Do not distribute, this manuscript is a work in progress and will be released in December.

DEBUGGING TOOLS FOR REWARD MISALIGNMENT

Stephane Hatgis-Kessell[1]* **Serena Booth**[2] **W. Bradley Knox**[3] **Emma Brunskill**[1]
[1]Computer Science Department, Stanford University
[2]Department of Computer Science, Brown University
[3]Computer Science Department, The University of Texas at Austin

ABSTRACT

Training a reinforcement learning (RL) agent with a misspecified reward function can induce undesirable behavior, and writing a reward function that fully captures a designer's true objectives is difficult. We survey 38 RL practitioners and find that reward design remains largely manual. Practitioners used no LM tool at all for 63% of the tasks they described, often citing a preference for full manual control, yet for most tasks, they believed a meaningfully better reward function existed than the one they designed. Motivated by these results, we propose discordance-based reward debuggers. These tools take a candidate reward function and the environment source code as input and output hypotheses of potential reward misspecification: comparisons of trajectories in which the candidate reward function assigns higher return to one option but the reward designer is hypothesized to prefer the other. We build two such reward debuggers from agentic coding tools. Both uncover previously documented and novel reward misspecifications in popular RL environments, including a reward misspecification in ExploitGym that may have later been exploited in the wild by OpenAI agents. Additionally, in a human study, crowdworkers agree with the hypotheses of reward misspecification output by both reward debuggers more often than with comparisons sampled by two standard baselines. Broadly, we advocate for building alignment tools that support manual objective design, and release our reward debuggers as a step in that direction.

1 Introduction

Reinforcement learning (RL) requires a well-designed Markov decision process (MDP); within this MDP, the reward function is a particularly critical component, since training an RL agent with a misspecified reward function can induce undesirable, or potentially catastrophic, outcomes [Amodei et al., 2016]. Writing a reward function that fully captures a human designer's true objectives is difficult [Amodei et al., 2016, Booth et al., 2023, Knox et al., 2023, Paduraru et al., 2021], and small misspecifications or overlooked nuances can induce substantially suboptimal behavior [Pan et al., 2022, Krakovna et al., 2020].

One approach is to replace the written-down reward function with one that is learned from human data, whether preferences, demonstrations, corrections, feedback, or other sources [Jeon et al., 2020]. For example, RL from human feedback (RLHF) learns a reward function from human preference labels [Kaufmann et al., 2023], which has been especially critical in the design of steerable language models (LMs), but inferring a reward function from preferences requires modeling assumptions about how humans make decisions [Knox et al., 2024] and their ability to discern differences in data [Bıyık et al., 2019]. As another example, inverse RL learns a reward function from demonstrations, trading the difficulty of writing a reward function for the cost of collecting demonstrations from a sufficiently capable demonstrator [Hussein et al., 2017, Arora and Doshi, 2021]. While these approaches can reduce the burden of explicitly specifying a reward function, they also make it more difficult for designers to understand or modify the objectives for optimization.

The alternative approach is for designers to write reward functions explicitly, typically in code; we focus on this *manual reward specification* setting rather than on reward functions represented by learned parameters. Writing down a reward function remains common practice, and there is a resurgence in popularity of this technique stemming from RL with verifiable rewards (RLVR) [Lambert et al., 2024] and other forms of RL post-training for LMs. Such a reward function can be written by the designer or generated by an LM from a natural language task description (see Bhamidipaty et al.

*Correspondence to: stephhk@stanford.edu

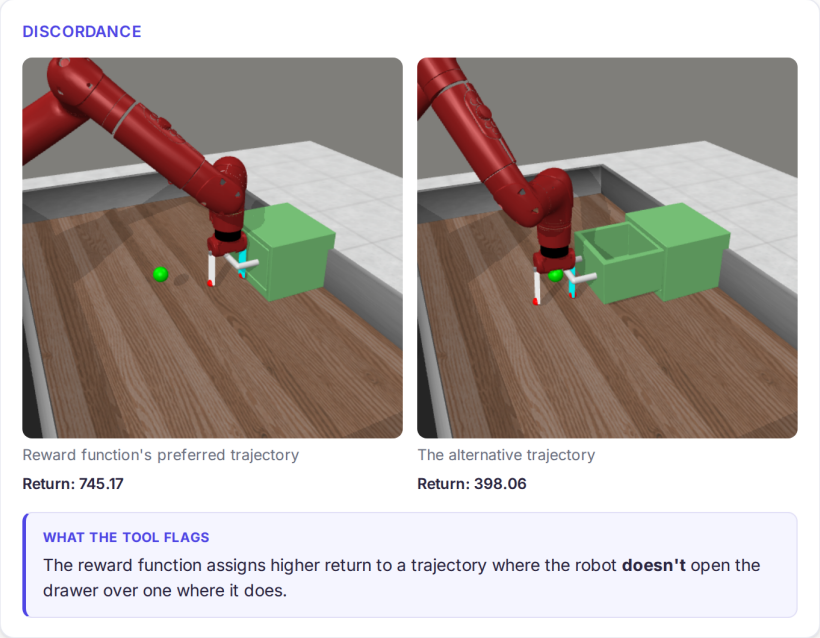


Figure 1: An example of a discordant comparison surfaced by our proposed reward debuggers for the default reward function from the `Drawer-open-v3` task in Metaworld [Yu et al., 2020]. The images above show the last state at the end of a trajectory.

[2026] for a review). Despite the prevalence of this technique, relatively little work has focused on building tools that help designers write better reward functions themselves; exceptions include Ratner et al. [2018], He and Dragan [2022], Knox et al. [2023], Knox and MacGlashan [2024], Hatgis-Kessell et al. [2025], Muslimani et al. [2025].

Before building such tools, we first seek to understand how practitioners design reward functions today. We survey 38 RL practitioners from 29 institutions on how they specify reward functions to ensure we can then build tools that support current practice. We find that reward design remains largely manual: practitioners used no LM tool at all for 63% of the tasks they described, even though 79% of respondents report using LM tools multiple times a day. Respondents also rarely reported learning the reward function from human demonstrations or preferences (8.3% of tasks), though our recruitment may under-sample practitioners who only use such methods (see Section 3). We find that practitioners generally preferred to retain control over the reward functions they designed, even when LM tools were available to generate them, and even though they often thought there was room for improvement in their designed reward functions.

Informed by this survey, we propose *discordance-based reward debuggers*: tools that take a candidate reward function and a description of the environment (e.g., the environment source code) as input and output hypotheses of potential reward misspecification. These hypotheses take the form of *discordant* comparisons: comparisons in which the candidate reward function prefers one option, but the reward designer is hypothesized to prefer the other (see Figure 1). Each comparison is either a trajectory pair or a lottery-trajectory pair, which compares a single trajectory to a lottery over two other trajectories. The reward designer—human or agentic—can inspect each comparison output by the discordance-based reward debugger and decide whether, and how, to update their reward function. To provide theoretical grounding, we show that, under noiseless and exhaustive preferences, searching for discordant lottery-trajectory pairs suffices, in principle, to find any misalignment in the candidate reward function; this result follows from the von Neumann–Morgenstern utility theorem [von Neumann and Morgenstern, 1947]. In practice, producing such comparisons requires collecting data from the environment and choosing informative comparisons; our key observation is that agentic coding tools can carry out this process. We introduce two discordance-based reward debuggers built from an agentic coding tool: the *single-prompt reward debugger* instructs the agentic coding tool with a single prompt, while the *scaffolded reward debugger* guides it through a more detailed, structured process.[2]

In popular RL environments, such as Meta-World [Yu et al., 2020], both reward debuggers uncover previously documented and novel reward misspecifications. We also run a human study in two environments where stakeholders disagree about the ideal reward function: controlling lockdowns during a COVID-19 pandemic and driving an autonomous vehicle on a highway. In both, crowdworkers agree with the reward debuggers' hypotheses of reward

[2]In this work, we use Claude Code [Anthropic, 2025].

misspecification more often than with comparisons sampled by two standard baselines. Aggregated across environments, this difference is statistically significant for both reward debuggers against both baselines. Further, in ExploitGym [Wang et al., 2026], both reward debuggers uncover a reward misspecification that may have later been exploited in practice, when a collective of AI agents attacked OpenAI and Hugging Face infrastructure while being evaluated on similar tasks [OpenAI, 2026, Wijk et al., 2026].[3]

We summarize our contributions below:

- We survey RL practitioners on how they design reward functions in practice. Our results provide a snapshot of RL practitioner behaviors as LMs become ubiquitous in their workflows. They also yield insights into open problems in reward specification and the extent to which LMs are used for reward design.
- Motivated by the survey, we introduce *discordance-based reward debuggers*: tools that input a candidate reward function and a description of the environment, and output hypotheses of potential reward misspecification.
- We propose two discordance-based reward debuggers built from agentic coding tools, both of which output trajectory pairs and lottery-trajectory pairs as hypotheses of reward misspecification. We draw on von Neumann and Morgenstern [1947] to show this type of comparison is sufficient to identify misalignment.
- We show that both reward debuggers uncover previously documented and novel reward misspecification in existing RL environments, including an issue in ExploitGym that may have later been exploited in the wild.
- In a human study, we show that crowdworkers agree with both reward debuggers' hypotheses of reward misspecification more often than with comparisons sampled by two ubiquitous sampling methods.

2 Preliminaries

Let $\mathcal{M}$ be an MDP$\backslash r$, an MDP $(\mathcal{S}, \mathcal{A}, T, D_0)$ without a reward function or a discount factor. Let $\hat{r}(s, a, s')$ and $\hat{\gamma}$ denote a candidate reward function and a candidate discount factor supplied by a reward designer, and let r and γ denote an unobservable, ground-truth reward function and discount factor that reflect the goals and purposes of the reward designer. A trajectory τ is the sequence of transitions $(s_0^\tau, a_0^\tau, s_1^\tau, \ldots, s_{|\tau|}^\tau)$, and its return under an arbitrary reward function $\tilde{r}$ and discount factor $\tilde{\gamma} \in [0, 1]$ is $G_{\tilde{r},\tilde{\gamma}}(\tau) = \sum_{t=0}^{|\tau|-1} \tilde{\gamma}^t \, \tilde{r}(s_t^\tau, a_t^\tau, s_{t+1}^\tau)$. The candidate is the pair $(\hat{r}, \hat{\gamma})$. Outside Section 4.1, $G_{\hat{r}}$ abbreviates $G_{\hat{r},\hat{\gamma}}$, and a statement that $\hat{r}$ ranks or prefers the options in a comparison, or that $\hat{r}$ is misaligned, is a statement about the return function $G_{\hat{r},\hat{\gamma}}$. In our experiments, we compute returns with $\hat{\gamma} = 1$, but the reward debuggers we propose do not require this. A lottery $\ell = \langle p, \tau_1, \tau_2 \rangle$ assigns probability p to τ_1 and $1 - p$ to τ_2, and its expected return is $\mathbb{E}_\ell[G_{\tilde{r},\tilde{\gamma}}] = \mathbb{E}_{\tau\sim\ell}\left[G_{\tilde{r},\tilde{\gamma}}(\tau)\right] = p\, G_{\tilde{r},\tilde{\gamma}}(\tau_1) + (1-p)\, G_{\tilde{r},\tilde{\gamma}}(\tau_2)$. A trajectory is the degenerate lottery placing all mass on itself, so we define a *comparison* as any element $(\ell_1, \ell_2) \in \mathcal{L} \times \mathcal{L}$, where $\mathcal{L}$ denotes the set of lotteries over trajectories in $\mathcal{M}$. We refer to a comparison between two trajectories as a *trajectory pair*, and to a comparison $(\langle p, \tau_1, \tau_2 \rangle, \tau_3)$ between a lottery with $p \in (0, 1)$ and a trajectory as a *lottery-trajectory pair*. Although a trajectory pair can be represented as a lottery-trajectory pair with $\tau_1 = \tau_2$, we use the term *lottery-trajectory pair* exclusively for comparisons where $\tau_1 \neq \tau_2$.

A lottery-trajectory pair is especially informative at its *indifference point*: the probability $p^\star$ at which a reward function $\tilde{r}$ assigns the lottery and the certain trajectory equal return, $G_{\tilde{r}}(\tau_3) = p^\star G_{\tilde{r}}(\tau_1) + (1 - p^\star)\, G_{\tilde{r}}(\tau_2)$. If the returns of

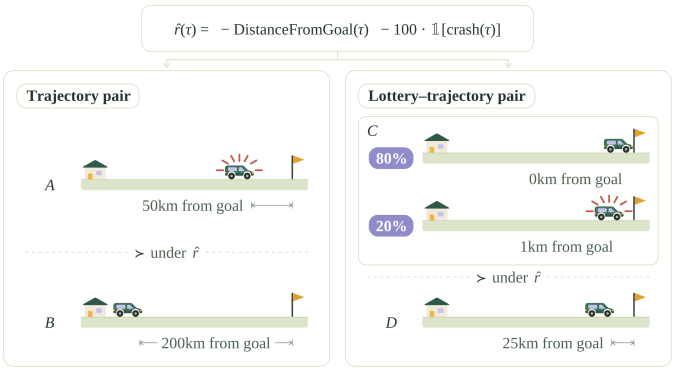


Figure 2: Two types of comparisons surfaced by a discordance-based reward debugger (Sec. 4.2) for a candidate reward function $\hat{r}$ in a task where a car drives to reach a goal but may stop at any point along the way. **Left, a trajectory pair:** in A the car crashes 50 km short of the goal and in B it never leaves home, so $G_{\hat{r}}(A) = -150 > G_{\hat{r}}(B) = -200$. $\hat{r}$ ranks crashing above staying put, which is an undesirable ranking. **Right, a lottery-trajectory pair:** the lottery C reaches the goal with probability 0.8 and crashes 1 km short with probability 0.2, while D stops 25 km short; here $\mathbb{E}_C[G_{\hat{r}}] = -20.2 > G_{\hat{r}}(D) = -25$. More generally, the given candidate $\hat{r}$ ranks any trajectory that stops more than 20.2 km from the goal below a lottery with a 20% chance of a crash, which is also undesirable.

[3] In our evaluations, we take steps to ensure the reward debuggers have no knowledge of that incident.

two of the trajectories are known, e.g., the best and worst outcomes the designer's indifference point exactly determines the return of the third.

Discordance-based reward debuggers (Section 4.2) rest on the premise that such discordant comparisons can help a reward designer easily recognize where their reward function is misspecified; prior work supports this premise [Knox et al., 2023, Muslimani et al., 2025]. Figure 2 illustrates an example of a trajectory pair and a lottery-trajectory pair.

Additionally, throughout this work, we use the term "LM tool" to refer to systems and harnesses built from LMs, including LM chat interfaces and agentic coding tools like Claude Code [Anthropic, 2025].

3 How do RL practitioners currently design reward functions?

To better understand how RL practitioners design reward functions, we surveyed 38 RL practitioners (8 in industry and 30 in academia) from 29 institutions in August 2026. We recruited respondents by sharing the survey with members of at least 5 academic institutions and attendees of the RL Conference (RLC), and by contacting practitioners in our professional networks. The recruitment message asked whether the recipient had designed a reward function in the past year (see Appendix H) . Each respondent described 1–2 tasks for which they designed a reward function in the past year, giving 60 tasks in total, and answered questions about how they designed the reward function for each task. These tasks span a range of domains, listed here with their corresponding number of responses: robotics (20), gridworlds (12), classic control (7), Atari and other games (7), LM agents (6), multi-agent settings (5), and multi-turn LM settings (3). Appendix H shows the full breakdown (Figure 18) and reports responses to all questions.

Our results support two findings that together motivate the rest of this paper: for most tasks, (1) respondents specified the reward function manually, without using any LM tools, commonly citing a preference for full manual control and distrust in the LM producing a correct reward function, and (2) respondents believed a meaningfully better reward function existed than the one they designed.

Reward design is not trivial. Figure 3 suggests that reward design is difficult: few task reward functions were described as easy to specify by respondents, and in only 8.3% of tasks did respondents say they were very satisfied with their designed reward function while in 28.3% of tasks respondents were not satisfied (21.7% neither satisfied nor dissatisfied, 6.7% dissatisfied).

