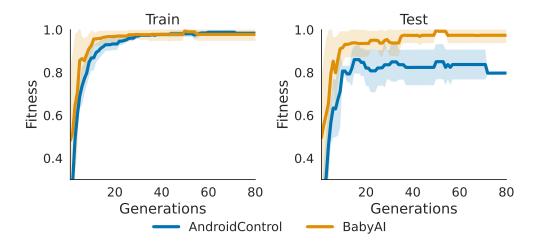


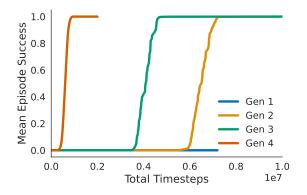
Figure 3 Fitness vs Number of generations. Evolution of train and test fitness across evolution generations, as defined by Algorithm 1, for BabyAI (multi-level settings) and AndroidControl (bottom) for "set alarm" task. We provide 8 expert trajectories and 8 negative trajectories for each task. Shading is standard deviation across 3 seeds.

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 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, GRACE consistently matches or outperforms GAIL across all tasks with lesser training data. On several tasks, GRACE matches Oracle PPO, 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.



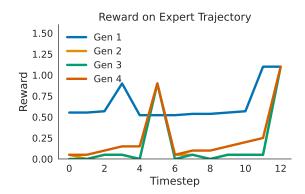


Figure 5 Shaping Using the default reward recovered by GRACE occasionally leads to failure in learning the correct behavior due to poor shaping. Through the targeted shaping in Phase 3, we significantly improve final performance and speed of learning.

PPO	GAIL	GRACE
1.00	1.00	1.00
1.00	0.35	1.00
0.31	0.15	0.32
0.21	0.00	0.26
1.00	0.92	1.00
1.00	0.98	1.00
1.00	0.37	1.00
0.09	0.01	0.09
0.79	0.20	0.35
0.95	0.31	0.92
	1.00 1.00 0.31 0.21 1.00 1.00 1.00 0.09 0.79	1.00 1.00 1.00 0.35 0.31 0.15 0.21 0.00 1.00 0.92 1.00 0.98 1.00 0.37 0.09 0.01 0.79 0.20

Table 1 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 Android-World Clock Tasks. Mean episode success over the 3 Android-World clock tasks: Clock-StopWatchPausedVerify, 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.

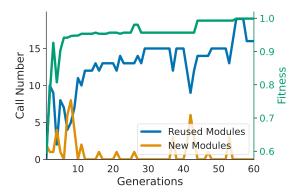




Figure 6 Module and function reuse across generations On the left, we show at each generation step the number of newly created modules and the number of existing and thus reused modules from prior rewards, contrasted with the fitness in the reward population. On the right, we show number of times a module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

5 Discussion

Limitations A key trade-off exists in Phase 3, where we allow the LLM to identify new goal states from trajectories generated by the learner agent. This presents a risk of introducing false positives: an inaccurate LLM could incorrectly label a failed state as a success, breaking the reward function. Conversely, when the LLM is accurate, this process allows the reward function to generalize beyond the initial expert demonstrations, preventing it from overfitting to a narrow set of examples and improving its robustness to novel states discovered during exploration.

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 generalisable 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.
- Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pp. 151–158, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 9781450374644. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.
- 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 J. E. Smith. Introduction to Evolutionary Computing. SpringerVerlag, 2003a. ISBN 3540401849.
- Agoston E. Eiben and James E. Smith. Introduction to Evolutionary Computing. Springer, 2003b.
- 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.
- 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.
- Kurt Godel. Letter to john von neumann, 1956. URL https://ecommons.cornell.edu/server/api/core/bitstreams/46aef9c4-288b-457d-ab3e-bb6cb1a4b88e/content.
- David E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989a. ISBN 0201157675.
- David E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989b.
- 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, Ngân Vũ, 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 Madry, 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, Michael Wang, Michael Wu, Michael Wu, Michael Wang, Michael Wu, Michael Wang, Michael Wu, Michael W

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.
- 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)*, 2017a.
- Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017b. URL https://arxiv.org/abs/1703.03864.
- 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.

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 $s \in S_g$ 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. (3.2)

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. 3.3 can be re-written in terms of the policy return akin to Eq. 1:

$$f(r) = \mathbb{E}_{s \sim \mathcal{S}_a}[r(s)] - \mathbb{E}_{s \sim \mathcal{S}_{na}}[r(s)] \tag{A.1}$$

$$= \mathbb{E}_{\tau \sim D^+, s \in \tau}[m(s)r(s)] - \mathbb{E}_{\tau \sim D^-, s \in \tau}[-m(s)r(s)]$$
(A.2)

$$= J(\pi_E, m \circ r) - J(\pi, -m \circ r) \tag{A.3}$$

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.

Phase 1 is responsible in defining the operator m. 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-40 (OpenAI et al., 2024), with the results presented in Table 2. 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 2 Model Accuracy Comparison

	gpt-4o w/		
Metric	Text	Images	Text and Images
Accuracy		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
${\bf GoToRedBlueBall}$	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
${\bf OpenRedBlueDoors}$	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 3 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 Baby AI 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 4
 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 5
 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

 $\textbf{Table 6} \ \ \textbf{Hyperparameters for GRACE Evolution}$

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]:
2
        if text is None:
            return None
        colour_words: Dict[str, int] = {
6
             "red": 0,
            "green": 1
            "blue": 2,
                        "purple": 3,
    "yellow": 3,
9
10
    "yellow": 4,
            "orange": 5,  # keep old mapping
11
    "grey": 5, # alias for the observed colour code in the trajectory
12
     "gray": 5,
13
14
        lower = text.lower()
15
16
        for word, code in colour_words.items():
            if word in text.lower(): lower:
17
                 return code
18
        return None
19
20
21
    def _parse_goal_type(text: Optional[str]) -> str:
22
        if text is None:
23
            return "key"
24
        txt = text.lower()
25
        if "ball" in txt:
26
            return "ball
27
        if "box" in txt:
28
29
    return "box"
        return "key"
30
```

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,
11
         "yellow": 4,
13
         "grey": 5,
"gray": 5, # US spelling
14
16
    }
    OBJECT2ID = {
         "empty": 0,
"wall": 1,
19
20
         "floor": 2,
         "door": 3,
         "key": 5,
"ball": 6,
23
         "box": 8,
         "agent": 10,
28
   # 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.
32  DIR2VEC: dict[int, Tuple[int, int]] ={
33 0: (1, 0), # south
     1: (0, 1), # east
35 2: (-1, 0), # north
3: (0, -1), # west
37 }
```

```
38
     def _parse_goal(extra_info: str ) -> Tuple[int, Optional[int]]:
39
      """Return *(object_id, colour_id)* parsed from *extra_info*."""
40
         if not extra\_info:
41
42
              raise ValueError("extra_info must specify the target, e.g. 'the red ball'.")
 43
         tokens = re.findall(r"[a-zA-Z]+", extra\_info.lower())
 45
         obj_id: Optional[int] = None
         col_id: Optional[int] = None
 46
         for tok in tokens:
 47
48
               if obj_id is None and tok in OBJECT2ID:
              if tok in COLOR2ID and col_id is None:
49
                   col_id = COLOR2ID[tok]
50
              if tok in OBJECT2ID and obj_id is None:
51
                  obj_id = OBJECT2ID[tok]
52
              if col_id is None and tok in COLOR2ID:
                   col id = COLOR2ID[tok]
54
         if obj_id is None:
55
             raise ValueError(
56
                  f"Could not parse target object from extra_info='{extra_info}'."
57
58
         return obj_id, col_id # colour may be None (wild-card)
59
60
61
     class Reward:
62
         """Success when definition (single-step, dense reward):
63
     100.0 - the **first** cell in front of the agent is *either*
64
      - on / adjacent (according to the
65
                         closest target object (Manhattan distance <= 1), OR</pre>
66
     - direction stored in the third observation channel) contains a
67
      matching target has disappeared from the observable grid (picked up).
68
69
     Shaping: r = 1 / (1+d) with d the Manhattan distance to the closest
70
     -still-visible target, clipped at 0 object.
     <1.0 - shaping reward 1/(d+1) otherwise.
72
      0.0 - if either the agent or (a matching) target is out of view. not visible.
73
74
      The implementation is modular so new goal
75
76
      types can be handled by extending the OBJECT/COLOR lookup tables or by
      replacing the *success predicate*.
77
78
79
80
          SUCCESS REWARD = 100.0
         def __init__(self, extra\_info: Optional[str] str = None):
81
              self.tgt_obj_id, self.tgt_col_id self._target_obj_id, self._target_colour_id = _parse_goal(extra_info)
82
83
84
         def __call__(self, state: np.ndarray) -> float: # enable direct call
     return self.reward_fn(state)
85
86
         def reward_fn(self, state: np.ndarray) -> float:
87
88
              """state: (H, W, 3) """
              agent_pos = self._find_agent(state)
89
              if agent_pos is None:
90
91
                  return 0.0
92
              # mask of all target objects still visible
93
              tgt_mask = (state[:, :, 0] == self.tgt_obj_id) & (
               state[:, :, 1] == self.tgt_col_id
95
96
97
              if not tgt_mask.any():
98
               # object gone -> picked up / carried
              return self.SUCCESS_REWARD
              # distance to the closest visible target
102
103
              tgt_positions = np.argwhere(tgt_mask)
```

```
dists = np.abs(tgt_positions - agent_pos).sum(axis=1)
104
105
106
               target_positions = self._find_targets(state)
               if target_positions.size == 0:
107
                # No matching target in view -> no shaping.
108
109
               return 0.0
112
               # Success predicate - target must be directly in front of the agent.
113
114
               if self._is_target_in_front(agent_pos, state):
115
               return 100.0
116
117
               # Shaping: inverse Manhattan distance (< 1.0) to the *nearest* target.
118
119
              dists = np.abs(target_positions - agent_pos).sum(axis=1)
120
              min_dist = int(dists.min())
121
              if min_dist <= 1:</pre>
              -return_self.SUCCESS_REWARD
123
124
              return 1.0 / (1.0 + min_dist)
126
127
         @staticmethod
128
         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
129
                    None.""" *None* if absent."""
              locs = np.argwhere(state[:, :, 0] == OBJECT2ID["agent"])
130
              if locs.size == 0:
                  return None
              return locs[0]
134
          def _find_targets(self, state: np.ndarray) -> np.ndarray:
135
      """Return an (N, 2) array of row/col positions of matching targets."""
      obj_mask = state[:, :, 0] == self._target_obj_id
137
      if self._target_colour_id is not None:
      col_mask = state[:, :, 1] == self._target_colour_id
139
      mask = obj_mask & col_mask
      else:
141
      mask = obj_mask
142
143
      return np.argwhere(mask)
144
      def _is_target_in_front(self, agent_pos: np.ndarray, state: np.ndarray) -> bool:
145
      """Return *True* iff the cell directly in front of the agent matches target."""
      row. col = agent pos
147
     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 → cannot be success.
if not (0 \le f_row < state.shape[0]) and 0 \le f_rol < state.shape[1]):
     return False
# Check object id
if state[f_row, f_col, 0] != self._target_obj_id:
# Check colour if colour was specified.
161 if (
      self._target_colour_id is not None
      and state[f_row, f_col, 1] != self._target_colour_id
164 ):
```



 $\textbf{Figure 8} \ \ \text{Example of code evolution across many generations}.$