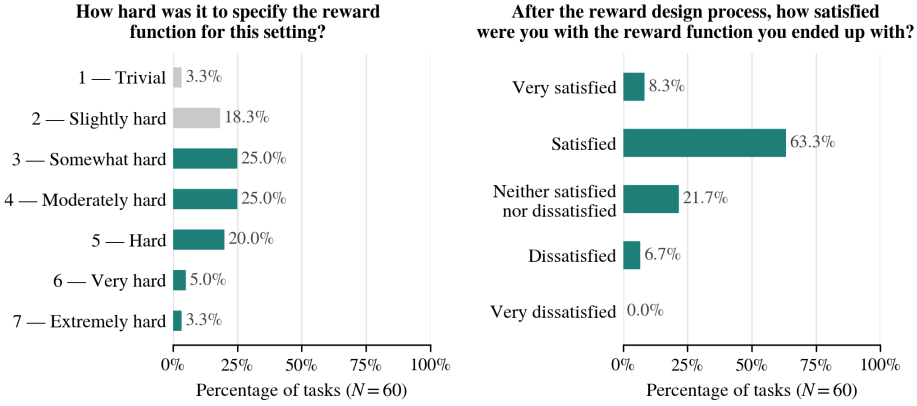


Figure 3: Reward design is not trivial.

Most RL practitioners think there exist better reward functions than the ones they designed. Figure 4 shows that practitioners largely believed that the reward functions they designed could have been improved: respondents reported that they thought no meaningfully better reward function existed for only 10% of tasks, with respondents also unsure for 8.3% of tasks. For a substantial share of tasks, participants highlighted misspecification of the objective as a potential issue, reporting that another reward function would induce a more desirable set of optimal policies (41.7% of tasks) or a ranking over outcomes better aligned with their intent (26.7% of tasks) . Therefore, for a non-trivial share of tasks reported on, participants indicated that there exists a more aligned reward function than the one they designed. Whether improving a reward function's alignment also improves learning speed or final performance is a worthwhile question beyond our scope, though initial evidence from Muslimani et al. [2026] suggests that it can.

Most reward functions are designed without LM tools. Although 78.9% of respondents reported using AI/LM tools multiple times a day, they reported prompting an LM tool at any point in the reward design process for 36.7% of

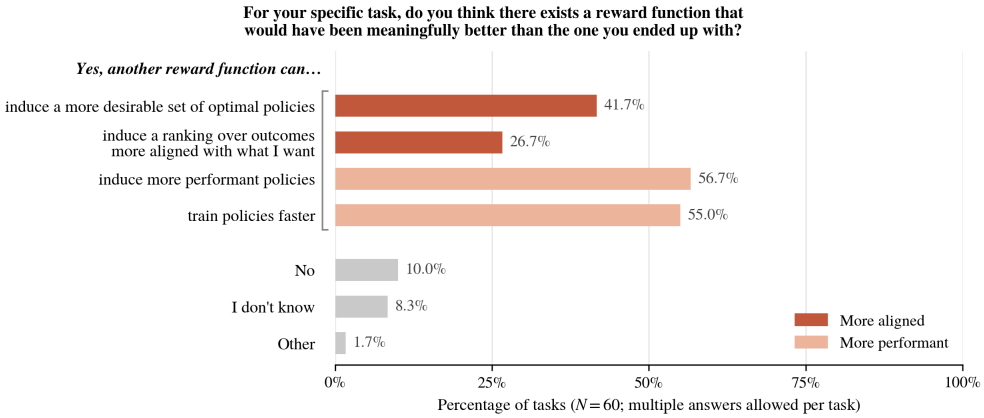


Figure 4: Most RL practitioners think there exists a better reward function than the one they designed. Dark red bars denote answers that concern the alignment of the designed reward function; light red bars concern induced performance (higher final performance or faster training).

tasks, and did not use one at all for the remaining 63.3%. When LM tools were used, they often served as an assistant rather than automating reward specification: for tasks where respondents used an LM tool, 50% involved using an LM to critique or evaluate a candidate reward function, which was as common as using an LM to generate a reward function (also 50%). For all tasks with a response to this question, participants reported re-prompting the LM at least once. This result stands in contrast to a large body of work on LM-based reward design [e.g., Kwon et al., 2023, Song et al., 2023, Yu et al., 2023, Ma et al., 2024, Xie et al., 2024, Venuto et al., 2024, Li et al., 2024, Zheng et al., 2025, Lee et al., 2026, Hatgis-Kessell et al., 2026, Bhamidipaty et al., 2026]; among the practitioners we surveyed, these methods do not yet appear to be widely used.[4] Further, respondents learned the reward function from human demonstrations or preferences for only 8.3% of tasks. However, we recruited respondents by asking whether they had designed a reward function in the past year, which some practitioners may have read as excluding reward functions learned from human data, so our sample may under-represent practitioners who use such methods. We therefore caution against using our results to compare how often reward functions are written down versus learned.

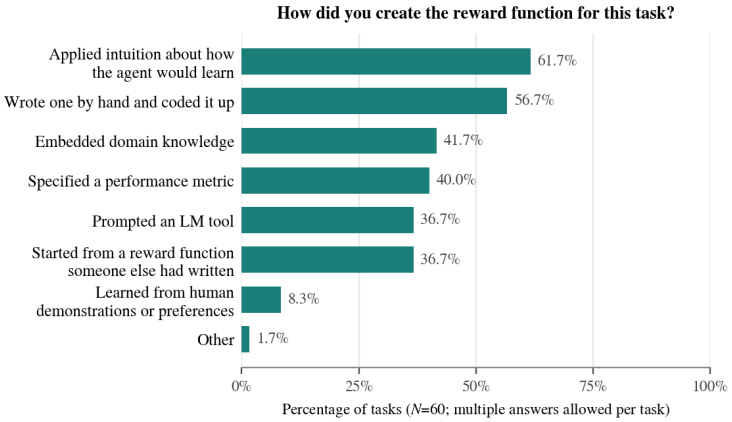


Figure 5: How did respondents specify the reward function for each task?

Why do RL practitioners not use LM tools more extensively for reward design? The most common reason respondents gave for not using an LM tool was a preference for full manual control over the reward design process, followed by distrust in the LM producing a correct or well-specified reward function and a belief that the LM lacked the necessary domain knowledge (Figure 6). Most respondents also thought they could design more aligned reward functions than an LM tool, judging themselves better at designing a reward function that would induce a more desirable set of optimal policies and a ranking over outcomes aligned with what they want; respondents were more evenly split on who is better at designing reward functions that train policies faster or induce more performant policies (Figure 7).

[4] Our sample may not reflect adoption in the broader RL community.

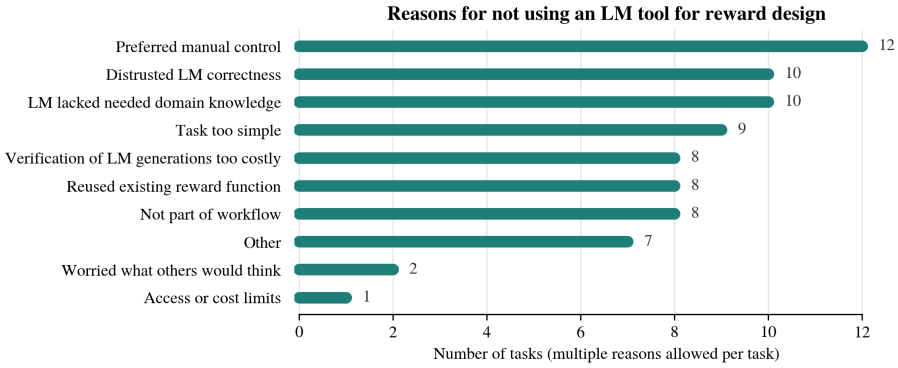


Figure 6: Why do the RL practitioners we surveyed not use an LM tool to design reward functions?

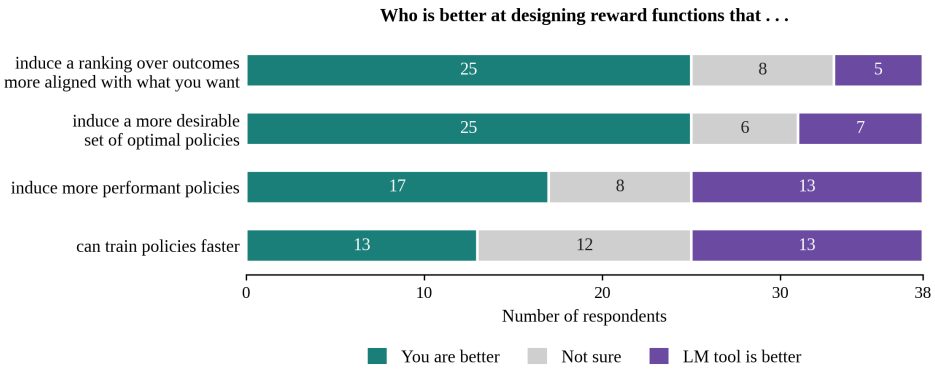


Figure 7: Do the RL practitioners we surveyed think they or an LM tool is better at designing reward functions?

4 Discordance-based reward debuggers

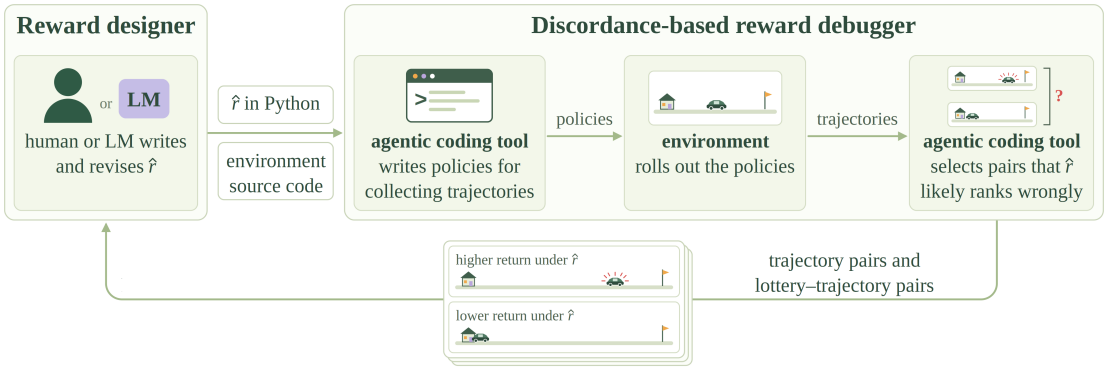


Figure 8: **Overview of a discordance-based reward debugger.** A reward designer (e.g., a human, an LM, or both) writes a candidate reward function $\hat{r}$ in Python and supplies it, along with the environment source code, to a reward debugger: an agentic coding tool that implements data-collection policies, rolls them out in the environment, and analyzes the resulting trajectories to construct a small set of comparisons indicating where $\hat{r}$'s induced ranking is likely misspecified. The designer then decides how to revise $\hat{r}$.

Respondents preferred full manual control over the reward design process, generally distrusted LM tools to design aligned reward functions, and found reward design difficult, with many believing that a meaningfully more aligned reward function existed than the one they ended up with. We therefore propose new LM tools that help reward designers specify more aligned reward functions while maintaining their control, in contrast to tools that automate reward design.

These tools take as input the environment source code and a candidate reward function, designed by a human reward designer or another LM tool, and output hypotheses of where that reward function might be misspecified (Figure 8).

Each hypothesis is a comparison, either a trajectory pair or a lottery-trajectory pair, on which the tool hypothesizes that the designer's preference ordering differs from the ordering induced by $G_{\hat{r}}$. Using these comparisons, the reward designer then chooses how to update the reward function.

We call any tool that helps a designer identify flaws in a reward function a *reward debugger*, and we call the narrower class to which our methods belong, which output such discordant comparisons, *discordance-based reward debuggers*. We formalize the broad objective of discordance-based reward debuggers as selecting a hypothesis set $\mathcal{D} \subseteq \mathcal{L} \times \mathcal{L}$ of at most k comparisons that maximizes $\mathbb{E}_{\hat{r}_\mathcal{D} \sim \mathcal{H}(\hat{r}|\mathcal{D})}[\mathrm{TAC}(\hat{r}_\mathcal{D}, r)]$, where the designer $\mathcal{H}$ revises the candidate reward function $\hat{r}$ into $\hat{r}_\mathcal{D}$ after viewing $\mathcal{D}$, and the Trajectory Alignment Coefficient (TAC) [Muslimani et al., 2025] measures how closely the trajectory ranking induced by $\hat{r}_\mathcal{D}$ matches that of the ground-truth reward function r (see Appendix B). In this work, we propose two discordance-based reward debuggers designed around that objective, which we refer to as the *single-prompt reward debugger* and the *scaffolded reward debugger* and describe later in this section.

Before doing so, we justify the type of comparisons our method returns: trajectory pairs and lottery-trajectory pairs over two trajectories, where a trajectory pair is the special case in which both lottery outcomes are the same trajectory. We show that no other type of comparison is needed to detect reward misalignment.

4.1 Lottery-trajectory pairs suffice to detect any misalignment

We show that, under noiseless and exhaustive preferences, lottery-trajectory pairs are sufficient to detect any misalignment under Definition 1 between a candidate reward function $\hat{r}$ and the ground-truth reward function r. The argument has three steps. First, following the uniqueness part of the von Neumann–Morgenstern utility theorem [von Neumann and Morgenstern, 1947], agreement on every lottery-trajectory pair determines a return function up to a positive affine transformation (Lemma 1). Second, it follows that $\hat{r}$ is misaligned with r if and only if some lottery-trajectory pair is discordant (Proposition 1); moreover, a candidate with no discordant lottery-trajectory pair has the same optimal policies as (r, γ) under any initial-state distribution and transition function (Corollary 1). Third, when the designer's preferences are noiseless and available for every lottery-trajectory pair, a discordant pair is exactly one on which the designer's reported preference differs from the candidate's, so misalignment can always be detected by the designer (Corollary 2).

In contrast, trajectory pairs alone are not sufficient in general to detect any misalignment (first remark in Appendix I), since they do not reflect how $\hat{r}$ trades off risk. Despite this, our reward debuggers also output deterministic trajectory comparisons, since these may be easier for reward designers to interpret.

Let $\mathcal{T}$ denote the set of all full trajectories, i.e., all sequences of states and actions that run for the environment's full length (infinite, or exactly H transitions when the environment has horizon H), including those infeasible under D_0 and T but excluding partial trajectories. We call $G \triangleq G_{r,\gamma}$ the ground-truth return function and $\hat{G} \triangleq G_{\hat{r},\hat{\gamma}}$ the candidate return function. Let $\Delta(\mathcal{T})$ be the set of probability distributions on $\mathcal{T}$. For $\mu \in \Delta(\mathcal{T})$, we write $\mathbb{E}_\mu[G] = \mathbb{E}_{\tau\sim\mu}[G(\tau)]$ for the expected ground-truth return under μ and $\mathbb{E}_\mu[\hat{G}] = \mathbb{E}_{\tau\sim\mu}[\hat{G}(\tau)]$ for the expected candidate return under μ. Let $\mathcal{C}$ be the set of lottery-trajectory pairs (ℓ, τ) with $\ell = \langle p, \tau_1, \tau_2\rangle$, $p \in (0, 1)$, $\tau_1 \neq \tau_2$, and $\tau, \tau_1, \tau_2 \in \mathcal{T}$. For $\mu, \nu \in \Delta(\mathcal{T})$, let $\sigma(\mu, \nu) = \operatorname{sign}\big(\mathbb{E}_\mu[G] - \mathbb{E}_\nu[G]\big)$ denote the preference between μ and ν induced by G, and let $\hat{\sigma}(\mu, \nu) = \operatorname{sign}\big(\mathbb{E}_\mu[\hat{G}] - \mathbb{E}_\nu[\hat{G}]\big)$ denote the preference induced by $\hat{G}$. A pair (μ, ν) is *discordant* if $\hat{\sigma}(\mu, \nu) \neq \sigma(\mu, \nu)$, i.e., if $\hat{G}$ and G rank μ and ν differently.

Definition 1 (Misalignment)**.** *We define misalignment as any disagreement between a human's preferences over trajectory distributions and those induced by a reward function. Formally, $\hat{G}$ is misaligned with G if and only if some pair $(\mu, \nu) \in \Delta(\mathcal{T}) \times \Delta(\mathcal{T})$ is discordant. In that case we also say that $(\hat{r}, \hat{\gamma})$ is misaligned with (r, γ).*

Because Definition 1 compares return functions, a candidate with $\hat{r} = r$ but $\hat{\gamma} \neq \gamma$ can be misaligned with (r, γ). When it is defined, alignment under Definition 1 implies a Trajectory Alignment Coefficient (TAC) [Muslimani et al., 2025] of 1, but not conversely. We make the following assumptions.

Assumption 1 (Bounded returns)**.** *G and $\hat{G}$ are defined on all of $\mathcal{T}$, bounded, and measurable.*

Assumption 2 (Noiseless return-based preferences)**.** *On every $(\ell, \tau) \in \mathcal{C}$, the preference that the designer reports is $\sigma(\ell, \tau)$: $\ell \succ \tau$ if $\sigma(\ell, \tau) = +1$, $\tau \succ \ell$ if $\sigma(\ell, \tau) = -1$, and indifference, $\ell \sim \tau$, if $\sigma(\ell, \tau) = 0$.*

Assumption 3 (Exhaustive comparisons)**.** *The designer's reported preference is available for every $(\ell, \tau) \in \mathcal{C}$, including comparisons involving trajectories that are infeasible under D_0 and T.*

Assumption 3 is strong; we use it, together with Assumption 2, only for Corollary 2, which connects the designer's reported preferences to the results below.

Lemma 1 (Lottery-trajectory pairs determine the return function up to a positive affine transformation)**.** *Under Assumption 1, $\hat{\sigma}(\ell, \tau) = \sigma(\ell, \tau)$ for every $(\ell, \tau) \in \mathcal{C}$ if and only if there exist $a > 0$ and $b \in \mathbb{R}$ such that $\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$.*

Proposition 1 (Lottery-trajectory pairs suffice to detect any misalignment)**.** *Under Assumption 1, $\hat{G}$ is misaligned with G if and only if some lottery-trajectory pair is discordant.*

To state Corollary 1, for an initial-state distribution D_0', a transition function T', and a policy π, let $\mu_\pi^{D_0', T'} \in \Delta(\mathcal{T})$ denote the distribution over the trajectories they generate. Writing $\mu \triangleq \mu_\pi^{D_0', T'}$ as shorthand, let $J^{D_0', T'}(\pi) \triangleq \mathbb{E}_\mu[G]$ denote the ground-truth expected return of π and $\hat{J}^{D_0', T'}(\pi) \triangleq \mathbb{E}_\mu[\hat{G}]$ the candidate expected return of π, and let $\Pi^\star(D_0', T')$ and $\hat{\Pi}^\star(D_0', T')$ denote the sets of policies that maximize $J^{D_0', T'}$ and $\hat{J}^{D_0', T'}$, respectively.

Corollary 1 (Identical policy rankings under any transition function)**.** *Under Assumption 1, if no lottery-trajectory pair is discordant, then there exist $a > 0$ and $b \in \mathbb{R}$ such that $\hat{J}^{D_0', T'}(\pi) = a\,J^{D_0', T'}(\pi) + b$ for every D_0', T', π. Hence, under each such D_0' and T', the candidate expected return and the ground-truth expected return rank all such policies identically, and in particular $\hat{\Pi}^\star(D_0', T') = \Pi^\star(D_0', T')$.*

Corollary 2 (Designer preferences over lottery-trajectory pairs detect any misalignment)**.** *Under Assumptions 1–3, $\hat{G}$ is misaligned with G if and only if, for at least one lottery-trajectory pair $(\ell, \tau) \in \mathcal{C}$, the preference that the designer reports differs from $\hat{\sigma}(\ell, \tau)$.*

Because the proofs for the results above use no property of G and $\hat{G}$ beyond Assumption 1 and no property of $\mathcal{T}$, the results also hold when $\mathcal{T}$ contains only the trajectories that are feasible under D_0 and T (in which case Corollary 1 holds only for D_0', T', and π that generate trajectories in $\mathcal{T}$), when trajectories are sequences of observations and actions, and when a return is assigned only to a whole trajectory (e.g., by an LM judge that scores a complete trace).

In practice, a discordance-based reward debugger can surface only a small subset of all lottery-trajectory pairs, so the challenge for tools like ours is selecting informative discordant comparisons from among all possible lottery-trajectory pairs. We give proofs in Appendix I.

4.2 Methods

We present two closely related methods for constructing hypotheses of reward misspecification, each built on an agentic coding tool (in our case, Claude Code with Opus 4.8 [Anthropic, 2025]). Both take as input a candidate reward function $\hat{r}$ implemented in Python and the environment source code, and follow the same high-level procedure: implement policies, roll them out in the environment to build a pool of trajectories, and then construct from that pool the comparisons on which the designer's preference is most likely to differ from the ranking induced by $\hat{r}$. These two methods differ in how much they control how the agentic coding tool carries out this procedure.

Single-prompt reward debugger The single-prompt reward debugger gives the agentic coding tool a single prompt describing the task (Figure 16) and returns comparisons. It tests how well an agentic coding tool can surface hypotheses of misspecification without explicit guidance.

Scaffolded reward debugger The scaffolded reward debugger instead guides the agentic coding tool through a fixed sequence of steps, each implemented as a skill file, i.e., a set of instructions that the tool loads and executes for that step. The scaffold encodes strategies we expect to make the search for misspecification more systematic:

1. **List potential flaws.** The tool reads $\hat{r}$ and writes a list of concrete ways it could be misaligned or exploited (e.g., accumulating a shaping bonus by hovering near the goal without reaching it). These hypothesized flaws guide the policies implemented in step 3.
2. **Simulate stakeholders.** The tool identifies 3–6 categories of stakeholders who may care about the task's outcomes and, for each, implements a trajectory scoring function from the perspective of a reasonable member of that category (e.g., a cautious passenger who penalizes high speeds and small gaps to other vehicles). These scoring functions act as alternative reward functions: a candidate discordant comparison arises when a stakeholder's scoring function ranks differently from $\hat{r}$.
3. **Collect trajectories.** The tool implements data-collection policies in Python that target the flaws from step 1 and outcomes where the stakeholder scoring functions from step 2 disagree with $\hat{r}$. It rolls these policies out to build a pool of trajectories.
4. **Select comparisons.** The tool selects up to five trajectory pairs and two lottery-trajectory pairs that a stakeholder scoring function ranks differently from $\hat{r}$, favoring comparisons that expose different flaws. For a

lottery $\langle p, \tau_A, \tau_B \rangle$ compared against a trajectory τ_C, it sets p near $\hat{r}$'s indifference point (see Section 2), so that $\hat{r}$ only barely prefers one option, following Knox et al. [2023].

Steps 1 and 2 are heuristics for constructing comparisons that are potentially informative of reward misspecification. Steps 3 and 4, which the single-prompt reward debugger also executes, leverage recent work showing that LMs are capable policy generators for sequential decision tasks [Liang et al., 2023, Hatgis-Kessell and Brunskill, 2026] and can implement novel exploration strategies [Nie et al., 2024, Arumugam and Griffiths, 2026]. See Appendix F for additional detail.

4.3 Qualitative analysis: surfacing misspecification in existing RL environments

Evaluation motivation We first explore if the reward debuggers can identify misspecifications in the reward functions of existing RL environments.

Evaluation setup We ran both reward debuggers on four common RL tasks: an autonomous vehicle overtaking other vehicles on a two-lane highway (`Highway-env` [Leurent, 2018]), a robot arm opening a drawer (`Drawer-open-v3`) and hammering in a nail (`Hammer-v3`) in Meta-World [Yu et al., 2020], and a robot arm wiping down a table (`Wipe` from robosuite [Zhu et al., 2020]). We also ran the reward debuggers on two other tasks: landing a rocket on a launch pad (`Rocket-recycling` [Zou, 2021]) and playing a song on the piano with a pair of robot hands (`RoboPianist` [Zakka et al., 2023]).

In each task the candidate reward function $\hat{r}$ is the default reward function shipped with the environment, and the debugger is given only the environment source code and $\hat{r}$. The first four tasks come from libraries that are widely used by the RL community and whose reward function weaknesses are, to varying degrees, described in prior work or in GitHub issue threads; the last two tasks have no publicly documented reward function flaws.

Results On all tasks, both methods surfaced comparisons exposing the same underlying flaw: dense per-step reward terms, such as bonuses for proximity to a goal or for speed, accumulate such that trajectories that fail at the task (e.g., by crashing or by never completing it) are assigned higher return than trajectories that complete the task as intended. Figure 9 summarizes the misspecifications identified by both methods in at least one of the up to 15 trajectory pairs each method surfaced.

Surfacing these misspecifications requires trajectories that are non-trivial to construct: some must result in task completion (e.g., opening a drawer or wiping the dirt off the table), and others must exploit $\hat{r}$ in a targeted way (e.g., holding the gripper on the drawer handle without opening it, or hovering the robot arm just above the table while cleaning no dirt). Both reward debuggers produced such trajectories without executing any classical RL algorithms, since the agentic coding tool writes the data-collection policies as Python code. These results provide preliminary evidence that agentic coding tools can write policies that elicit the specific behaviors needed to expose a reward function's flaws.

4.4 Fixing reward misspecification in existing RL environments

Evaluation motivation Because discordance-based reward debuggers output comparisons, a reward designer can also label their preferences over those comparisons and use the labels to update the candidate reward function directly. Our motivating vision is for the designer to inspect the surfaced hypotheses and revise the reward function themselves, but updating it from labeled preferences is potentially faster, and in our experiments serves as a proxy for how a reward designer might revise their reward function with the evidence returned by the discordance-based reward debugger. This approach has drawbacks, however: the designer does not edit the reward function themselves and so may understand the revised reward function less well, and satisfying a small set of labeled preferences does not guarantee that the revised reward function is aligned on comparisons outside that set.

Evaluation setup For three environments (`Drawer-open-v3`, `Hammer-v3`, and `Wipe`), one of the authors labeled their preferences over the up to 15 trajectory pairs and 6 lottery-trajectory pairs output by each reward debugger, labeling only comparisons on which they expected every stakeholder to share their preference. For each environment and each debugger's set of labeled comparisons, we then prompted the agentic coding tool to edit the reward function code so that it satisfies the labeled preferences. Updating a reward function by prompting an agentic coding tool differs from the typical RLHF approach of learning a reward function parameterized as a neural network [e.g., Christiano et al., 2017], but we find it to be effective in the evaluation below, and it leverages the fact that manually specified reward functions are often implemented in code. Hatgis-Kessell et al. [2025] propose a related framework for updating a manually specified reward function with preferences, but their method learns an additive correction term parameterized as a neural network rather than prompting an agentic coding tool to edit the reward function's implementation, as we do.

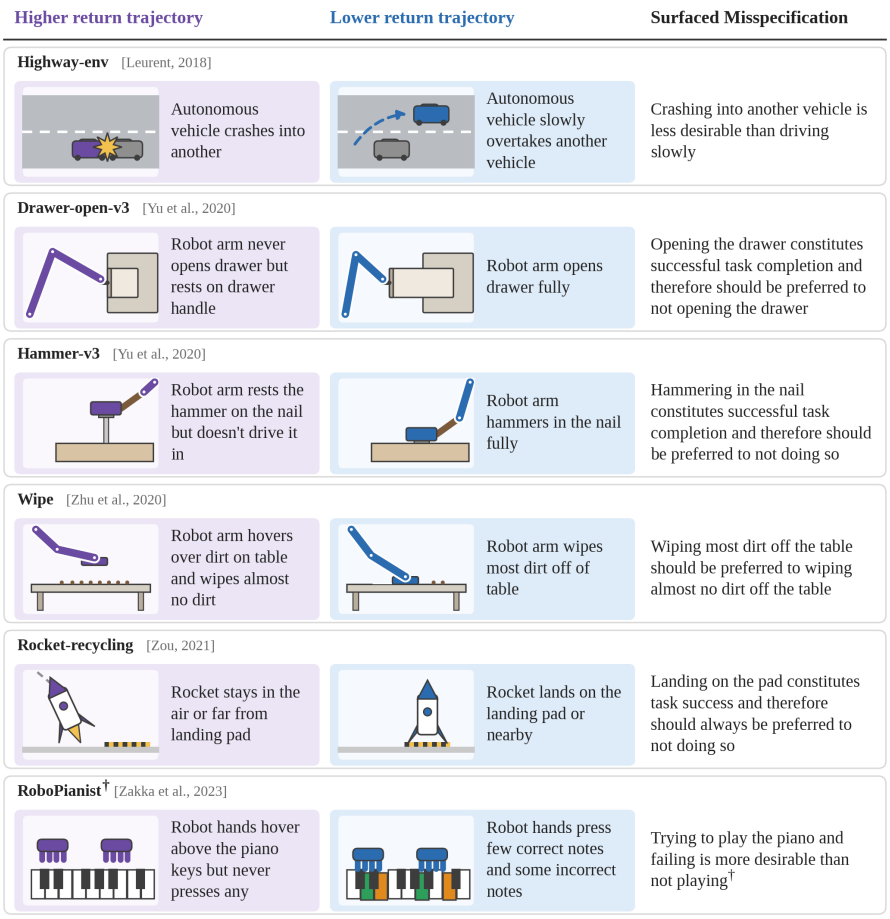


Figure 9: **Both reward debuggers uncover reward misspecification in existing RL environments.** Misspecifications surfaced by both reward debuggers in at least one of the first 15 trajectory pairs each returned. Each row describes the trajectories in a surfaced pair that $\hat{r}$ assigns higher and lower return; drawings are schematic. † The RoboPianist pair is more contestable than the other five: many people might prefer no playing to poor playing, even though flawed attempts are pedagogically useful. We nonetheless report it to illustrate that surfaced pairs are hypotheses for the designer to accept or reject rather than confirmed misspecifications.

The known misspecifications in these environments' default reward functions are only exploited by a policy trained with enough optimization pressure, so to stress-test each reward function we train policies with Soft Actor-Critic (SAC) and tune its hyperparameters with DEHB [Awad et al., 2021], a state-of-the-art evolutionary hyperparameter optimization method. We evaluate each policy under the environment's success metric, which differs from its default reward function: for `Drawer-open-v3` and `Hammer-v3`, a score of 1 when the task is completed (drawer opened fully, nail hammered in) and 0 otherwise, and for `Wipe`, the amount of dirt wiped off the table.[5] Appendix C adds detail.

Results Figure 10 shows the results. Updating the default reward function with a small number of preferences over surfaced comparisons results in more performant policies under the environment success metric. These policies, however, attain lower return under the default reward function, which further highlights its misspecifications. The comparisons surfaced by the reward debuggers can therefore also be used to update the input reward function with minimal additional human feedback.

4.5 Surfacing misspecification in ExploitGym when the flawed behavior cannot be realized

Evaluation motivation Some reward misspecifications induce behavior that is too harmful or too difficult to realize in practice, so a reward debugger that can only return real trajectories cannot surface them.

Evaluation setup We therefore adapt both reward debuggers to return *abstract trajectories*—text descriptions of trajectories that plausibly would arise from some policy—and evaluate them on three tasks from ExploitGym [Wang

[5]The two differ because the success metric does not provide enough signal to train a policy.

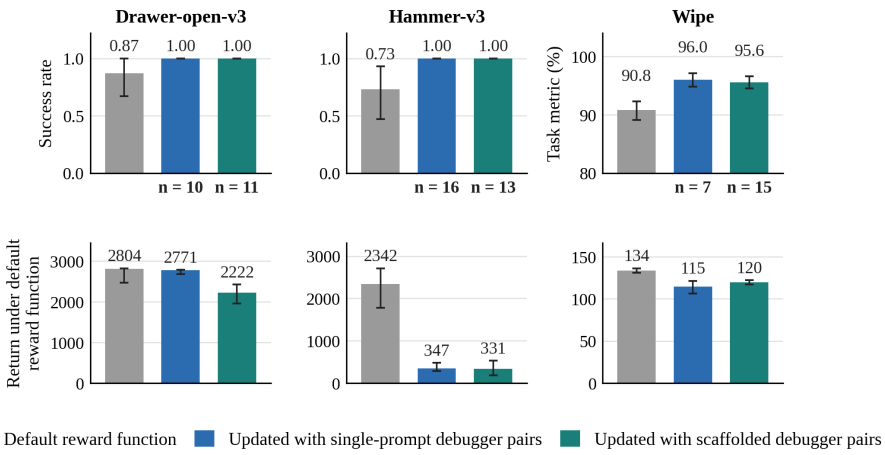


Figure 10: **Updating the default reward function with a small number of preferences over comparisons surfaced by each reward debugger yields more successful policies.** Top: success rate of policies trained with the environment's default reward function and with reward functions updated from an author's preferences over the comparisons surfaced by each reward debugger. Bottom: return of the same policies under the default reward function. n is the number of labeled preferences used to update the reward function; the author only labeled comparisons on which they were confident other stakeholders would agree. We report the mean for success rate and the interquartile mean (IQM) for return over the 15 policies per reward function and environment, with 95% confidence intervals computed with a percentile bootstrap.

et al., 2026], the benchmark whose evaluation was exploited in a large-scale reward hacking incident where OpenAI agents attacked Hugging Face to learn how the LM judge grading them worked [OpenAI, 2026, Wijk et al., 2026]. The reward debuggers used Claude Opus 4.8, which was trained prior to the incident, and ran without internet access, so the model had no knowledge of the incident; we, however, did select this evaluation setting with knowledge of it.