A.8 Generated Rewards

```
1 # -----
2 #
                         IMPORTS
3 # -----
  import json
5 import math
6 import re
7 from typing import Callable, List, Optional, Set, Tuple
9 # -----
            GENERIC & NORMALISATION HELPERS
11 # -----
13
  def _contains_any(text: str, keywords) -> bool:
14
      text_l = text.lower()
      return any(k.lower() in text_l for k in keywords)
16
17
18
  def _has_stopwatch(text: str) -> bool:
19
20
    t = text.lower()
      return any(p in t for p in ("stopwatch", "stop watch", "stop-watch"))
21
22
23
# ------ Tab-selection helpers ------
25
26
27 def _tab_selected(state: str, label: str) -> bool:
    pattern = (
28
         rf'"(content_description|text)"\s*:\s*"{label}"[^\n]*?"is_selected"\s*:\s*true'
29
30
      return bool(re.search(pattern, state, re.I))
31
32
33
34 def _alarm_tab_selected(state: str) -> bool:
      return _tab_selected(state, "Alarm") or _tab_selected(state, "Alarms")
35
36
37
38 def _timer_tab_selected(state: str) -> bool:
      return _tab_selected(state, "Timer")
39
40
41
42 def _stopwatch_tab_selected(state: str) -> bool:
      return _tab_selected(state, "Stopwatch")
43
44
45
   def _clock_tab_selected(state: str) -> bool:
46
     return _tab_selected(state, "Clock")
47
48
49
50 # ----- Text normalisation helper -----
51
52
53 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)
54
55
56
      return txt2
57
58
59 # -----
                   TIMER / DURATION PARSING
60 #
61 # -----
62
  def _parse_requested_time(text: str) -> int:
65
      text = text.replace("-",
      hours = minutes = seconds = 0
66
      for patt, mult in (
    (r"(\d+)\s*hour", 3600),
67
68
          (r"(\d+)\s*minute", 60),
(r"(\d+)\s*second", 1),
69
70
71
72
          m = re.search(patt, text, re.I)
73
          if m:
74
              val = int(m.group(1)) * mult
              if mult == 3600:
75
                  hours = val // 3600
76
              elif mult == 60:
77
78
                 minutes = val // 60
              else:
79
           seconds = val
80
```

```
if hours == minutes == seconds == 0:
81
             m = re.search(r''(\d+)\s^*-?\s^*min'', text, re.I)
82
             if m:
83
                 minutes = int(m.group(1))
84
85
             else:
                 m = re.search(r''(\d+)'', text)
86
                  if m:
87
                     minutes = int(m.group(1))
88
        total = hours * 3600 + minutes * 60 + seconds
89
        return total if total > 0 else 60
90
91
92
93 # --
                     ADDITIONAL HELPERS
94 #
95 #
96
97
    def parse adjust timer amount(instr: str) -> Optional[int]:
98
99
        instr_l = instr.lower()
        verb = r"(?:add|increase|extend|plus|up|extra|more|additional)"
100
        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] = []
104
105
        for pat in (pat1, pat2):
             for m in pat.finditer(instr_l):
106
107
                 num = int(m.group(1))
108
                 u = m.group(2)
109
                  if u.startswith("hour"):
110
                      seconds.append(num * 3600)
                  elif u.startswith("minute"):
111
                      seconds.append(num * 60)
112
                  else:
113
                      seconds.append(num)
114
        if seconds:
115
             return max(1, min(seconds))
116
        return None
118
119
    def _parse_alarm_time(instr: str) -> Tuple[int, int, Optional[str]]:
120
        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_1)
124
125
             h, minute, ap = int(m.group(1)), int(m.group(2)), m.group(3)
126
        else:
             m = re.search(r"\b(\d{1,2})\s*(am|pm)\b", instr_1)
128
129
                 h, minute, ap = int(m.group(1)), 0, m.group(2)
             else:
130
131
                 return 7, 0, "am"
132
        if ap:
             ap = ap.lower()
             if ap == "pm" and h != 12:
134
                 h += 12
135
             if ap == "am" and h == 12:
136
                 h = 0
        return h % 24, minute, ap
138
139
140
    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)
141
142
        if m:
143
             minutes = int(m.group(1))
144
             seconds = int(m.group(2))
145
             return (0, minutes, seconds)
146
147
        m = re.search(r"(\d+)h\s*(\d+)m\s*(\d+)s", state, re.IGNORECASE)
148
149
        if m:
            hours = int(m.group(1))
150
             minutes = int(m.group(2))
151
             seconds = int(m.group(3))
             return (hours, minutes, seconds)
154
        # Case 3: "MM:SS" format, ensuring it's not part of a timestamp (like 12:30 PM)
        \label{lem:continuous} \begin{tabular}{ll} for $mm$_match in $re.finditer(r"(\d\{1,2\}):(\d\{2\})(?!\s*[AaPp][Mm])", state): \\ \end{tabular}
156
157
             mm, ss = int(mm_match.group(1)), int(mm_match.group(2))
158
             if not (0 <= ss < 60):</pre>
159
                 continue
             context = state[mm_match.end() : mm_match.end() + 80].lower()
if "minute" in context or "timer" in context or "remaining" in context:
160
161
             return (0, mm, ss)
162
```

```
163
        if not _timer_tab_selected(state):
164
            return None
165
166
        tokens = re.findall(r'"text"\s*:\s*"([^"]+)"', state)
167
        tokens = [t.strip() for t in tokens]
168
169
        for i in range(len(tokens) - 4):
170
171
            if (
                 re.fullmatch(r'' \setminus d\{1,2\}'', tokens[i])
172
                 and tokens[i + 1] ==
                 and re.fullmatch(r"\d{2}", tokens[i + 2])
174
                 and tokens[i + 3] == '
175
                 and re.fullmatch(r"\d{2}", tokens[i + 4])
176
            ):
177
                 h = int(tokens[i])
178
179
                 m_val = int(tokens[i + 2])
                 s = int(tokens[i + 4])
if 0 <= m_val < 60 and 0 <= s < 60:
180
181
182
                      return (h, m_val, s)
183
        for i in range(len(tokens) - 2):
184
185
             if (
                 re.fullmatch(r"\d{1,2}", tokens[i])
and tokens[i + 1] == ":"
186
187
                 and re.fullmatch(r'' \setminus d\{2\}'', tokens[i + 2])
188
189
190
                 m_val = int(tokens[i])
191
                 s_val = int(tokens[i + 2])
192
                 if 0 <= s_val < 60:
193
                     return (0, m_val, s_val)
194
195
        return None
196
197
    def _extract_timer_value(state: str) -> int:
199
        timer_components = _extract_timer_components(state)
200
        if timer_components:
201
             hh, mm, ss = timer_components
             return int(hh) * 3600 + int(mm) * 60 + int(ss)
202
203
            return None
204
205
206
207 # --- UI helpers -----
208
209
210 def _button_visible(state: str, label: str) -> bool:
211
       return bool(
           re.search(rf'"(content_description|text)"\s*:\s*"{label}"', state, re.I)
212
213
214
215
216 def _timer_screen_visible(state: str) -> bool:
        if _timer_tab_selected(state):
217
            return True
218
        s = state.lower()
219
        return "remaining" in s or "minutes timer" in s
220
221
222
def _is_timer_running(state: str) -> bool:
return _button_visible(state, "Pause")
225
226
227 def _timer_keypad_mode(state: str) -> bool:
228 return bool(re.search(r"\b\d{1,2}h\s*\d{1,2}m\s*\d{1,2}s\b", state))
229
230
231 def _is_timer_paused(state: str) -> bool:
        if _timer_keypad_mode(state):
232
             return False
233