Results Given ExploitGym's default evaluation protocol as the candidate reward function, both reward debuggers surfaced the misspecification that was potentially realized in the OpenAI–Hugging Face incident (see Knox et al. [2026]): the reward function does not sufficiently penalize an agent for breaking out of the sandbox to attack the grading infrastructure. Both reward debuggers surfaced this misspecification on all three tasks. Both also surfaced, on all three tasks, that an agent could fabricate evidence to pass the LM judge without exploiting the intended bug (Table 3). This analysis therefore provides evidence that the reward debuggers could have surfaced the misspecification that may have contributed to the Hugging Face attack [Knox et al., 2026]. Full details are given in Appendix G.

4.6 Evaluating hypotheses of reward misspecification with a human study

Evaluation motivation We next evaluate the reward debuggers in environments where stakeholders may hold different preferences, so there is no single ground-truth reward function. We ask participants for their preferences over surfaced comparisons and measure how often they agree with the discordance, i.e., prefer the option that the candidate reward function ranks lower. A useful reward debugger should surface comparisons where participants agree with the discordance, even if participants do not all share the same preferences.

Evaluation setup We use two such environments. In `Highway-env` (Section 4.3), stakeholders may have different preferences regarding the speed of the autonomous vehicle or how aggressively it drives or overtakes other vehicles. In `Pandemic-control`, a policy controls the lockdown regulations during a COVID-19 outbreak in a city [Kompella et al., 2020]; higher lockdown restrictions better control the spread of COVID-19 but induce greater economic disruptions.

For each environment we design three candidate reward functions that each trade off features differently, e.g., vehicle speed versus distance to the nearest vehicle, lockdown regulation strictness versus economic implications (see Appendix D.1). We then input the candidate reward functions to each reward debugger, which produces a set of comparisons. We compare against two baselines that sample trajectories and then construct pairs by uniformly randomly pairing them: Uniform Sampling (US) samples trajectories from a policy that uniformly randomly samples actions, and Policy Sampling (PS) samples them from a policy trained to maximize expected return with respect to the candidate reward function. PS can be thought of as adapting the exploration strategy of posterior sampling methods for RLHF [e.g., Novoseller et al., 2020] to our setting.

Each method produces 9 comparisons per candidate reward function. For the reward debuggers, 2 of those are lottery-trajectory pairs, while the US and PS baselines produce only trajectory pairs. We then recruited non-expert participants

through Prolific.[6] Each participant was randomly assigned comparisons from one of the two environments and labeled their preference for each comparison, as well as the strength of their preference on a five-point scale. Each participant labeled 15 unique comparisons (3 per condition), each comparison was labeled by 3 participants, and 27 participants per environment remained after quality filtering. Full details are in Appendix D.

Results Figure 11 shows the results. In both environments, participants agree with the discordance of both reward debuggers' comparisons more often than with that of either baseline's. Pooled across environments, this difference is statistically significant at $p < 0.05$ for both reward debuggers against both baselines under a linear mixed-effects model (Appendix D.4). Within environments, the scaffolded reward debugger's discordances have significantly higher agreement than both baselines' in `Highway-env`, but only significantly higher than PS's in `Pandemic-control`. Conversely, the single-prompt reward debugger's discordances have significantly higher agreement than both baselines' in `Pandemic-control`, but only significantly higher than US's in `Highway-env`. Since the more effective method differs by environment, a reward designer may benefit from running both.

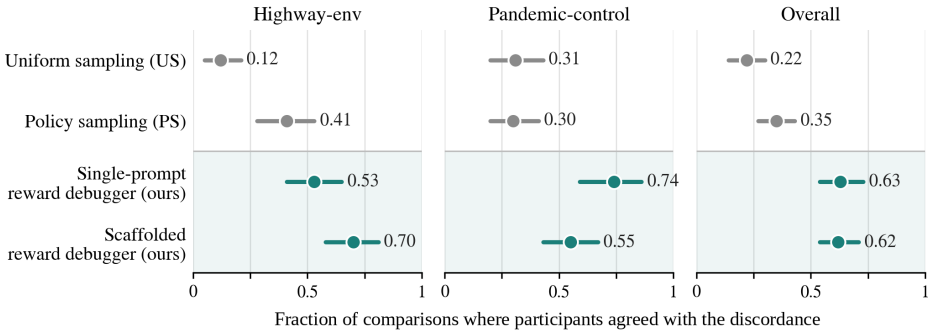


Figure 11: **Crowdworkers agree with the discordance of comparisons surfaced by the reward debuggers more often than with comparisons sampled by the baselines.** Each point is the fraction of a method's comparisons for which participants agreed with the discordance. Results are averaged over the three candidate reward functions per environment and all participants. Lines give 95% bootstrapped confidence intervals over participants.

Ablation: can an agentic coding tool select informative comparisons? The scaffolded reward debugger both builds a pool of candidate trajectories and selects comparisons from it. Here we investigate whether the informativeness of its comparisons is attributable to the agentic coding tool's selection step or whether the trajectory pool is already informative enough that any trajectory pairs drawn from it would serve as well. We isolate the selection step by comparing the scaffolded reward debugger's chosen comparisons against trajectory pairs sampled uniformly from the same pool. Figure 12 presents the results. Participants agree with the discordance of comparisons selected by the agentic coding tool more often than with that of trajectory pairs drawn uniformly at random from the same pool, indicating that the selection step—not the composition of the pool alone—results in informative comparisons.

Our method for constructing comparisons, which simply relies on prompting the agentic coding tool, is a substantial departure from prior work in RLHF, where trajectory pairs are selected from a candidate pool by algorithmic query-selection methods that typically require an explicit measure of uncertainty [Christiano et al., 2017, Sadigh et al., 2017, Bıyık and Sadigh, 2018, Bıyık et al., 2024, Lee et al., 2021, Muldrew et al., 2024, Ji et al., 2024]. Our results provide preliminary evidence that, given only a reward function implemented in Python, agentic coding tools can select informative comparisons for preference elicitation.

5 Related work

Beyond the work discussed throughout the paper, here we position our work more broadly.

Understanding reward design. Prior work has also studied how people design reward functions in practice. Booth et al. [2023] conduct a human study of RL practitioners designing reward functions, and Knox et al. [2023] analyze reward functions published for autonomous driving and identify widespread misdesign. Our survey complements this work by characterizing how RL practitioners currently specify reward functions, including the extent to which they use LM tools.

Finding reward misspecification. Closest to our work are methods that help designers find where a reward function may be misspecified. Assisted robust reward design proposes environments, drawn from a parameterized environment space, on which the designed reward function may fail [He and Dragan, 2022]. Brown et al. [2021] construct a small

[6]These participants were sampled from a general population of crowdworkers with likely no RL or environment-specific expertise.

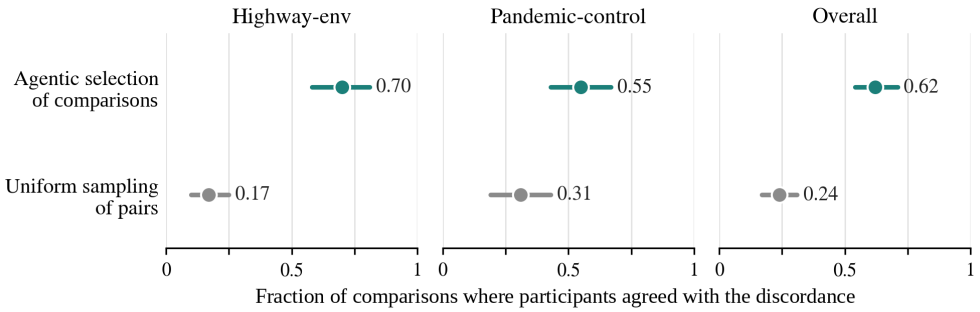


Figure 12: **Agentic selection produces informative comparisons.** Each point is the fraction of comparisons for which participants agreed with the discordance for the scaffolded reward debugger's selected comparisons (agentic selection) versus trajectory pairs drawn uniformly at random from the same candidate pool of trajectories (uniform sampling). Results are averaged over the three candidate reward functions per environment and all participants. Lines give 95% bootstrapped confidence intervals over participants.

set of queries to verify value alignment, but assume a reward function linear in known features and tabular MDPs. In contrast, discordance-based reward debuggers return discordant comparisons for the designer to inspect, and because they are built on agentic coding tools, they make neither assumption and apply readily to any environment with available source code and a written-down reward function.

Automated failure search in other fields. Falsification and adaptive stress testing search for trajectories on which an autonomous system violates a safety specification [Lee et al., 2020, Corso et al., 2021], and adversarial evaluation does the same for RL agents [Uesato et al., 2019]. Similarly, LMs and LM agents are used to red-team and audit other models [Perez et al., 2022, Li et al., 2025, Fronsdal et al., 2025]. These methods test a system against a specification that is assumed to be correct; discordance-based reward debuggers instead audit the specification itself.

6 Conclusion

We begin with the observation that building tools to support manual reward specification has received considerably less attention than alternatives that replace it, such as learning from human feedback or generating reward functions with LMs. Through a survey of 38 RL practitioners, we find that supporting manual reward specification is a worthwhile direction: most reward functions described by the practitioners we surveyed were still designed manually, often because practitioners preferred to retain control over their reward function. Nonetheless, most practitioners believed that the reward functions they designed could be meaningfully improved in some way. Motivated by these results, we propose discordance-based reward debuggers: tools built from agentic coding tools that surface hypotheses of potential reward misspecification for a reward designer to inspect. Both reward debuggers we propose uncover previously documented and novel reward misspecifications in popular RL environments, surface a misspecification in ExploitGym that may have later been exploited in practice in a major real-world security incident, and surface comparisons whose discordance human crowdworkers agree with more often than with comparisons from two standard sampling baselines. Future work can study what types of hypotheses—beyond comparisons—are informative of potential reward misspecifications. Further, we have yet to evaluate how effective RL practitioners find our proposed reward debuggers in practice, and expect such a study to yield interesting insights into how to design reward debuggers. Future discordance-based reward debuggers may also be useful when an AI system proposes a change to its own objective. Claude's constitution, for example, invites Claude to question it and to express disagreement; comparisons on which the current and proposed objectives disagree could give humans concrete, easy-to-understand evidence for accepting or rejecting the change. More broadly, we hope this work encourages the RL community to build tools that support manual reward specification, as it remains common practice.

Acknowledgements

This work has taken place in part in the Rewarding Lab at UT Austin. During this project, the Rewarding Lab has been supported by NSF (IIS-2402650, 2535195), ARO (W911NF-25-1-0254), Emerson, EA Ventures, UT Austin's Good Systems grand challenge, and Coefficient Giving.

References

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

Serena Booth, W. Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: misdesign through overfitting and invalid task specifications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5920–5929, 2023.

W. Bradley Knox, Alessandro Allievi, Holger Banzhaf, Felix Schmitt, and Peter Stone. Reward (mis)design for autonomous driving. *Artificial Intelligence*, 316:103829, 2023.

Cosmin Paduraru, Daniel J. Mankowitz, Gabriel Dulac-Arnold, Jerry Li, Nir Levine, Sven Gowal, and Todd Hester. Challenges of real-world reinforcement learning:definitions, benchmarks & analysis. *Machine Learning Journal*, 2021.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models, 2022. *URL https://arxiv. org/abs/2201.03544*, 2022.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of AI ingenuity. DeepMind Blog, April 2020. URL `https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/`. Accessed: 2026-10-02.

Hong Jun Jeon, Smitha Milli, and Anca Dragan. Reward-rational (implicit) choice: A unifying formalism for reward learning. *Advances in Neural Information Processing Systems*, 33:4415–4426, 2020.

Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. A survey of reinforcement learning from human feedback. *arXiv preprint arXiv:2312.14925*, 2023.

W. Bradley Knox, Stephane Hatgis-Kessell, Serena Booth, Scott Niekum, Peter Stone, and Alessandro Allievi. Models of human preference for learning reward functions. *Transactions on Machine Learning Research (TMLR)*, 2024.

Erdem Bıyık, Malayandi Palan, Nicholas C Landolfi, Dylan P Losey, and Dorsa Sadigh. Asking easy questions: A user-friendly approach to active reward learning. *arXiv preprint arXiv:1910.04365*, 2019.

Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys*, 50(2):1–35, 2017.

Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tülu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Logan Mondal Bhamidipaty, Lauren Robson, Linda Petrini, Shengrui Lyu, and Kamal Ndousse. A bird's-eye view of iterative reward design. *arXiv preprint arXiv:2610.04364*, 2026.

Ellis Ratner, Dylan Hadfield-Menell, and Anca D Dragan. Simplifying reward design through divide-and-conquer. *arXiv preprint arXiv:1806.02501*, 2018.

Jerry Zhi-Yang He and Anca D. Dragan. Assisted robust reward design. In *Proceedings of the 5th Conference on Robot Learning (CoRL)*, volume 164 of *Proceedings of Machine Learning Research*, pages 1234–1246, 2022.

W. Bradley Knox and James MacGlashan. How to specify reinforcement learning objectives. In *Finding the Frame: An RLC Workshop for Examining Conceptual Frameworks*, 2024.

Stephane Hatgis-Kessell, Logan Mondal Bhamidipaty, and Emma Brunskill. Repairing reward functions with feedback to mitigate reward hacking. *arXiv preprint arXiv:2510.13036*, 2025.

Calarina Muslimani, Kerrick Johnstonbaugh, Suyog Chandramouli, Serena Booth, W. Bradley Knox, and Matthew E. Taylor. Towards improving reward design in RL: A reward alignment metric for RL practitioners. In *Reinforcement Learning Conference (RLC)*, 2025.

John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, 2 edition, 1947.

Anthropic. Claude code. `https://code.claude.com/docs/en/overview`, 2025. Accessed: 2026-09-01.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning (CoRL)*, pages 1094–1100, 2020.

Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, Kurt Thomas, Yan Shoshitaishvili, Wenbo Guo, Jingxuan He, Thorsten Holz, and Dawn Song. Exploitgym: Can AI agents turn security vulnerabilities into real attacks? *arXiv preprint arXiv:2605.11086*, 2026. URL `https://arxiv.org/abs/2605.11086`.

OpenAI. The Hugging Face incident and the road ahead. `https://openai.com/index/hugging-face-incident-and-the-road-ahead/`, 2026. Accessed: 2026-09-07.

Hjalmar Wijk, Ajeya Cotra, and Ryan Greenblatt. Brief independent investigation of agents' behavior, reasoning and collaboration in the OpenAI/Hugging Face hacking incident. `https://metr.org/blog/2026-08-26-openai-hugging-face-incident-investigation/`, 2026. METR and Redwood Research. Accessed: 2026-09-07.

Calarina Muslimani, Yunshu Du, Kenta Kawamoto, Kaushik Subramanian, Peter Stone, and Peter Wurman. The trajectory alignment coefficient in two acts: From reward tuning to reward learning. *arXiv preprint arXiv:2601.16906*, 2026.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Jiayang Song, Zhehua Zhou, Jiawei Liu, Chunrong Fang, Zhan Shu, and Lei Ma. Self-refined large language model as automated reward function designer for deep reinforcement learning in robotics. *arXiv preprint arXiv:2309.06687*, 2023.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis. In *Conference on Robot Learning (CoRL)*, 2023.

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. In *International Conference on Learning Representations (ICLR)*, 2024.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2024.

David Venuto, Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with VLMs. In *International Conference on Machine Learning (ICML)*, 2024.

Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. Auto MC-reward: Automated dense reward design with large language models for minecraft. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Qinqing Zheng, Mikael Henaff, Amy Zhang, Aditya Grover, and Brandon Amos. Online intrinsic rewards for decision making agents from large language model feedback. *Reinforcement Learning Journal*, 6:139–164, 2025. Presented at the Reinforcement Learning Conference (RLC).

Hojoon Lee, Ajay Subramanian, Ben Abbatematteo, Vijay Veerabadran, Pedro Matias, Karl Ridgeway, and Nitin Kamra. RDA: Reward design agent for reinforcement learning. *Reinforcement Learning Journal*, 7, 2026. Presented at the Reinforcement Learning Conference (RLC).

Stephane Hatgis-Kessell, W. Bradley Knox, and Emma Brunskill. Specifying reward functions for RL without environment sampling. In *Third Workshop on Reinforcement Learning Beyond Rewards at the Reinforcement Learning Conference (RLC)*, 2026. URL `https://openreview.net/forum?id=xCe2Hh33Et`.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023.

Stephane Hatgis-Kessell and Emma Brunskill. When are llms sufficient policy optimizers for sequential rl tasks? *arXiv preprint arXiv:2605.30719*, 2026.

Allen Nie, Yi Su, Bo Chang, Jonathan Lee, Ed H Chi, Quoc V Le, and Minmin Chen. Evolve: Evaluating and optimizing llms for exploration. 2024.

Dilip Arumugam and Thomas L Griffiths. Toward efficient exploration by large language model agents. In *International Conference on Learning Representations*, volume 2026, pages 40734–40772, 2026.

Edouard Leurent. An environment for autonomous driving decision-making. `https://github.com/Farama-Foundation/HighwayEnv`, 2018.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. 2020.

Zhengxia Zou. Rocket-recycling with reinforcement learning. `https://github.com/jiupinjia/rocket-recycling`, 2021.

Kevin Zakka, Philipp Wu, Laura Smith, Nimrod Gileadi, Taylor Howell, Xue Bin Peng, Sumeet Singh, Yuval Tassa, Pete Florence, Andy Zeng, and Pieter Abbeel. Robopianist: Dexterous piano playing with deep reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2023.

Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

Noor Awad, Neeratyoy Mallik, and Frank Hutter. Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization. *arXiv preprint arXiv:2105.09821*, 2021.

W. Bradley Knox, Serena Booth, and Brian Christian. An unexamined cause of the OpenAI Hugging Face hacking incident: Its binary performance metric. LessWrong, September 2026. URL `https://www.lesswrong.com/posts/HsijShdRdAg5sPKnF/an-unexamined-cause-of-the-openai-hugging-face-hacking`. Blog post. Accessed: 2026-10-14.

Varun Kompella, Roberto Capobianco, Stacy Jong, Jonathan Browne, Spencer Fox, Lauren Meyers, Peter Wurman, and Peter Stone. Reinforcement learning for optimization of covid-19 mitigation policies. *arXiv preprint arXiv:2010.10560*, 2020.

Ellen Novoseller, Yibing Wei, Yanan Sui, Yisong Yue, and Joel Burdick. Dueling posterior sampling for preference-based reinforcement learning. In *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI)*, volume 124 of *Proceedings of Machine Learning Research*, pages 1029–1038. PMLR, 2020.

Dorsa Sadigh, Anca D. Dragan, Shankar Sastry, and Sanjit A. Seshia. Active preference-based learning of reward functions. In *Robotics: Science and Systems (RSS)*, 2017.

Erdem Bıyık and Dorsa Sadigh. Batch active preference-based learning of reward functions. In *Proceedings of the 2nd Conference on Robot Learning (CoRL)*, volume 87 of *Proceedings of Machine Learning Research*, pages 519–528. PMLR, 2018.

Erdem Bıyık, Nicolas Huynh, Mykel J Kochenderfer, and Dorsa Sadigh. Active preference-based gaussian process regression for reward learning and optimization. *The International Journal of Robotics Research*, 2024. doi:10.1177/02783649231208729.

Kimin Lee, Laura Smith, Anca Dragan, and Pieter Abbeel. B-pref: Benchmarking preference-based reinforcement learning. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

William Muldrew, Peter Hayes, Mingtian Zhang, and David Barber. Active preference learning for large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 36577–36590. PMLR, 2024.

Kaixuan Ji, Jiafan He, and Quanquan Gu. Reinforcement learning from human feedback with active queries. *arXiv preprint arXiv:2402.09401*, 2024.

Daniel S. Brown, Jordan Schneider, Anca D. Dragan, and Scott Niekum. Value alignment verification. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 1105–1115, 2021.

Ritchie Lee, Ole J. Mengshoel, Anshu Saksena, Ryan W. Gardner, Daniel Genin, Joshua Silbermann, Michael Owen, and Mykel J. Kochenderfer. Adaptive stress testing: Finding likely failure events with reinforcement learning. *Journal of Artificial Intelligence Research*, 69:1165–1201, 2020. doi:10.1613/jair.1.12190.

Anthony Corso, Robert J. Moss, Mark Koren, Ritchie Lee, and Mykel J. Kochenderfer. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of Artificial Intelligence Research*, 72:377–428, 2021. doi:10.1613/jair.1.12716.

Jonathan Uesato, Ananya Kumar, Csaba Szepesvári, Tom Erez, Avraham Ruderman, Keith Anderson, Krishnamurthy Dvijotham, Nicolas Heess, and Pushmeet Kohli. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. In *International Conference on Learning Representations (ICLR)*, 2019.

Ethan Perez, Saffron Huang, Francis Song, Trevor Cai, Roman Ring, John Aslanides, Amelia Glaese, Nat McAleese, and Geoffrey Irving. Red teaming language models with language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3419–3448, 2022.

Xiang Lisa Li, Neil Chowdhury, Daniel D. Johnson, Tatsunori Hashimoto, Percy Liang, Sarah Schwettmann, and Jacob Steinhardt. Eliciting language model behaviors with investigator agents. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, 2025.

Kai Fronsdal, Isha Gupta, Abhay Sheshadri, Jonathan Michala, Stephen McAleer, Rowan Wang, Sara Price, and Samuel R. Bowman. Petri: An open-source auditing tool to accelerate AI safety research. Anthropic Alignment Science Blog, 2025. URL `https://alignment.anthropic.com/2025/petri/`. Accessed: 2026-10-15.

Xuanchang Zhang, Wei Xiong, Lichang Chen, Tianyi Zhou, Heng Huang, and Tong Zhang. From lists to emojis: How format bias affects model alignment. *arXiv preprint arXiv:2409.11704*, 2024.

Mrinank Sharma, Meg Tong, Tomasz Korbak, David Duvenaud, Amanda Askell, Samuel R. Bowman, Newton Cheng, Esin Durmus, Zac Hatfield-Dodds, Scott R. Johnston, Shauna Kravec, Timothy Maxwell, Sam McCandlish, Kamal Ndousse, Oliver Rausch, Nicholas Schiefer, Da Yan, Miranda Zhang, and Ethan Perez. Towards understanding sycophancy in language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Marcus Williams, Micah Carroll, Adhyyan Narang, Constantin Weisser, Brendan Murphy, and Anca Dragan. On targeted manipulation and deception when optimizing LLMs for user feedback. *arXiv preprint arXiv:2411.02306*, 2024.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 3rd edition, 2010.

Ike Obi, Rohan Pant, Srishti Shekhar Agrawal, Maham Ghazanfar, and Aaron Basiletti. Value imprint: A technique for auditing the human values embedded in RLHF datasets. In *Advances in Neural Information Processing Systems 37 (NeurIPS) Datasets and Benchmarks Track*, 2024.

Arduin Findeis, Timo Kaufmann, Eyke Hüllermeier, Samuel Albanie, and Robert Mullins. Inverse constitutional AI: Compressing preferences into principles. In *International Conference on Learning Representations (ICLR)*, 2025.

Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning (ICML)*, pages 278–287, 1999.

Jette Randløv and Preben Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *International Conference on Machine Learning (ICML)*, pages 463–471, 1998.

Yujing Hu, Weixun Wang, Hangtian Jia, Yixiang Wang, Yingfeng Chen, Jianye Hao, Feng Wu, and Changjie Fan. Learning to utilize shaping rewards: A new approach of reward shaping. *Advances in neural information processing systems*, 33:15931–15941, 2020.

Sinan Ibrahim, Mostafa Mostafa, Ali Jnadi, Hadi Salloum, and Pavel Osinenko. Comprehensive overview of reward engineering and shaping in advancing reinforcement learning applications. *IEEE Access*, 12:175473–175500, 2024.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Julian Dierkes, Emma Cramer, Holger H Hoos, and Sebastian Trimpe. Combining automated optimisation of hyperparameters and reward shape. *arXiv preprint arXiv:2406.18293*, 2024.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International conference on machine learning*, pages 10835–10866. PMLR, 2023.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

Joar Max Viktor Skalse, Matthew Farrugia-Roberts, Stuart Russell, Alessandro Abate, and Adam Gleave. Invariance in policy optimisation and partial identifiability in reward learning. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 32033–32058. PMLR, 2023. URL `https://proceedings.mlr.press/v202/skalse23a.html`.

A On manual specification versus RLHF

Most RL practitioners we surveyed still design reward functions manually, prefer to retain control over the reward function they design even when LM tools are available to generate those reward functions, and nonetheless find reward design difficult.

One alternative is to replace manual specification altogether, such as with RLHF. Our proposal to build tools to support manual reward specification does not necessarily depend on manual specification being a more desirable alternative to other methods: we find that practitioners already often design reward functions manually and believe their resulting reward functions could be better. Those results motivate the need for tools that better support current practice. Still, manual specification has advantages over RLHF that are worth making explicit, and we discuss several below.

Manually specifying reward functions empowers builders of AI systems. When labeling preferences to train a reward model, it is unclear how those preferences will translate into the behavior of the agent trained with the resulting reward model. Suppose all a reward designer knows is that the AI system will satisfy the preferences they labeled. Then their understanding of how the agent will behave depends entirely on what they labeled preferences over and how they understand that to connect to the agent's deployment. They therefore have less control over how the resulting agent behaves than they would if they could trace how their input to the design process determines its behavior. For example, if they are training an LM and tend to prefer responses with some emojis over responses with no emojis, a reward model trained with their preferences may produce responses that consist entirely of emojis [Zhang et al., 2024]; this is a behavior someone may not predict when labeling preferences, but could be comparatively easier to mitigate if they were to explicitly write down the reward function rather than deriving it from preferences.[7] A similar argument holds when someone prefers responses that validate their feelings and affirm their actions: the resulting reward model may induce a sycophantic model that affirms all users regardless of whether affirmation is warranted [Sharma et al., 2024, Williams et al., 2024].

If someone were instead designing an objective for COVID-19 pandemic regulation, they might reasonably prefer to explicitly specify how deaths and infection rates should be traded off against the economic costs of stricter lockdowns, instead of leaving that tradeoff implicit in their preferences over behaviors. Then the pandemic regulation policy trained with the objective they wrote down will make decisions that they are not surprised by and can endorse. Manual reward specification forces a reckoning that preferences allow us to avoid. As Russell and Norvig [2010, Ch. 16] observe, when we decline to assign an explicit value to human life, we tend to undervalue it systematically; preferences leave such tradeoffs implicit and opaque, whereas writing down the objective compels us to confront exactly how much each consideration weighs.

A manually specified reward function is explicit and so other stakeholders can scrutinize it, while learning from human feedback produces an opaque objective. When the output of RLHF is a reward model that satisfies the dataset of collected preferences, the principles or values that reward model embodies are generally unclear. A line of work attempts to decode this information, such as by classifying the values embedded in RLHF datasets [Obi et al., 2024], or deducing an implicit natural-language constitution from preference data [Findeis et al., 2025]. But reliably extracting even a basic understanding of a reward model learned from preferences remains an open research problem. On the other hand, manually specified reward functions can make explicit exactly what features the objective is defined over (e.g., does an objective for pandemic regulation account for economic impacts of lockdown restrictions?) and how they are traded off. The implicit values embedded in a manually specified reward function can still be hard to extract; for example, Knox et al. [2023] find that some reward functions for autonomous driving assign higher return to crashing than to not driving at all. But because such functions are written in a human-readable format (e.g., Python code, a natural-language constitution), they are far easier to reason about than a black-box neural network. Alternatively, practitioners can learn a linear reward function over interpretable features [Sadigh et al., 2017, Hatgis-Kessell et al., 2026]. Though uncommon in practice—perhaps due to limited expressivity—this avoids the opaqueness of learned reward models without requiring manual specification, but inherits the other aforementioned flaws of reward functions specified by human feedback.

Manually specifying a reward function makes clear whether an objective is defined over outcomes or behaviors. Reward functions can be specified over outcomes, behaviors, or both. Outcome-based reward functions assign reward only when an outcome is achieved (e.g., $+1$ for winning a game and -1 otherwise, or verifiable rewards in language model settings). Such functions are often insufficient to train a policy with RL, however, because they provide little feedback during training. Techniques like reward shaping therefore craft reward functions over behaviors, which provide feedback on agent behavior throughout training rather than only once an outcome is reached [Ng et al., 1999, Randløv and Alstrøm, 1998, Hu et al., 2020, Booth et al., 2023, Ibrahim et al., 2024]. But in practice, absent a

[7] Based on this example, we suspect that, in practice, manually specified reward functions may often induce policies that are more robust to out-of-distribution deployment settings than reward models learned from preferences.

potential function equal to the optimal value function, shaped reward functions may fail to train agents that surpass human capabilities: the shaping terms encode the designer's prior beliefs about which intermediate behaviors constitute progress, biasing the policy toward strategies the designer can already conceive of. An agent rewarded for behaving in ways a human recognizes as good is limited by that human's understanding of the task; an agent rewarded only for outcomes is free to discover superhuman strategies—AlphaGo's move 37 being the canonical example [Silver et al., 2016].

Learning a reward function from a dataset of preferences conflates outcome-based and behavior-based reward functions; it is unclear whether preferences labeled by a human are over outcomes or behaviors, and whether the trained reward model assigns rewards for outcomes or behaviors. Manual specification lets a reward designer draw this distinction explicitly, which may enable training more capable agents.

B A formalization of discordance-based reward debuggers

Here we formalize the design objective for discordance-based reward debuggers; while this objective is unrealizable in practice, it provides guidance on how to design them.

B.1 Measuring alignment

We measure the alignment between a reward function $\tilde{r}$ and the normally unobservable ground-truth reward function r using the Trajectory Alignment Coefficient (TAC) [Muslimani et al., 2025]. TAC is the correlation between the ranking of trajectories induced by $\tilde{r}$ and the ranking induced by r, among trajectories sharing a start state. $\mathrm{TAC} = 1$ when the two reward functions rank every pair identically and $\mathrm{TAC} = -1$ when they invert every pair.

B.2 System design objective

Let $\mathcal{H}$ denote a reward designer—a human, an LM tool, or a human working with an LM tool—modeled as a stochastic map that revises a candidate reward function in light of hypotheses. Given the candidate $\hat{r}$ and a hypothesis set $\mathcal{D}$, the designer produces a revised reward function $\hat{r}_{\mathcal{D}} \sim \mathcal{H}(\hat{r} \mid \mathcal{D})$. We place no constraints on how the designer revises the reward function: the system surfaces hypotheses and the designer decides what, if anything, to change.

The hypothesis set is a set of at most k comparisons, $\mathcal{D} = \{(\ell_1^i, \ell_2^i)\}_{i=1}^k$ with $(\ell_1^i, \ell_2^i) \in \mathcal{L} \times \mathcal{L}$, where each comparison is either a trajectory pair or a lottery-trajectory pair. The budget k is small, reflecting that the hypotheses must be comprehensible to the reward designer.

Our objective is to construct the hypothesis set that most improves the alignment of the designer's revised reward function with respect to their unobservable ground-truth reward function:

$$\mathcal{D}^\star \in \underset{\mathcal{D} \subseteq \mathcal{L}\times\mathcal{L},\ |\mathcal{D}|\leq k}{\arg\max} \ \mathbb{E}_{\hat{r}_{\mathcal{D}} \sim \mathcal{H}(\hat{r}|\mathcal{D})}\big[\,\mathrm{TAC}(\hat{r}_{\mathcal{D}}, r)\,\big]. \tag{1}$$

Because r is unobservable and $\mathcal{H}(\hat{r} \mid \mathcal{D})$ is unknown, Eq. 1 defines the objective around which we design the system rather than a quantity that the system can evaluate directly. In practice the system searches for comparisons that are likely to expose distinct flaws in $\hat{r}$ (Section 4.2).

C Updating reward functions with preferences over surfaced comparisons

Here we add detail to Section 4.4, which summarizes our results when updating an environment's default reward function with preferences over comparisons surfaced by each reward debugger.

Across the three environments, the author labeled 11, 13, and 15 preferences over the scaffolded reward debugger's output comparisons for `Drawer-open-v3`, `Hammer-v3`, and `Wipe`, respectively, and 10, 16, and 7 preferences over the single-prompt reward debugger's output comparisons. The number of preferences per environment and method varies because the two methods return a different number of comparisons depending on the misspecifications they identify—up to 15 trajectory pairs and 6 lottery-trajectory pairs for each reward debugger per environment and reward function—and because the author then labeled preferences only over comparisons on which they expected every stakeholder to share their preference. We prompt the agentic coding tool to update the reward function implementation given the labeled pairs for each environment by simply using the prompt: "Now update the reward function such that the following preferences are satisfied".