        if _button_visible(state, "Start") and not _button_visible(state, "Pause"):
234
235
             return True
        if not _timer_screen_visible(state):
236
            return False
237
238
        s = state.lower()
        return "timer paused" in s or ("paused" in s and "timer" in s)
239
240
241
242 def _timer_keypad_zero(state: str) -> bool:
243     if not all(
244         re.search(rf'"text"\s*:\s*"{lbl}"', state, re.I)
```

```
for lbl in ("hour", "min", "sec")
245
246
247
        return len(re.findall(r'"text"\s*:\s*"0{2}"', state)) >= 3
248
249
250
251 def _timer_deleted(state: str) -> bool:
       s = state.lower()
252
       if "no timers" in s:
253
           return True
254
        val = _extract_timer_value(state)
255
       if val == 0 and not _is_timer_running(state):
256
           return True
257
       return _timer_keypad_zero(state)
258
259
260
def _stopwatch_running(state: str) -> bool:
262
       return (
            _button_visible(state, "Pause")
263
           or _button_visible(state, "Stop")
or "stopwatch running" in state.lower()
264
265
266
267
268
269 def _stopwatch_time_zero(state: str) -> bool:
       if re.search(r"b0{1,2}(?::0{2}){1,3}\b(?!:\d{2})", state):
270
271
           return True
       nums = re.findall(r'"text"\s*:\s*"(\d{2})"', state)
return bool(nums) and all(n == "00" for n in nums)
272
273
274
275
276 def _timer_paused_notification(state: str) -> bool:
       return bool(
277
            re.search(r"the\s+clock\s+notification:\s*timer", state, re.I)
278
279
            or re.search(r"timer\s+paused", state, re.I)
280
281
   def _alarm_context_present(state: str) -> bool:
283
       return _alarm_tab_selected(state) or bool(re.search(r"\balarm\b", state, re.I))
285
    def _parse_new_timer_label(instr_l: str) -> str:
287
        for kw in (" as ", " named ", " called ", " name "):
    if kw in instr_l:
288
289
               part = instr_l.split(kw, 1)[1]
290
291
       return part.strip()
return ""
                part = re.split(r"[.,;]|\bfor\b|\btimer\b", part, flags=re.I)[0]
292
293
294
295
296 def _timer_label_present(state: str, label: str) -> bool:
       if not label:
297
           return False
298
        return bool(
299
          re.search(
300
               rf'"(text|content_description)"\s*:\s*"{re.escape(label)}"', state, re.I
301
302
303
304
305
   def _safe_json_dumps(obj) -> str:
306
307
           return json.dumps(obj, ensure_ascii=False)
308
        except Exception:
309
            return json.dumps({"error": "debug-serialization failed"})
310
311
312
313 def _any_alarm_present(state: str) -> bool:
       sl = state.lower()
if "alarm set" in sl:
314
315
           return True
316
       317
318
            return True
       return False
319
320
321
322 def _is_alarm_deleted(state: str) -> bool:
323
       s = state.lower()
324
       return anv(
      re.search(p, s)
for p in (
325
326
```

```
r"alarm (deleted|removed|dismissed)",
327
                                  r"\bno (active )?alarms?\b",
328
                                  r"tap here to create an alarm",
329
                                  r"alarm deleted",
330
                        )
331
332
333
334
       def _snooze_completed(state: str) -> bool:
335
                s low = state.lower()
336
                if "alarm snoozed" in s_low:
337
                         return True
338
                 if re.search(r"snoozed\s+for\s+\d+", s_low):
339
                         return True
340
                if re.search(r"\bsnooz(ing|ed)\b". s low):
341
                         return True
342
                if "select snooze duration" in s_low:
    return True
343
344
                return False
345
346
347
348 def _rename_dialog_open(state: str) -> bool:
349
                 s = state.lower()
                if "enter timer name" in s:
350
351
                         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)
352
353
                return bool(has_buttons and has_edit)
354
355
356
357
       def _detect_alarm_time(state: str) -> bool:
358
                return bool(re.search(r"\b\d{1,2}\s*:\s*\d{2}(?:\s*[ap]m)?\b", state, re.I))
359
360
361
        def _selected_weekdays(state: str) -> Set[str]:
                selected = set()
363
                 for key, full, abbrev in (
                         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"),
364
365
366
367
369
370
371
372
                         patt = rf'("content\_description"|"text") \\ \xspace* $$ rf'("content\_descript
                  is_checked")\s*:\s*true
                         if re.search(patt, state, re.I):
373
374
                                 selected.add(key)
375
                return selected
376
377
       def _alarm_time_present(state: str, hour24: int, minute: int, ap: Optional[str]):
378
                s = state.lower().replace("\u200a", "")
379
                h12 = hour24 \% 12  or 12
380
                patterns = [rf'' b0*{h12}:{minute:02d}\s*(?:am|pm)?\b"]
381
                if minute == 0:
382
                         patterns.append(rf'' b0*{h12} s*(?:am|pm)b'')
383
                patterns.append(rf"\b0*{hour24}:{minute:02d}\b")
384
                for p in patterns:
385
                         if re.search(p, s):
386
                                 if ap and not re.search(rf"{ap}\b", s):
387
                                           continue
388
                                  return True
389
                return False
390
391
392
393 # ----- NEW HELPER -----
394
395
       def _day_toggle_buttons_visible(state: str) -> bool:
396
                """Detect if the row of weekday toggle buttons is visible."""
matches = re.findall(r'"text"\s*:\s*"(S|M|T|W|F)"', state)
397
398
399
                unique = set(matches)
                return len(matches) >= 5 and len(unique.intersection({"M", "T", "W", "F"})) >= 3
400
401
402
403 # -----
                                                                REWARD CLASS
404 #
405 #
406
407
```

```
408 class Reward:
         """Dense reward function for Google Clock tasks."""
409
410
         \_SHAPING\_INC = 0.3
411
        _ADJ_INC_THRESHOLD = 10
412
413
414
                                   INIT
415
416
        def __init__(self, extra_info: Optional[str] = None):
417
             self.raw_instr: str = extra_info or
418
             self.instruction: str = self.raw_instr.lower()
419
             self.instruction_norm_full = _normalize_time_text(self.raw_instr)
420
             self.instruction_norm = self.instruction_norm_full.lower()
421
422
             # Task detection
423
             self.task_type = self._infer_task()
424
425
             # Stopwatch flags
426
             self.restart_mode = False
427
             self._reset_seen = False
428
429
             # Goal parsing / bookkeeping
430
             self.goal_seconds = 0
self.goal_label = ""
431
432
             self.goal\_hour24 = 0
433
434
             self.goal_minute = 0
             self.goal\_hms = (0, 0, 0)
435
436
             self.goal_ap: Optional[str] = None
437
             self.city_keyword =
             self.city_keywords: List[str] = []
438
439
             self.recurrence_days: Set[str] = set()
440
             self.alarm_any_time = False
441
442
             # Timer-adjust bookkeeping
             self.initial_timer_val: Optional[int] = None
443
444
             self.prev_timer_val: Optional[int] = None
445
             self.max_timer_val: Optional[int] = None