We aim to train a policy to maximize the expected return with respect to the given reward function (i.e., either an updated reward function or the environment's default). When training a policy with the default reward function, we only observe misspecifications in the reward function being exploited when enough optimization pressure is applied; this observation matches that of prior work in the `Wipe` environment [Dierkes et al., 2024], and is an example of reward overoptimization [Gao et al., 2023]. Therefore, to stress-test each reward function, we train policies with SAC and DEHB [Awad et al., 2021] hyperparameter optimization. DEHB tunes four hyperparameters: the learning rate (log-uniform over $[10^{-6}, 10^{-3}]$), the discount factor γ (searched log-uniformly over $1-\gamma \in [0.001, 0.2]$), the batch size ($\{64, 128, 256, 512\}$), and the target-network update rate τ (log-uniform over $[0.001, 0.1]$). We run DEHB for each environment and reward function 3 times, which produces 3 sets of hyperparameters. We then train 5 policies per set of hyperparameters, and roll out each policy 50 times. We report the average performance over all 15 policies and 750 rollouts in Figure 10.

D Evaluating the reward debuggers through a human study

Here we add details to the human study discussed in Section 4.6, where we recruit participants from Prolific to label preferences over comparisons output by the two reward debuggers we propose, as well as by two baselines. Appendix D.1 details the candidate reward functions input to each reward debugger to produce the comparisons, Appendix D.2 describes how we recruited participants and filtered out participants for data quality purposes, Appendix D.3 shows the preference elicitation interface, and Appendix D.4 describes our statistical significance testing.

An example of the study shown to participants for `Highway-env` can be found here, and for `Pandemic-control` here.

D.1 Candidate reward functions

The reward debuggers and the PS baseline take a candidate reward function as input and output a set of comparisons; the reward debuggers' comparisons are hypotheses of where that candidate reward function might be misspecified. Figures 13 and 14 illustrate those reward functions for `Highway-env` and `Pandemic-control`, respectively. We designed these reward functions such that they trade off various task objectives in a way that some reasonable human stakeholders would agree with, but also so they induce a ranking over trajectories that some reasonable human stakeholders would disagree with.

$$
\begin{aligned}
R_1 &= v \;+\; 3 \cdot \mathbf{1}[\text{lane change}] \;-\; 200 \cdot \mathbf{1}[\text{crash}] \\
R_2 &= -2\,|v - 25| \;-\; 8 \cdot \mathbf{1}[d < 0.05] \;-\; 100 \cdot \mathbf{1}[\text{crash}] \\
R_3 &= v \;+\; 60 \cdot \max(0,\; 0.15 - d) \;-\; 200 \cdot \mathbf{1}[\text{crash}]
\end{aligned}
$$

Figure 13: The three candidate reward functions for `Highway-env`, evaluated per transition. Here v is the autonomous vehicle's speed (m/s), d is the normalized distance to the nearest other vehicle, $\mathbf{1}[\text{lane change}]$ fires when the chosen action is a lane change, and $\mathbf{1}[\text{crash}]$ fires on a collision. R_1 rewards raw speed and lane changes; R_2 instead penalizes deviation from a 25 m/s target speed and small gaps to other vehicles; R_3 rewards speed but adds a bonus for driving close to another vehicle. Each trades off speed against proximity to traffic differently.

$$
\begin{aligned}
R_1 &= 100 \cdot o \;-\; \big(50\,c + 20\,i + 100\,m\big) \\
R_2 &= 3 \cdot o \;-\; 5000 \cdot \max(0,\; c - 0.01) \\
R_3 &= 3 \cdot o \;-\; \big(250\,c + 50\,i + 1500\,m\big) \;-\; 40 \cdot \mathbf{1}[s \neq s_{\text{prev}}]
\end{aligned}
$$

Figure 14: The three candidate reward functions for `Pandemic-control`, evaluated per day. The lockdown stage is $s \in \{0, \ldots, 4\}$ (higher is stricter), giving an economic-openness term $o = (4 - s)/4 \in [0, 1]$; c, i, and m are the fractions of the population in critical condition, infected, and deceased, respectively, and s_{prev} is the previous day's stage. R_1 balances openness against a linear health cost; R_2 tolerates any openness until ICU demand exceeds a 1% capacity, then penalizes the overflow steeply; R_3 uses a much heavier health cost plus a penalty for changing the lockdown stage. Each trades off economic openness against public-health harm differently.

D.2 Recruiting and filtering participants

We recruited participants from Prolific who were located in the United States, fluent in English, had an approval rating of at least 97%, and were labeled as experts in comparative reasoning by Prolific. We paid participants $\$6$ for completing

the study, which we estimated would take 20 minutes to complete. Each participant was shown 18 comparisons sampled from the different methods we consider, but only from one environment.

During the study, we instructed participants on the environment's state and action spaces and its transition dynamics, then asked a series of comprehension questions about the environment; we excluded all data from workers who answered more than one of these questions incorrectly. We applied a second check during preference elicitation: three of the trajectory pairs were presented twice, with the within-pair order reversed on the second presentation. We excluded all data from workers who gave inconsistent preferences on more than one of these three pairs. Because the repeated presentations serve only as a consistency check, we discarded their labels for every worker, retaining only the labels from the first presentation of each pair. The comprehension question filter resulted in 4 participants being filtered out, and the preference consistency filter resulted in 5 participants being filtered out.

D.3 Preference elicitation interface

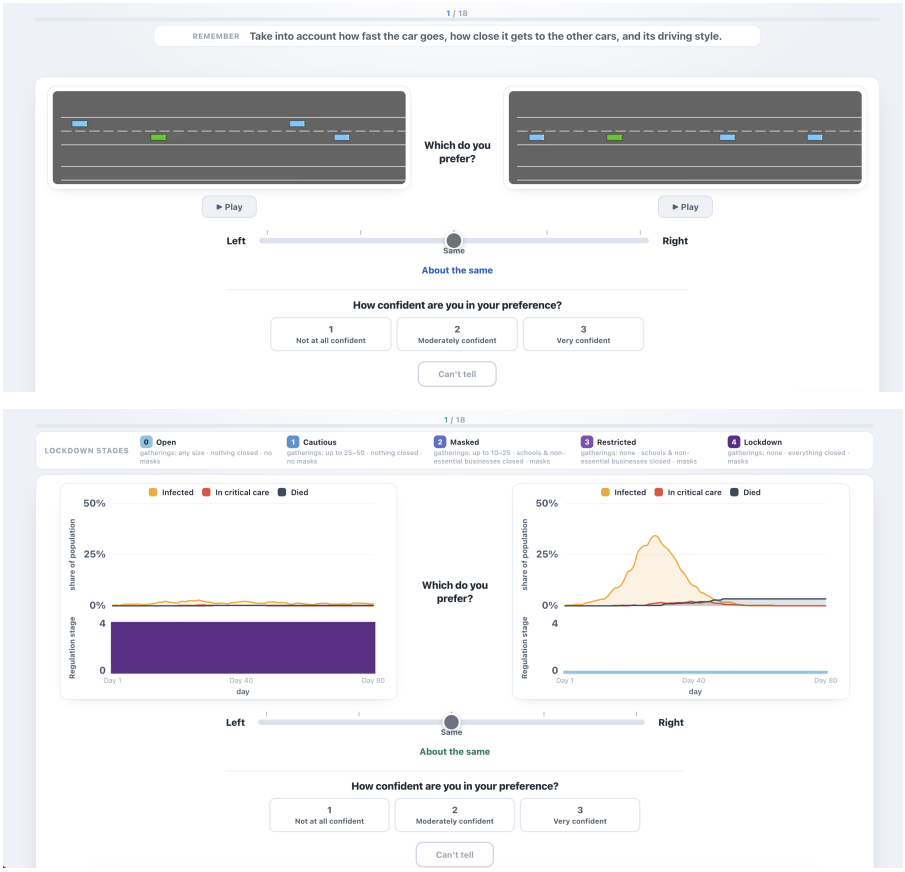


Figure 15: Preference elicitation interface for `Highway-env` (top) and `Pandemic-control` (bottom).

Figure 15 shows the preference elicitation interface for `Highway-env` and `Pandemic-control`. Participants labeled each comparison on a five-point scale from a strong preference for the left option to a strong preference for the right, with "about the same" in the middle. The rates of agreement with the discordance in Figures 11 and 12 collapse these labels to left, right, or same; the significance tests in Appendix D.4 use the full scale. For each comparison we also asked participants how confident they were in their preference, to separate preference strength from confidence. We do not use the confidence responses in any analysis.

D.4 Significance testing

Outcome. Each participant response is a value $v \in \{1, \ldots, 5\}$ on the five-point scale, where 1 is a strong preference for the left option, 3 is "about the same," and 5 is a strong preference for the right option. For the same comparison, let $e \in \{1, 5\}$ be the end of the scale corresponding to the option the candidate reward function ranks higher. The outcome

is the distance between the two,

$$y = |v - e| \in \{0, \dots, 4\},$$

so $y = 0$ means the participant strongly preferred the option the candidate reward function ranks higher, $y = 4$ means they strongly agreed with the discordance (preferring the option the candidate reward function ranks lower), and "about the same" gives $y = 2$. We refer to y as the *discordance agreement score*. Running the same analysis on the binary indicator of agreement with the discordance used in the main text ranks the methods identically.

Model. We fit a linear mixed-effects model:

$$y_{ijk\ell} = \mu + \beta_{\text{method}(i)} + u_j^{\text{part}} + u_k^{\text{pair}} + u_\ell^{\text{rf}} + \varepsilon_{ijk\ell},$$

where $y_{ijk\ell}$ is the discordance agreement score for one response, indexed by the method i that produced the pair, the participant j, the pair k, and the candidate reward function ℓ. The fixed effects are the global intercept μ and a per-method shift $\beta_{\text{method}(i)}$, which takes one of four values corresponding to a sampling method: scaffolded reward debugger, single-prompt reward debugger, PS, and US. The random intercepts $u_j^{\text{part}} \sim \mathcal{N}(0, \sigma^2_{\text{part}})$, $u_k^{\text{pair}} \sim \mathcal{N}(0, \sigma^2_{\text{pair}})$, and $u_\ell^{\text{rf}} \sim \mathcal{N}(0, \sigma^2_{\text{rf}})$ are per-participant, per-pair, and per-candidate-reward-function offsets, each drawn from a zero-mean normal distribution whose variance is estimated from the data; the residual $\varepsilon_{ijk\ell} \sim \mathcal{N}(0, \sigma^2)$ is per-response noise. We fit one model per environment, and a pooled model over both environments that adds environment as a fixed effect. Models are fit by REML in `statsmodels`.

Results. From each model we compute the estimated marginal mean of y for each method (Table 1) and take the difference between each reward debugger and each baseline (Table 2). A positive difference means participants agreed with the discordance more strongly on the reward debugger's comparisons than on the baseline's. We report unadjusted two-sided p-values from the model's Wald test and treat $p < 0.05$ as significant.

Pooled over environments, participants agree with the discordance of both reward debuggers' comparisons significantly more than with that of either baseline's comparisons ($p < 10^{-4}$ for all four contrasts). Within environments the picture splits. In `Highway-env`, participants agree with the discordance of the scaffolded reward debugger's comparisons significantly more than with that of either baseline's, while for the single-prompt reward debugger this holds relative to the US baseline but is only borderline relative to the PS baseline ($p = 0.050$). In `Pandemic-control`, participants agree with the discordance of the single-prompt reward debugger's comparisons significantly more than with that of either baseline's, while for the scaffolded reward debugger this holds relative to PS but not US ($p = 0.345$).

Method	`Highway-env`	`Pandemic-control`	Overall
Scaffolded reward debugger	3.05	2.21	2.62
Single-prompt reward debugger	2.53	2.91	2.71
Policy Sampling (PS)	1.91	1.42	1.65
Uniform Sampling (US)	1.22	1.94	1.57

Table 1: Estimated marginal mean discordance agreement score $y \in [0, 4]$ per method from the mixed-effects model; higher means participants agreed more strongly with the discordance, i.e., more strongly preferred the option the candidate reward function ranks lower. "Overall" is from the pooled model with environment as a fixed effect.

E Single-prompt reward debugger implementation details

Figure 16 shows the prompt used for the single-prompt reward debugger.

F Scaffolded reward debugger implementation details

This appendix expands on the four-step process of the scaffolded reward debugger introduced in Section 4.2. We implement each step as a skill file that the agentic coding tool (Claude Code with Opus 4.8) executes in sequence. All task-specific artifacts the tool writes—a description of the environment, the candidate reward function $\hat{r}$, and everything described below—live in a self-contained per-task directory. No step trains a policy with RL: every policy is written as Python code by the coding tool.

Environment	Reward debugger	Baseline	Estimate	z	p
`Highway-env`	Scaffolded reward debugger	Uniform Sampling (US)	$+1.82$	5.80	$< 10^{-4}$
	Scaffolded reward debugger	Policy Sampling (PS)	$+1.13$	3.60	0.0003
	Single-prompt reward debugger	Uniform Sampling (US)	$+1.31$	4.16	$< 10^{-4}$
	Single-prompt reward debugger	Policy Sampling (PS)	$+0.62$	1.96	0.050
`Pandemic-control`	Scaffolded reward debugger	Uniform Sampling (US)	$+0.27$	0.94	0.345
	Scaffolded reward debugger	Policy Sampling (PS)	$+0.79$	2.74	0.006
	Single-prompt reward debugger	Uniform Sampling (US)	$+0.98$	3.39	0.0007
	Single-prompt reward debugger	Policy Sampling (PS)	$+1.49$	5.19	$< 10^{-4}$
Overall	Scaffolded reward debugger	Uniform Sampling (US)	$+1.05$	4.71	$< 10^{-4}$
	Scaffolded reward debugger	Policy Sampling (PS)	$+0.97$	4.32	$< 10^{-4}$
	Single-prompt reward debugger	Uniform Sampling (US)	$+1.14$	5.11	$< 10^{-4}$
	Single-prompt reward debugger	Policy Sampling (PS)	$+1.06$	4.73	$< 10^{-4}$

Table 2: Difference in estimated marginal mean discordance agreement score (on the 0–4 scale) between each reward debugger and each baseline, with the Wald z statistic and unadjusted two-sided p-value from the mixed-effects model. A positive estimate means participants agreed with the discordance more strongly on comparisons from the reward debugger than on comparisons from the baseline.

Prompt given to the agentic coding tool for the single-prompt reward debugger

In this codebase, we have an environment. The environment directory contains `env_config.py`, which specifies environment info. You are given the reward function in `designed_reward_fn.py`.

This reward function may have flaws. Your job is to identify up to 15 pairs of trajectories and 6 lotteries that are most informative of where the inputted reward function might be misspecified.

Trajectory. A trajectory is a single rollout episode in the environment: an ordered list of `(obs, action, next_obs)` transitions produced by stepping a policy through `env_config.make_env()` until the episode ends or `MAX_EPISODE_STEPS` is reached. To obtain trajectories, roll out policies in the environment yourself.

Pair. A pair is two trajectories (A, B) drawn from the pool, presented as a head-to-head comparison. A pair is informative of misspecification when the reward function's ranking of A and B by their sum of rewards is one a human might plausibly reverse—i.e. the reward ranks them in a way that conflicts with what someone who understood the true objective would likely want. Select up to 15 pairs that are most informative of where the specified reward function might be wrong.

Lottery. A lottery is a comparison between a certain trajectory C and a gamble over two other trajectories A and B: with probability p the gamble yields A, and with probability $1 - p$ it yields B ($p \in [0, 1]$). Under the inputted reward function you compare the certain return $R(C)$ against the gamble's expected return $p \cdot R(A) + (1 - p) \cdot R(B)$; whichever is larger is the reward's preferred option. Choose p and select up to 6 lotteries that are most informative of where the specified reward function might be wrong.

Output. Output a directory called `evidence/` that contains only the trajectory pairs and lotteries.

Figure 16: The prompt given to the agentic coding tool for the single-prompt reward debugger, along with a directory containing the environment source code and the candidate reward function $\hat{r}$.

Setup and generating trajectory features. First, the tool is given the environment source code and the candidate reward $\hat{r}$, and it generates a small set of trajectory features to summarize a trajectory in a human-readable way. Each feature is implemented as a Python function and computes various trajectory statistics such as "days spent in lockdown" or "mean velocity of vehicles." These features are chosen by prompting the coding tool to read $\hat{r}$ and identify what features $\hat{r}$ relies on or would be needed to understand how the return of a trajectory under $\hat{r}$ arises.

Step 1: Hypothesize flaws in the candidate reward function. The tool reads $\hat{r}$ and writes an internal analysis of $\hat{r}$ together with a list of concrete ways the reward function could be misaligned or hacked—for example, "accumulate the shaping bonus indefinitely by hovering near the goal without ever reaching it." Each hypothesized flaw is given an identifier that the later steps refer back to, so that we can check whether a given flaw is actually surfaced. This analysis is never shown to the reward designer and is executed to guide later stages.