446
             self.increments = 0
             self.needed_increments = 0
447
             self._countdown_seen = False
448
             # Alarm creation flag
450
451
             self._alarm_creation_seen = False
452
453
             # delete-alarm bookkeeping
454
             self._alarm_present_ever = False
455
456
             # adjust-alarm bookkeeping
             self.orig_hour24 = 0
457
             self.orig_minute = 0
458
             self._orig_seen = False
459
460
             # pause-timer stability tracking
461
             self._prev_timer_val_for_pause: Optional[int] = None
462
             self._same_val_steps: int = 0
463
464
             # snooze-specific
465
             self._snooze_dialog_seen = False
466
467
             # Generic bookkeeping
468
             self.goal_achieved = False
469
             self._best_level = 0
470
             self._t = 0
471
             self._confirm_goal_seen = False
472
473
             # Map tasks to progress-functions
474
             self._progress_fns: dict[str, Callable[[str], int]] = {
   "reset_stopwatch": self._pl_reset_stopwatch,
   "restart_stopwatch": self._pl_restart_stopwatch,
475
476
477
                 "start_stopwatch": self._pl_start_stopwatch,
"pause_stopwatch": self._pl_pause_stopwatch,
478
479
                 480
481
482
                 "add_city": self._pl_add_city,
"set_alarm": self._pl_set_alarm,
483
484
                 "adjust_alarm": self._pl_adjust_alarm,
"rename_timer": self._pl_rename_timer,
485
486
487
             }
488
489
           # Goal-specific parsing / bookkeeping
```

```
if self.task_type == "set_timer" or self.task_type == "run_timer":
490
                 self.goal_seconds = _parse_requested_time(self.instruction)
h = self.goal_seconds // 3600
491
492
                 rem = self.goal_seconds % 3600
493
                 m = rem // 60
494
                 s = rem \% 60
495
                 self.goal_hms = (h, m, s)
496
             if self.task_type == "adjust_timer":
497
                 inc_secs = _parse_adjust_timer_amount(
498
                      self.instruction_norm_full
499
                 ) or _parse_requested_time(self.instruction)
500
                 self.goal_seconds = max(1, inc_secs)
501
                 self.needed_increments = max(1, math.ceil(self.goal_seconds / 60))
502
             if self.task_type == "rename_timer":
503
                 self.goal_seconds = _parse_requested_time(self.instruction)
504
             self.goal_label = _parse_new_timer_label(self.instruction)
if self.task_type == "set_alarm":
505
506
                 explicit = re.search(
507
                     r'' d{1,2}(:d{2})? s*(am|pm)", self.instruction_norm_full, re.I
508
509
510
                 if explicit:
511
                      self.alarm_any_time = False
512
                      self._parse_alarm_goal_time()
513
                 else:
514
                      self.alarm_any_time = True
                 self.recurrence_days = self._parse_recurrence_days(self.instruction_norm)
515
             if self.task_type == "adjust_alarm":
    self.goal_hour24, self.goal_minute = self._parse_adjusted_alarm()
516
517
518
                 self.goal_ap = None
519
                 self.orig_hour24, self.orig_minute, _ = _parse_alarm_time(
520
                      self.instruction_norm_full
521
             if self.task_type == "add_city":
522
                 self.city_keyword = self._parse_city_name(self.instruction) or "italy"
523
                 self.city_keywords = [self.city_keyword]
524
                  first = self.city_keyword.split()[0] if self.city_keyword else ""
526
                 if first and first not in self.city_keywords:
527
                      self.city_keywords.append(first)
             if self.task_type == "reset_stopwatch'
528
                 if re.search(r"\brestart\b", self.instruction) or re.search(
    r"start\s+(?:over|again)", self.instruction
530
531
                      self.restart_mode = True
533
534
                           PUBLIC API
536
        def reward_fn(self, state: str) -> float:
537
538
             self._t += 1
             if self.task_type == "set_alarm":
539
540
                 self._update_alarm_creation_seen(state)
             if self.goal_achieved:
541
                 return 100.0
542
             if self.task_type in self._progress_fns:
543
                 return self._reward_from_progress(self._progress_fns[self.task_type], state)
544
             if self.task_type == "set_timer" or self.task_type == "run_timer":
545
                 return self._reward_timer(state, self.task_type == "set_timer")
546
             if self.task_type == "adjust_timer":
547
                 return self._reward_adjust_timer(state)
548
             if self.task_type == "snooze_alarm";
549
                 return self._reward_snooze(state)
550
             return 0.0
551
552
        def debug_fn(self, state: str) -> str:
553
             dbg = {
   "step": self._t,
554
555
                 "task_type": self.task_type,
556
                 "goal_achieved": self.goal_achieved,
"best_level": self._best_level,
557
558
559
             if self.task_type in {"set_timer", "run_timer", "adjust_timer"}:
560
561
                 dbg.update(
562
                      {
                            goal_seconds": self.goal_seconds,
563
                           "increments": self.increments,
564
                           "countdown_seen": self._countdown_seen,
565
566
                      }
567
             if self.task_type == "rename_timer":
568
             dbg["goal_label"] = self.goal_label
if self.task_type == "snooze_alarm":
    dbg["dialog_seen"] = self._snooze_dialog_seen
569
571
```

```
return _safe_json_dumps(dbg)
572
573
574
                        TASK INFERENCE
575
576
        def _infer_task(self) -> str:
577
             instr = self.instruction
578
             has_sw = _has_stopwatch(instr)
579
580
             if has_sw and _contains_any(instr, ["pause", "stop"]):
581
                 return "pause_stopwatch
582
             elif has_sw and _contains_any(
   instr, ["restart", "start over", "start again", "begin again"]
583
584
             ):
585
                  return "restart_stopwatch"
586
             if has_sw and _contains_any(instr, ["reset", "zero", "set to zero", "clear"]):
587
                 return "reset_stopwatch"
588
             if has_sw:
589
                 return "start_stopwatch"
590
591
592
             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"])
593
594
595
596
             ):
                  return "add city"
597
598
             if "timer" in instr:
599
                 if _contains_any(instr, ["delete", "remove", "clear"]):
600
601
                      return "delete_timer
                  if _contains_any(instr, ["pause", "stop", "cancel"]):
602
603
                      return "pause_timer'
604
                  if _contains_any(instr, ["rename", "name", "called", "label"]):
605
                      return "rename_timer"
606
                  if re.search(
                     r"\badd\b[^\n]*?\b\d+\s*(?:hour|minute|second)s?\s+timer", instr
607
608
609
                      dont_start_req = bool(
610
                           re.search(
                               r"(?:\b(?:don'?t|do\s+not)\s+(?:start|run)\b)"
611
                               r"|(?:\bwithout\s+starting\b)'
612
                               r''|(?:\b(?:but|and)\s+don'?t\s+start\b)"
613
                               r'' | (?:\bleave\s+it\s+paused\b)"
614
                               r" | (?:\bkeep\s+it\s+paused\b)",
615
616
                               instr.
617
                          )
618
                      if dont_start_req:
619
620
                          return "set_timer"
621
                           return "run_timer"
622
                  if _contains_any(instr, ["increase", "extend", "more", "up"]):
623
                      return "adjust_timer"
624
                  if re.search(
625
                      r"\badd\b[^\n]*?\b(minutes?|hours?|seconds?)\b[^\n]*?\bto\b[^\n]*?\btimer\b",
626
                      instr,
627
628
                      return "adjust_timer"
629
                 return "run_timer"
630
631
             if "snooze" in instr:
    return "snooze_alarm"
632
633
             if _contains_any(instr, ["delete", "remove"]) and "alarm" in instr:
634
                  return "delete_alarm"
635
             if "alarm" in instr and _contains_any(
636
                 instr.
637
638
                  Γ
                      "delay"
639
                      "resched",
640
                      "push",
641
                      "move",
642
                      "change",
643
                      "shift",
"defer",
644
645
                      "later",
646
                      "increase",
647
648
                 ],
649
             ):
             return "adjust_alarm"
if "alarm" in instr:
650
651
                 return "set_alarm"
652
653
```

```
if _contains_any(
654
                instr, ["add", "timezone", "time zone", "city", "world clock"]
655
656
                return "add_city"
657
            return "none"
658
659
        def _update_alarm_creation_seen(self, state: str):
660
            s = state.lower()
661
            if any(kw in s for kw in ("add alarm", "alarm time", "select time")):
662
                self._alarm_creation_seen = True
663
664
665
                  GENERIC reward helpers
666
667
        def _reward_from_progress(self, fn: Callable[[str], int], state: str) -> float:
668
            lvl = fn(state)
669
            if self.task_type == "set_alarm":
670
671
                if 1v1 >= 3:
                    if self._alarm_creation_seen:
672
                         self.goal_achieved = True
673
674
                         return 100.0
675
                     if self._confirm_goal_seen or self._best_level >= 2:
676
                         self.goal_achieved = True
                         return 100.0
677
678
                     self._confirm_goal_seen = True
679
                     self._best_level = max(self._best_level, 2)
680
                    return 0.99
                self._confirm_goal_seen = False
681
682
            if 1v1 >= 3:
683
                self.goal_achieved = True
684
                return 100.0
            if lvl > self._best_level:
685
                inc = (lvl - self._best_level) * self._SHAPING_INC
self._best_level = lvl
686
687
688
                return min(inc, 0.99)
            return 0.0
690
691
                          TIMER-specific dense reward
692
693
        def _reward_timer(self, state: str, start_req: bool) -> float:
694
            reward = 0.0
695
            if _timer_tab_selected(state):
696
697
                reward += 0.2
            current_val = _extract_timer_components(state)
if current_val is None:
698
699
700
                return min(reward, 0.99)
            cur_hh, cur_mm, cur_ss = current_val
701
            current_digit_string = f"{cur_hh:02d}{cur_mm:02d}{cur_ss:02d}".lstrip("0")
702
            if current_digit_string == "":
703
                current_digit_string = "0"
704
            goal_digit_string = f"{self.goal_hms[0]:02d}{self.goal_hms[1]:02d}{self.goal_hms[2]:02d}".lstrip(
705
706
707
            if goal_digit_string == "":
708
                goal_digit_string = "0"
709
            running = _is_timer_running(state)
710
            if current_digit_string == goal_digit_string and running:
711
                if start_req and running:
712
                    self.goal_achieved = True
713
                     return 100.0
714
715
                if not start_req and not running:
                    self.goal_achieved = True
716
                    return 100.0
717
            matching_digits = 0
718
            for i in range(0, min(len(current_digit_string), len(goal_digit_string))):
719
                if goal_digit_string[i] == current_digit_string[i]:
720
721
                    matching_digits += 1
722
                else:
                    # Stop counting as soon as a mismatch occurs
723
724
                    break
            reward += (matching_digits / len(goal_digit_string)) * 0.7
725
726
            return min(reward, 0.99)
727
728
729
                   Other dense rewards (adjust_timer, snooze)
730
731
        def _reward_adjust_timer(self, state: str) -> float:
732
            reward = 0.0
733
            if _timer_screen_visible(state):
734
               reward += 0.2
735
          current_val = _extract_timer_value(state)
```

```
if current_val is None:
736
                 return min(reward, 0.99)
737
             if self.initial_timer_val is None:
738
                 self.initial_timer_val = self.prev_timer_val = self.max_timer_val = (
739
                     current val
740
741
                 return min(reward, 0.99)
742
            if current_val > (self.max_timer_val or 0):
743
            self.max_timer_val = current_val
diff_step = current_val - (self.prev_timer_val or current_val)
744
745
             if diff_step > self._ADJ_INC_THRESHOLD:
746
                 self.increments += max(1, int(round(diff_step / 60.0)))
747
             elif diff_step < -1:</pre>
748
                 self._countdown_seen = True
749
             self.prev_timer_val = current_val
750
             net_increase_max = (self.max_timer_val or current_val) - self.initial_timer_val
751
             fraction_by_inc = self.increments / max(1, self.needed_increments)
752
             fraction_by_delta = net_increase_max / max(1, self-goal_seconds)
progress_fraction = min(1.0, max(fraction_by_inc, fraction_by_delta))
753
754
             reward += 0.8 * progress_fraction
755
             tol = max(2, int(self.goal_seconds * 0.05))
756
             goal_reached_primary = (
    self.increments >= self.needed_increments
757
758
759
                 or net_increase_max >= self.goal_seconds - tol
760
761
             committed = (
762
                 _is_timer_running(state) or _is_timer_paused(state) or self._countdown_seen
763
764
             keypad = _timer_keypad_mode(state)
765
             secondary_success = (
766
                 not goal_reached_primary
                 and net_increase_max >= 0.4 * self.goal_seconds
767
                 and self.increments >= 1
768
                 and self._countdown_seen
769
770
                 and committed
                 and not keypad
771
773
             if (goal_reached_primary or secondary_success) and committed and not keypad:
                 self.goal_achieved = True
774
                 return 100.0
775
             return min(reward, 0.99)
777
778
        def _reward_snooze(self, state: str) -> float:
779
             s_low = state.lower()
             if "select snooze duration" in s_low:
780