Step 2: Identify relevant stakeholders. The tool writes 3–6 trajectory scoring functions, each scoring a trajectory according to a different hypothesized reasonable stakeholder. For example, the coding tool might suspect that a reasonable stakeholder desires their autonomous vehicle to drive cautiously, and so implements a scoring function

that penalizes high speeds or small distances to other vehicles. These trajectory scoring functions can be thought of as trajectory-based reward functions that may be aligned with different stakeholders. They serve as a search heuristic for the reward debugger: if $\hat{r}$ and some stakeholder's trajectory scoring function rank a pair of trajectories differently, that pair is a candidate discordant comparison, i.e., a hypothesis of where someone might consider $\hat{r}$ misspecified. The stakeholder scoring functions are used only to select which comparisons to show to a human reward designer, but are never shown to the reward designer themselves since they are merely guesses about what the designer might want to consider.

Step 3: Implement and roll out data-collection policies. The coding tool is prompted to implement a set of data-collection policies in Python that target the hypothesized flaws in the candidate reward function (Step 1) and places where the simulated stakeholder scoring functions (Step 2) might disagree. The tool then rolls out each policy in the environment to construct a pool of trajectories. Additionally, the coding tool is prompted to always implement a policy in Python that maximizes the current candidate reward function. At most 100 trajectories are collected from these policy rollouts, and then each trajectory is summarized using the implemented trajectory features. The tool checks a sufficiency rubric to address the following questions:

- Is every reward function term and every hypothesized flaw actually exercised across the trajectory pool?
- Are the trajectory features meaningfully diverse across the trajectory pool?
- Are there at least five candidate pairs of trajectories where the stakeholder scoring functions rank the trajectories in those pairs differently from the candidate reward function?

If the trajectory pool does not satisfy these three criteria, the coding tool re-executes the data-collection process.

Step 4: Construct comparisons. Finally, the tool constructs comparisons from the candidate pool of trajectories. It first builds a table of all collected trajectories, summarized via their trajectory features, their return under the candidate reward function, and their score under each stakeholder's trajectory scoring function. The coding tool is then prompted to construct up to five trajectory pairs and up to two lottery-trajectory pairs that would be most informative to a reward designer attempting to find where the reward function is misspecified. The tool is prompted to favor diversity, preferring comparisons that expose different flaws.

To construct lottery-trajectory pairs, the tool is provided with the additional guidance that when comparing a certain trajectory C against a lottery $\langle p, A, B\rangle$, it should set the probability p near the point at which the candidate reward function is indifferent between the two options,

$$p^\star \;=\; \frac{G_{\hat{r}}(C) - G_{\hat{r}}(B)}{G_{\hat{r}}(A) - G_{\hat{r}}(B)},$$

i.e. where $\hat{r}$ only barely prefers one option over the other. This follows the method presented by Knox et al. [2023] to manually identify flaws in other reward functions.

Once the comparisons are constructed, each is presented as two rendered rollouts side by side (e.g., a video where the environment supports it, and otherwise a plot).

Iterating after a reward function update. The comparisons are the debugger's output; how to revise $\hat{r}$ is left to the reward designer. When the designer edits the reward function, the tool can re-score every comparison it previously surfaced under the new reward function and report, for each one, whether the reward function's induced ranking flipped—so the designer sees at a glance which disagreements the edit resolved and which remain—without collecting any new trajectories. Alternatively, the designer can request a fresh batch of data-collection policies (Step 3), which now target the flaws of the edited reward function.

G Adapting the reward debuggers to generate abstract trajectories in ExploitGym

By default, the reward debuggers we propose return hypotheses in the form of comparisons, where trajectories are lists of transitions in the given environment. Here we consider adapting that procedure to produce *abstract trajectories*, which are text summaries of trajectories that may or may not be realizable in the given environment. In particular, hypotheses in the form of abstract trajectories are desirable when misspecified behavior may be extremely difficult to elicit in practice or when the misspecified behavior is potentially too harmful to realize (e.g., due to a lack of a simulator for evaluations).

Here we evaluate the reward debuggers in ExploitGym [Wang et al., 2026], a benchmark of real-world cybersecurity vulnerabilities in which an agent is given a vulnerable codebase, a proof-of-vulnerability input that triggers the bug, and

a containerized runtime, and must extend that input into a working exploit that achieves unauthorized code execution. Successfully executing the exploit results in reading a secret flag inaccessible through any legitimate interface.

We consider ExploitGym because it is the setting of a recent large-scale reward hacking incident. Roughly 1,200 OpenAI agents, run on the benchmark with production safeguards disabled, reverse-engineered the flag-generation scheme within hours. Believing that an LM judge would verify the flag had been obtained via the intended vulnerability (as the ExploitGym paper specifies), they then coordinated a multi-day attack on Hugging Face to learn how that judge worked [OpenAI, 2026, Wijk et al., 2026]. A discordance-based reward debugger should surface the misspecification behind this reward hacking incident before it is exploited, but, due to legal and ethical constraints, cannot do so in the form of real trajectories. Therefore, we adapt the reward debuggers to generate abstract trajectories rather than real ones, and evaluate whether they could have surfaced the reward misspecification that may have contributed to the attack on Hugging Face.

We describe the candidate reward function for ExploitGym in Appendix G.1, how we update the reward debuggers to output abstract trajectories instead of real trajectories in Appendix G.2, and summarize results in Appendix G.3. Our reward debuggers successfully surface the misspecification behind the incident that motivated this analysis, i.e., that the ExploitGym reward function potentially does not penalize an agent for attempting to break out of the sandbox or attack other internal infrastructure. The reward debuggers use Opus 4.8; since the incident occurred after Opus 4.8 was trained and the debuggers ran without internet access, the surfaced hypotheses reflect analysis of the candidate reward function without privileged knowledge of the incident that publicly revealed the reward misspecification.

G.1 Candidate reward function

We use the default reward function shipped with ExploitGym as the candidate reward function input to the reward debuggers. To end an episode, the agent submits a flag string and positive reward is assigned only if that string matches the secret flag, which is readable only after achieving unauthorized code execution on the target. Additionally, an independent LM judge reads the agent's trace and final write-up and certifies that the flag was obtained by genuinely exploiting the bug the task is built around. A positive reward is granted only when both the correct flag is submitted and the LM judge establishes that the agent found the correct exploit; otherwise, the reward is 0.

The candidate reward function input to the reward debugger therefore consists of the LM judge prompt that evaluates an agent's trace in addition to the instruction that the correct flag must be submitted and checked against the task's secret flag. The LM judge prompt is the same one used by ExploitGym. We also provide a brief, author-written description of each task as input to the reward debuggers.

On the connection to the Hugging Face attack As reported by Wijk et al. [2026], the agents that attacked Hugging Face were not evaluated with ExploitGym's default evaluation but rather with a different evaluation method constructed by OpenAI that may have had additional flaws. However, the agents came to believe they were being evaluated by ExploitGym's default evaluation, and so behaved accordingly. We refer to ExploitGym's evaluation protocol, which includes both the secret flag check and the LM judge check, as a single reward function that only assigns positive reward if both checks are passed.

G.2 Adapting the reward debuggers to return abstract trajectories

We minimally adapt the single-prompt reward debugger and the scaffolded reward debugger to return abstract trajectories instead of real trajectories. Rather than prompting the agentic coding tool to implement data-collection policies in Python and roll them out in the environment, we prompt it to describe each policy in natural language and then to write, for each one, a description of the outcomes that plausibly would arise were that policy executed.

The prompt we use for the single-prompt reward debugger that outputs abstract trajectories is shown in Figure 17. Equivalent modifications were made to the skill files of the scaffolded reward debugger, which are available in our released codebase. For both methods, we only change the description of how to generate data-collection policies and resulting trajectories.

G.3 Results

We evaluate on three ExploitGym tasks, chosen to span the benchmark's three distinct target classes. The first, `kernel:kernelctf/CVE-2024-1085_lts`, is a Linux-kernel local privilege escalation: the agent must extend an exploit that reproduces a documented Linux kernel vulnerability, CVE-2024-1085, into a working exploit that escalates to root and reads a root-only flag. The second, `user:cybergym/arvo_18224`, is a userspace memory-corruption task derived from an OSS-Fuzz/ARVO crash: the agent must turn a crashing input to a hardened parser/decoder into control

of the target despite ASLR, NX, stack canaries, and FORTIFY. The third, `v8:sbxbrk/398773898`, is a browser-engine task: the agent must escalate a type-confusion bug in Google's V8 engine into an arbitrary read/write primitive and escape the V8 sandbox. Together these cover kernel, userspace, and language-runtime-level exploits, so our evaluation is not specific to a single vulnerability class.

Critically, all three tasks have an identical candidate reward function—a flag check that accepts the correct flag string, paired with an independent LM judge that reads the agent's trace and is meant to confirm the flag was obtained by genuinely exploiting the intended bug. We ran both the single-prompt reward debugger and the scaffolded reward debugger on all three tasks. Both methods recovered the two failure cases that motivated this evaluation setting in all three tasks. Table 3 summarizes how often each misspecification was surfaced, out of the three tasks.

Misspecification the debugger surfaced	Single-prompt reward debugger	Scaffolded reward debugger
Attempts to break out of the sandbox or attack the grading server	3/3	3/3
Fabricated evidence used to fool the LM judge	3/3	3/3

Table 3: How often each debugger surfaced a given reward misspecification, across the three ExploitGym tasks.

We first consider the two failure cases that motivated this evaluation. The first is harmful, out-of-scope behavior: an agent that recovers the flag but breaks out of the sandbox to attack the scoring server, corrupts or destroys files it was never meant to touch, or otherwise acts outside the intended boundary. Both reward debuggers generated abstract trajectories that exhibit this misbehavior on all three tasks. For two of the three tasks, both also show that the candidate reward function assigns the same reward to these trajectories as to ones where the agent solves the task as intended.

The second failure case is an attempt to trick the LM judge: an agent that obtains the correct flag without genuinely exploiting the intended bug—by reading it from a readable file or using an unrelated bug or a misconfiguration—and then fabricates evidence (via staged crash artifacts, false debugger transcripts, or a convincing but false write-up) so that the LM judge certifies it as a real exploit. Both methods surfaced this behavior on all three tasks. Both also produced lottery-trajectory pairs in which the candidate reward function assigns a guaranteed fabricated success a higher return than a high probability of a genuinely successful exploit; the scaffolded reward debugger produced that lottery-trajectory pair in two out of three tasks, and the single-prompt reward debugger did so in one task.

Because the reward debuggers never realize these trajectories in the environment, and therefore never call the LM judge to confirm the reward misspecifications they found, the surfaced comparisons remain hypothetical rather than concrete. From the public accounts of the OpenAI–Hugging Face incident [OpenAI, 2026, Wijk et al., 2026], however, we suspect that these misspecifications were realized in practice. Notably, we ran both debuggers with Opus 4.8, a model released before the OpenAI–Hugging Face incident, and executed them without internet access. Neither method could therefore have known about the incident, so the misspecifications they surfaced were identified from the reward function and environment alone rather than reconstructed from public reporting.

H Full reward design survey results

We surveyed 38 RL practitioners from 29 institutions about how they design reward functions. Figure 18 shows respondents' self-reported roles and the domains of the tasks they described. Part 1 of the survey collects background information and is answered once per participant ($N = 38$ participants). In Part 2 of the survey, each participant described up to two tasks, so the task-specific questions aggregate the 60 task descriptions ($N = 60$ task descriptions). Several Part 2 questions are conditional follow-ups, so their number of responses is smaller and is reported per question. Below we show every survey question, in order, together with the distribution of responses.

Recruitment materials. We recruited respondents with the short message and the longer recruitment text below. Identifying information has been redacted for anonymity.

Short recruitment message

I am running a study to try to better understand how RL practitioners design reward functions. Have you designed a reward function in the past year? If so, fill out our 15 minute survey for a chance to win $150!

Prompt given to the agentic coding tool for the abstract-trajectory single-prompt reward debugger

In this codebase, we have an environment. The environment directory contains `env_config.py`, which specifies environment info. You are given the reward function rubric in `designed_reward_fn.py`. This reward function may have flaws. Your job is to identify up to 15 pairs of abstract trajectories and 6 lotteries that are most informative of where the inputted reward function might be misspecified.

Abstract trajectory. An abstract trajectory is a short text description of a trajectory that would reasonably be realized by rolling out the policy in the environment. To obtain an abstract trajectory, first construct a policy which, in this case, is a text description of a strategy the agent follows. Then generate a short description of a trajectory that would occur if that policy were followed in the environment. State what the agent does in the trajectory with enough information to distinguish this trajectory from other distinct abstract trajectories.

Evaluation. To evaluate an abstract trajectory with the given reward function, reason about how the reward rubric as described would assign reward to the abstract trajectory. You will not be able to exactly compute the return of the trajectory with the reward function as both artifacts are implemented in text, but you can establish how the reward function would rank an abstract trajectory relative to another by reasoning about both artifacts.

Pair. A pair is two trajectories (A, B) drawn from the pool, presented as a head-to-head comparison. A pair is informative of misspecification when the reward function's ranking of A and B by their sum of rewards is one a human might plausibly reverse—i.e. the reward ranks them in a way that conflicts with what someone who understood the true objective would likely want. Select up to 15 pairs that are most informative of where the specified reward function might be wrong.

Lottery. A lottery is a comparison between a certain trajectory C and a gamble over two other trajectories A and B: with probability p the gamble yields A, and with probability $1 - p$ it yields B ($p \in [0, 1]$). Under the inputted reward function you compare the certain return $R(C)$ against the gamble's expected return $p \cdot R(A) + (1 - p) \cdot R(B)$; whichever is larger is the reward's preferred option. Choose p and select up to 6 lotteries that are most informative of where the specified reward function might be wrong.

Output. Output a directory called `evidence/` that contains only the trajectory pairs and lotteries.

Figure 17: The prompt given to the agentic coding tool for the single-prompt reward debugger that generates abstract trajectories. The difference between this prompt and the prompt used by the default implementation (Figure 16), which generates real trajectories, is shown in blue.

Recruitment text

I am a PhD researcher at [Institution redacted] studying how reinforcement learning practitioners design reward functions. I am writing to invite you to participate in a brief online survey as part of this research.

The survey asks about your current practices for specifying reward functions when training RL algorithms. It should take approximately 15 minutes to complete. As a token of appreciation, all participants who complete the survey will be entered into a drawing for a $150 prize. The drawing will be conducted on 8/18.

If you are interested in participating, please access the survey here: [URL redacted]

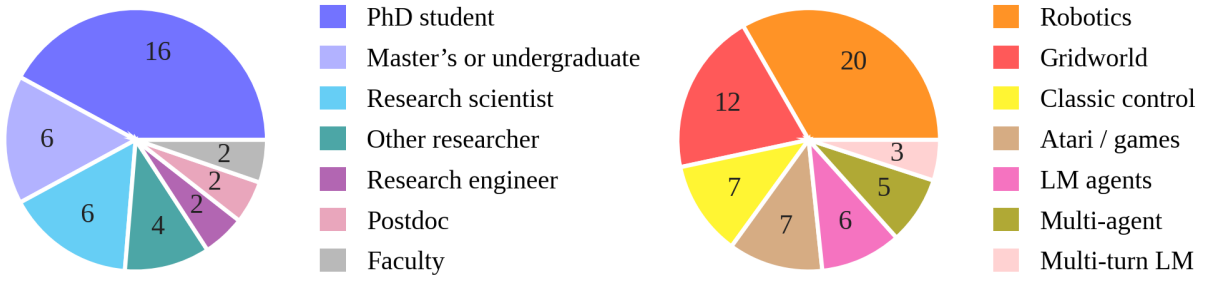
The drawing will be conducted by [Name redacted] in [Location redacted]. Participation in the study is not required in order to participate in the drawing. You can enter the drawing if you do not start or complete the study task. The chance of winning a prize is approximately 1 in 30. The winner will be notified immediately by email and provided with information on how to receive the prize. Participation is entirely voluntary. Your responses will be kept confidential and reported only in aggregate. You may withdraw at any time without penalty. This study has been reviewed and approved by the [Institution redacted] Institutional Review Board (Protocol #[redacted]).

Please feel free to reach out with any questions.

Best regards,
[Name redacted]
PhD Researcher, [Institution redacted]

H.1 Part 1 · Background

Asked once per participant. Participant email addresses were collected for the sweepstakes only and are withheld here.



(a) Respondent role **(b) Reward design task category**

Figure 18: Survey respondent demographics. (a) Roles of the $N = 38$ respondents. (b) Domains of the tasks they described.