781
                 self._snooze_dialog_seen = True
782
             classic_done = (
                 "alarm snoozed" in s_low
783
                 or bool(re.search(r"snoozed\s+for\s+\d+", s_low))
or bool(re.search(r"\bsnooz(ing|ed)\b", s_low))
784
785
786
787
            row_done = (
                 self._snooze_dialog_seen
788
                 and "select snooze duration" not in s_low and "snooze" in s_low
789
790
                 and bool(re.search(r"\b\d+\s+minutes?\b", s_low))
791
             if classic_done or row_done:
793
                 self.goal_achieved = True
794
795
                 return 100.0
             reward = 0.0
796
             if _alarm_tab_selected(state):
797
                 reward += 0.2
798
             if re.search(r'"(content_description|text)"\s*:\s*"snooze"', state, re.I):
799
                 reward += 0.3
800
             if "select snooze duration" in s_low:
801
                 reward += 0.2
802
803
             return min(reward, 0.99)
804
805
                 Progress-level helpers (stopwatch/timer/alarm)
806
807
        def _pl_reset_stopwatch(self, state: str) -> int:
808
809
             if self.restart mode:
                 if _stopwatch_running(state) and self._reset_seen:
810
811
                      return 3
812
                 if _stopwatch_time_zero(state):
813
                     self._reset_seen = True
814
                      return 2
                 if _button_visible(state, "Reset") and (
815
                     _stopwatch_tab_selected(state) or "stopwatch" in state.lower()
816
817
```

```
return 1
818
                return 0
819
            if _stopwatch_time_zero(state):
820
821
                return 3
            if _button_visible(state, "Reset") and (
    _stopwatch_tab_selected(state) or "stopwatch" in state.lower()
822
823
824
                return 2
825
            if _stopwatch_tab_selected(state):
826
827
                return 1
            return 0
828
829
        def _pl_pause_stopwatch(self, state: str) -> int:
830
            if not _stopwatch_running(state):
831
                return 3
832
            if _stopwatch_tab_selected(state):
833
834
                return 1
            return 0
835
836
837
        def _pl_restart_stopwatch(self, state: str) -> int:
838
            running = _stopwatch_running(state)
            at_zero = _stopwatch_time_zero(state)
839
840
            if running and self._reset_seen:
                return 3
841
842
            if at_zero:
843
                self._reset_seen = True
844
                return 2
845
            if _stopwatch_tab_selected(state):
846
                return 1
847
            return 0
848
849
        def _pl_start_stopwatch(self, state: str) -> int:
850
            if _stopwatch_running(state):
851
                return 3
852
            if "stopwatch" in state.lower() or _stopwatch_tab_selected(state):
                return 2
854
            if _contains_any(state.lower(), ["the clock", '"clock"', "alarms", "timer"]):
855
856
            return 0
        def _pl_pause_timer(self, state: str) -> int:
858
            if _is_timer_paused(state):
859
                return 3
860
861
            current_val = _extract_timer_value(state)
            if current_val is not None:
862
863
                if self._prev_timer_val_for_pause == current_val:
864
                     self._same_val_steps += 1
865
866
                     self._same_val_steps = 0
                self._prev_timer_val_for_pause = current_val
867
            else:
868
                self._same_val_steps = 0
869
            stable_and_visible = (
870
871
                _timer_tab_selected(state)
                 and current_val is not None
872
                and self._same_val_steps >= 1
873
                and not _is_timer_running(state)
874
875
            if stable_and_visible:
876
877
                return 3
            if _timer_paused_notification(state) and _timer_tab_selected(state):
878
879
                 return 3
            if _timer_paused_notification(state):
880
881
                 return 2
            if _is_timer_running(state):
882
883
                 return 2
            if _timer_tab_selected(state):
884
885
                return 1
886
            return 0
887
        def _pl_delete_timer(self, state: str) -> int:
888
889
            if _timer_deleted(state):
890
                return 3
            if _contains_any(
891
                state.lower(), ["delete", "remove", "clear", "backspace", "cancel"]
892
893
894
                return 2
895
            if _timer_tab_selected(state):
896
                return 1
897
            return 0
       def _pl_delete_alarm(self, state: str) -> int:
899
```

```
s_low = state.lower()
900
            had_alarm_before = self._alarm_present_ever
901
            alarm_now = _any_alarm_present(state) or _detect_alarm_time(state)
902
            if alarm_now:
903
                self._alarm_present_ever = True
904
            if _is_alarm_deleted(state) and had_alarm_before:
905
                 return 3
906
            if " delete" in s_low or re.search(r"trash|remove", s_low):
907
                 return 2
908
            if alarm now:
909
910
                return 1
            return 0
911
912
        def _pl_add_city(self, state: str) -> int:
913
            city_seen = self.city_keywords and any(
914
                re.search(rf"\b{re.escape(kw)}\b", state, re.I) \begin{tabular}{ll} for kw in self.city\_keywords \\ \end{tabular}
915
916
917
            in search = (
                re.search(r"search for a city", state, re.I)
or "select time zone" in state.lower()
918
919
920
            if city_seen and _clock_tab_selected(state) and not in_search:
921
922
                 return 3
            if city_seen:
923
924
                 return 2
            if _clock_tab_selected(state):
925
926
                 return 1
            return 0
927
928
929
        def _pl_set_alarm(self, state: str) -> int:
930
             if self._alarm_goal_met(state):
931
                return 3
932
            if "select time" in state.lower() or "alarm set for" in state.lower():
933
                 return 2
934
            if _alarm_tab_selected(state):
935
                 return 1
936
            return 0
937
938
        def _pl_adjust_alarm(self, state: str) -> int:
            if not self._orig_seen and _alarm_time_present(
939
                state, self.orig_hour24, self.orig_minute, None
940
                 self._orig_seen = True
942
943
            if (
                 _alarm_time_present(state, self.goal_hour24, self.goal_minute, None)
944
945
                and self._orig_seen
946
947
            if "select time" in state.lower() or "alarm set for" in state.lower():
948
949
            if _alarm_tab_selected(state) or self._orig_seen:
950
951
                return 1
            return 0
952
953
        def _pl_rename_timer(self, state: str) -> int:
954
            dialog_open = _rename_dialog_open(state)
955
            label_seen = _timer_label_present(state, self.goal_label)
956
            if label_seen and not dialog_open:
957
                 return 3
958
            if dialog_open:
959
960
                return 2
            if _timer_tab_selected(state):
961
                return 1
962
            return 0
963
964
965
        # Additional parsing / goal-checking helpers
966
967
968
        def _parse_recurrence_days(self, instr_l: str) -> Set[str]:
            days = {
    "sunday",
969
970
                "monday'
971
                 "tuesday"
972
                 "wednesday",
973
                 "thursday",
974
                 "friday"
975
                 "saturday"
976
                 "weekdays",
977
                "weekday",
"week day"
978
979
                 "week days",
980
                "weekends",
981
```

```
"every day",
982
                   "everyday",
983
984
              found: Set[str] = set()
985
              for d in days:
986
                  if d in instr_l:
987
                       if d in {
988
                            "weekdays",
989
                            "weekday'
990
                            "week day"
991
                            "week days",
992
                            "every day",
993
                            "everyday",
994
                       }:
995
                            found.update(
996
                                 {"monday", "tuesday", "wednesday", "thursday", "friday"}
997
998
                       elif d == "weekends":
999
                            found.update({"saturday", "sunday"})
                       else:
1001
1002
                            found.add(d)
              return found
1003
1004
1005
         def _alarm_goal_met(self, state: str) -> bool:
1006
               time & presence
              if self.alarm_any_time:
1007
1008
                   time_ok = _any_alarm_present(state)
1009
              else:
1010
                  time_ok = _alarm_time_present(
1011
                      state, self.goal_hour24, self.goal_minute, self.goal_ap
1012
1013
              if not time_ok or not _alarm_context_present(state):
1014
                   return False
1015
1016
              # recurrence handling
              if not self.recurrence_days:
1018
                   return True
1019
              # exact match
              if self.recurrence_days.issubset(_selected_weekdays(state)):
              # 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
1025
1026
1027
1028
              return False
1029
1030
         def _parse_alarm_goal_time(self):
              times = self._extract_times(self.instruction_norm_full)
1032
              if not times:
                   self.goal_hour24, self.goal_minute, self.goal_ap = _parse_alarm_time(
1034
                      self.instruction_norm_full
1036
                  return
1037
              alarm_pos = self.instruction_norm.rfind("alarm")
              chosen = next((t[:3] for t in times if t[3] > alarm_pos), times[0][:3])
1039
              self.goal_hour24, self.goal_minute, self.goal_ap = chosen
1040
1041
         def _parse_adjusted_alarm(self) -> Tuple[int, int]:
1042
              base_h, base_m, _ = _parse_alarm_time(self.instruction_norm_full)
1043
              m = re.search(
1044
                  r"\bby\s+(\d+)\s*(hour|hours|minute|minutes)\b", self.instruction_norm
1045
1046
              if m:
1047
                  num = int(m.group(1))
1048
                   unit = m.group(2)
1049
                   delta = num * (60 if "hour" in unit else 1)
                   total = (base_h * 60 + base_m + delta) % (24 * 60)
                   return total // 60, total % 60
1052
              time_tokens: List[Tuple[int, int]] = []
pat = re.compile(r"(\d{1,2})(?:[:.]\s*(\d{2}))?\s*(am|pm)", re.I)
1054
              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
1056
1058
1059
                   if ap == "am" and h == 12:
                       h = 0
1060
1061
                   time_tokens.append((h % 24, mnt))
1062
              if len(time_tokens) >= 2:
1063
                 return time_tokens[1]
```

```
return base_h, base_m
1064
1065
          @staticmethod
1066
          def _parse_city_name(instr_l: str) -> str:
1067
               parts = instr_l.split("add", 1)
if len(parts) >= 2:
1068
1069
                   tokens = parts[1].strip().split()
1070
                    city = []
                    for w in tokens:
                        if w in {"the", "a", "an"}:
continue
1073
1074
                         if w in {
"time",
1075
1076
                             "timezone",
                              "zone",
1078
                             "city",
1079
                             "in",
"to",
"for",
1080
1081
1082
                              "app",
1083
                              "on",
"world",
1084
1085
                              "country",
1086
1087
                        }:
                             break
1088
                         city.append(w)
1089
1090
                    if city:
               return " ".join(city).strip()
if " in " in instr_1:
1091
1092
                    _, after = instr_l.split(" in ", 1)
1093
                   tokens = after.strip().split()
1094
                    city = []
1095
1096
                    for w in tokens:
                        if w in {"the", "a", "an"}:
1097
1098
                             continue
1099
                         wd = w.rstrip(".,;!")
1100
                         if wd in {
                             "time",
"timezone",
1101
1102
                              "zone",
1103
1104
                              "city",
                              "for",
1105
                              "app",
1106
                              "on",
1107
                              "world",
1108
1109
                              "country",
1110
1111
                        city.append(wd)
urn " ".join(city).strip()
1112
                    return "
1113
               return ""
1114
          @staticmethod
1116
          def _extract_times(instr: str) -> List[Tuple[int, int, str, int]]:
1117
               instr_n = _normalize_time_text(instr)
1118
               pat = re.compile(r"(\d{1,2})(?:[:.]\s^*(\d{2}))?\s^*(am|pm)", re.I)
1119
               res = []
1120
               for m in pat.finditer(instr_n):
1121
                   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()))
1124
              return res
```

Listing 1 Android Control Generated Reward.

A.9 Prompts

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.

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.

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
- ullet 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:

- Analyze the trajectory to understand what constitutes success
- Identify intermediate progress that should be rewarded
- 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
- ullet 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:

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

```
{incorrect_trajectories}
```

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

OUTPUT FORMAT:

{expert_traj_str}

```
Answer in a json format as follows:

'reasoning': Briefly explain the corrective change you made.

'reward_class_code': Code for the Reward function class in the format:

> Reward format and extra info as above
```

 $\textbf{Prompt 3} \ \ \text{The prompt used for evolutionary mutation, providing feedback on incorrect trajectories.}$

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.)
- ullet Design for modularity you might extend this reward later to handle multiple goal types
- ullet 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 format and extra info as above
```

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