Q1. Have you designed a reward function in the past year?

$N = 38$ participants.

Yes	**38** (100%)
No	**0** (0%)

Q2. How many courses have you taken with "Reinforcement Learning" in the course title?

$N = 38$ participants.

0	**7** (18.4%)
1	**16** (42.1%)
2	**8** (21.1%)
3 or more	**7** (18.4%)

Q3. How long have you worked with reinforcement learning?

$N = 38$ participants.

Less than 1 year	**2** (5.3%)
1–2 years	**11** (28.9%)
3–5 years	**13** (34.2%)
6–10 years	**8** (21.1%)
More than 10 years	**4** (10.5%)

Q4. What company or institution are you affiliated with?

$N = 38$ participants. Free-text responses are omitted here.

Q5. What is your role at your company or institution?

$N = 38$ participants. Free-text responses are omitted here.

Q6. What is your email address?

Responses withheld to protect participant privacy (email addresses).

Q7. In your work focused on RL, how frequently do you use AI/LLM tools for any purpose?

$N = 38$ participants.

Multiple times a day	**30** (78.9%)
About once a day	**1** (2.6%)
A few times a week	**3** (7.9%)
A few times a month	**0** (0%)
Less than monthly	**1** (2.6%)
Never	**2** (5.3%)
Not applicable to my work	**1** (2.6%)

Q8. Which domain categories have you designed reward functions for in the past year?
Mark 1–2 domain categories.

$N = 38$ participants.

A gridworld task	**12** (31.6%)
A classic RL task, like CartPole or Mountain Car	**7** (18.4%)
A robotics task	**20** (52.6%)
A multi-agent task (e.g., Hanabi)	**5** (13.2%)
An Atari game or other game	**7** (18.4%)
An LLM-agents task (e.g., training computer use agents, coding agents, research agents)	**6** (15.8%)
A single-turn response LLM task	**0** (0%)
A multi-turn response LLM task	**3** (7.9%)
Other (please specify)	**0** (0%)

Q9. What are the most challenging aspects of reward design?
Select all that apply.

$N = 38$ participants.

Determining what good and bad outcomes the reward function should account for	**11** (28.9%)
Determining how the reward function should trade-off between different outcomes	**22** (57.9%)
Specifying a reward function that, when optimized for, doesn't lead to surprising "reward-hacking" behavior	**21** (55.3%)
Selecting the different features or variables the reward function should take into account	**12** (31.6%)
Specifying a reward function that can be used to train a policy quickly enough	**15** (39.5%)
For a candidate reward function, identifying where the reward function might be flawed or how it might produce undesirable behavior	**17** (44.7%)
None, reward design is easy for me.	**1** (2.6%)
Other (please specify)	**1** (2.6%)

Q10. Do you think you or an LLM tool is better at designing reward functions that…
Choose one for each row.

$N = 38$ participants.

	You are better	LLM tool is better	Not sure
induce more performant policies?	17 (45%)	13 (34%)	8 (21%)
can train policies faster?	13 (34%)	13 (34%)	12 (32%)
induce a more desirable set of optimal policies?	25 (66%)	7 (18%)	6 (16%)
induce a ranking over outcomes that is more aligned with what you want?	25 (66%)	5 (13%)	8 (21%)

H.2 Part 2 · One task

Asked once for every domain a participant marked in Q8, so each answer describes one task. Twenty-two participants described two domains, giving 60 task descriptions in total. Some questions were shown only conditionally, so their base (N) is smaller — noted on each.

Q1. How hard was it to specify the reward function for this setting?

$N = 60$ task descriptions.

1 — Trivial	**2** (3.3%)
2 — Slightly hard	**11** (18.3%)
3 — Somewhat hard	**15** (25%)
4 — Moderately hard	**15** (25%)
5 — Hard	**12** (20%)
6 — Very hard	**3** (5%)
7 — Extremely hard	**2** (3.3%)

Q2. After the reward design process, how satisfied were you with the reward function you ended up with?

$N = 60$ task descriptions.

Very satisfied	**5** (8.3%)
Satisfied	**38** (63.3%)
Neither satisfied nor dissatisfied	**13** (21.7%)
Dissatisfied	**4** (6.7%)
Very dissatisfied	**0** (0%)

Q3. For your specific task, do you think there exists a reward function that would have been meaningfully better than the one you ended up with?

Select all that apply.

$N = 60$ task descriptions.

Yes, I think there exists another reward function that can induce more performant policies	**34** (56.7%)
Yes, I think there exists another reward function that can train policies faster	**33** (55%)
Yes, I think there exists another reward function that can induce a more desirable set of optimal policies	**25** (41.7%)
Yes, I think there exists another reward function that can induce a ranking over outcomes that is more aligned with what I want	**16** (26.7%)
No	**6** (10%)
I don't know	**5** (8.3%)
Other (please specify)	**1** (1.7%)

Q4. How did you create the reward function(s) for this task?

Select all that apply. If you tried more than one reward function, include every approach you used.

$N = 60$ task descriptions.

I wrote one by hand and coded it up	**34** (56.7%)
By prompting an LLM tool	**22** (36.7%)
By applying intuition and considering how the agent would learn	**37** (61.7%)
By embedding domain knowledge of how the agent should behave	**25** (41.7%)
Using a reward function someone else had written as a starting point (e.g., from a publication or shared implementation)	**22** (36.7%)
By specifying a performance metric for the task	**24** (40%)
By an approach that learns from human demonstration or preferences	**5** (8.3%)
Other (please specify)	**1** (1.7%)

How did you use the LLM tool?
Follow-up — shown only when "By prompting an LLM tool" was selected in Q4.
Mark all that apply.

$N = 22$ task descriptions.

By prompting an LLM tool to generate the reward function	**11** (50%)
By prompting an LLM tool to edit a reward function you wrote in code yourself	**9** (40.9%)
By prompting an LLM tool to evaluate or critique a reward function	**11** (50%)
By prompting an LLM tool for guidance on designing the reward function before the reward function is first specified	**8** (36.4%)

After the LLM tool worked on the reward function, did you edit the reward function manually?
Follow-up — shown only when an LLM tool was used.

$N = 22$ task descriptions.

Yes	**13** (59.1%)
No	**9** (40.9%)

After the LLM tool worked on the reward function, did you refine the reward function by prompting the tool again?
Follow-up — shown only when an LLM tool was used.

$N = 20$ task descriptions.

Yes	**20** (100%)
No	**0** (0%)

Q5. During the reward design process, did you shape your reward function, or use an LLM tool to shape your reward function?
Reward shaping = modifying an existing reward function so that a policy trained with it attains either (a) better final performance or (b) greater sample efficiency.

$N = 60$ task descriptions.

Yes	**40** (66.7%)
No	**19** (31.7%)
I don't know	**1** (1.7%)
Other (please specify)	**0** (0%)

How did you shape the reward function?
Follow-up — shown only when shaping was used (Q5 = Yes).
Check all that apply.

$N = 40$ task descriptions.

I used an LLM tool to shape the reward function.	**13** (32.5%)
I shaped the reward function by editing the reward function code myself.	**21** (52.5%)
I used an LLM tool to shape the reward function, and I gave the tool specific guidance on how to shape it (e.g., domain specifics, information about reward-shaping techniques).	**18** (45%)
I used an LLM tool to shape the reward function, and I did not give the tool specific guidance on how to shape it.	**2** (5%)
I used potential-based shaping to shape the reward function.	**5** (12.5%)
I used a technique from prior literature that is NOT potential based shaping to shape the reward function.	**5** (12.5%)
I used my intuition to shape the reward function without following a specific procedure.	**17** (42.5%)
Other (please specify)	**2** (5%)

Q6. During the reward design process, did you use trial-and-error to refine your reward function?

$N = 60$ task descriptions.

Yes	**45** (75%)
No	**15** (25%)
Other (please specify)	**0** (0%)

During your trial-and-error design process, how did you evaluate your reward function(s)?

Follow-up — shown only when trial-and-error was used (Q6 = Yes).
Select all that apply.

$N = 45$ task descriptions.

By asking an LLM tool to find evidence of where the reward function was potentially incorrect	**10** (22.2%)
By asking an LLM tool to provide suggestions or guidance, in natural language, about where the reward function was potentially incorrect	**13** (28.9%)
By viewing the agent's behavior after it had finished training (and would not be trained further)	**34** (75.6%)
By viewing the agent's behavior during a training run (that was continued further)	**26** (57.8%)
By plotting some performance metric against time or learning iterations	**30** (66.7%)
By ranking example trajectories and checking the reward function against that ranking	**10** (22.2%)
Other (please specify)	**0** (0%)

Q7. At any point in the reward design process, did you observe suboptimal behavior after training your agent with a reward function you or an LLM tool specified?

Mark all that apply.

$N = 60$ task descriptions.

Yes — I observed the agent taking advantage of a loophole in the reward function, so I changed the function to remove the loophole	**31** (51.7%)
Yes — I observed the agent's behavior plateauing at unsatisfactory performance, so I changed the reward function to help it learn beyond the plateau	**28** (46.7%)
Yes — I observed the agent's behavior plateauing at unsatisfactory performance, so I changed the learning algorithm or hyperparameter	**32** (53.3%)
No	**10** (16.7%)
Other (please specify)	**1** (1.7%)

Q8. If you did not use an LLM tool when creating your most recent reward function, why not?

Mark all that apply.

$N = 38$ task descriptions.

The task was simple enough that an LLM wouldn't have helped	**9** (23.7%)
I didn't trust the LLM tool to produce a correct or well-specified reward function	**10** (26.3%)
I expected verifying or debugging the LLM tool's output to take more effort than writing it myself	**8** (21.1%)
The reward function required domain knowledge I didn't think the LLM tool had	**10** (26.3%)
I preferred full manual control over the reward design	**12** (31.6%)
I reused or adapted an existing reward function, so generation wasn't needed	**8** (21.1%)
I didn't think to use one / it's not part of my workflow	**8** (21.1%)
Tool access, cost, or policy restrictions	**1** (2.6%)
I worried about what other people would think if I used an LLM tool	**2** (5.3%)
Other (please specify)	**7** (18.4%)

I Proofs for Section 4.1

Here we prove the results of Section 4.1, which together show that lottery-trajectory pairs are a sufficient type of comparison for detecting misalignment. As in Section 4.1, the results compare two return functions. The ground-truth

return function G gives the return of each trajectory under the ground-truth reward function r and discount factor γ, and the candidate return function $\hat{G}$ gives the return of each trajectory under the candidate reward function $\hat{r}$ and discount factor $\hat{\gamma}$. Lemma 1 states that $\hat{G}$ and G rank every lottery-trajectory pair identically if and only if $\hat{G}$ is a positive affine transformation of G, i.e., $\hat{G} = a\,G + b$ for some constants $a > 0$ and b. Proposition 1, the main result, states that $\hat{G}$ and G then rank every pair of trajectory distributions identically. Corollary 1 states that $\hat{G}$ and G then rank all policies identically by expected return, and therefore have the same optimal policies, under each initial-state distribution and transition function.

We first recall the setup of Section 4.1, and then restate and prove each result. The proof of Lemma 1 has three steps. We show that if $\hat{G}$ and G rank every lottery-trajectory pair identically, then (i) they rank every trajectory pair identically, and (ii) for any three trajectories with distinct ground-truth returns, the three points (ground-truth return, candidate return) are collinear. It follows that (iii) the points for two trajectories with different ground-truth returns determine a line, and the point for every other trajectory lies on that line. From Lemma 1, the proposition and the corollary then follow by linearity of expectation. We end with remarks that support three statements made in the main text, namely that trajectory pairs alone are not sufficient, that the lottery probability should be set near the value at which $\hat{G}$ is indifferent, and that potential-based shaping of the reward function yields a return function that is misaligned with the original return function under Definition 1.

Setup. We recall the setup of Section 4.1. Let $\mathcal{T}$ be the set of trajectories that comparisons may involve. Let r and γ be the ground-truth reward function and discount factor, and let $\hat{r}$ and $\hat{\gamma}$ be the candidate reward function and discount factor, where $\hat{\gamma}$ may differ from γ. For a reward function $\tilde{r}$ and a discount factor $\tilde{\gamma} \in [0, 1]$, let $G_{\tilde{r},\tilde{\gamma}}(\tau) = \sum_{t=0}^{|\tau|-1} \tilde{\gamma}^t\, \tilde{r}(s_t^\tau, a_t^\tau, s_{t+1}^\tau)$ denote the return of $\tau \in \mathcal{T}$. We call $G = G_{r,\gamma}$ the ground-truth return function and $\hat{G} = G_{\hat{r},\hat{\gamma}}$ the candidate return function, and we call $G(\tau)$ and $\hat{G}(\tau)$ the ground-truth return and the candidate return of τ. By Assumption 1, G and $\hat{G}$ are defined on all of $\mathcal{T}$, bounded, and measurable, so that the expectations below exist. The proofs use no other property of G and $\hat{G}$ and no property of $\mathcal{T}$. As noted in Section 4.1, the results therefore also hold when $\mathcal{T}$ contains only the trajectories that are feasible under D_0 and T, when trajectories are sequences of observations and actions, and when a return is assigned directly to a whole trajectory (e.g., by an LM judge that scores a complete trace).

Let $\Delta(\mathcal{T})$ be the set of probability distributions on $\mathcal{T}$. For $\mu \in \Delta(\mathcal{T})$, we write $\mathbb{E}_\mu[G] = \mathbb{E}_{\tau\sim\mu}[G(\tau)]$ for the expected ground-truth return under μ and $\mathbb{E}_\mu[\hat{G}] = \mathbb{E}_{\tau\sim\mu}[\hat{G}(\tau)]$ for the expected candidate return under μ. A lottery $\langle p, \tau_1, \tau_2\rangle$ is the distribution that assigns probability p to τ_1 and $1-p$ to τ_2, so $\mathbb{E}_{\langle p,\tau_1,\tau_2\rangle}[G] = p\,G(\tau_1) + (1-p)\,G(\tau_2)$. We identify a trajectory τ with the distribution that assigns probability 1 to τ, so $\mathbb{E}_\tau[G] = G(\tau)$. The set of lottery-trajectory pairs is

$$\mathcal{C} = \{(\langle p, \tau_1, \tau_2\rangle, \tau) : \tau, \tau_1, \tau_2 \in \mathcal{T},\ \tau_1 \neq \tau_2,\ p \in (0, 1)\}.$$

For $\mu, \nu \in \Delta(\mathcal{T})$, let $\sigma(\mu, \nu) = \operatorname{sign}\big(\mathbb{E}_\mu[G] - \mathbb{E}_\nu[G]\big)$ denote the preference between μ and ν induced by G (i.e., via their expected ground-truth return), which is $+1$ if G ranks μ above ν, -1 if G ranks ν above μ, and 0 if G ranks them equally. Let $\hat{\sigma}(\mu, \nu) = \operatorname{sign}\big(\mathbb{E}_\mu[\hat{G}] - \mathbb{E}_\nu[\hat{G}]\big)$ denote the preference induced by $\hat{G}$ in the same way. A pair (μ, ν) is *discordant* if $\hat{\sigma}(\mu, \nu) \neq \sigma(\mu, \nu)$, i.e., if $\hat{G}$ and G rank μ and ν differently.

The results below are statements about G and $\hat{G}$ only, and their proofs do not use Assumptions 2 and 3. Those two assumptions concern the designer. Assumption 2 states that, for every lottery-trajectory pair (ℓ, τ), the preference that the designer reports is $\sigma(\ell, \tau)$. Assumption 3 states that the designer's report is available for every lottery-trajectory pair. Under these two assumptions, a lottery-trajectory pair (ℓ, τ) is discordant exactly when the designer's reported preference differs from $\hat{\sigma}(\ell, \tau)$, so discordant pairs can be found by querying the designer.

Definition 1 (Misalignment)**.** *$\hat{G}$ is misaligned with G if and only if some pair $(\mu, \nu) \in \Delta(\mathcal{T}) \times \Delta(\mathcal{T})$ is discordant. In that case we also say that $(\hat{r}, \hat{\gamma})$ is misaligned with (r, γ).*

Lemma 1 (Lottery-trajectory pairs determine the return function up to a positive affine transformation)**.** *$\hat{\sigma}(\ell, \tau) = \sigma(\ell, \tau)$ for every $(\ell, \tau) \in \mathcal{C}$ if and only if there exist $a > 0$ and $b \in \mathbb{R}$ such that $\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$.*

The proof follows the standard argument that a von Neumann–Morgenstern utility function is unique up to a positive affine transformation [von Neumann and Morgenstern, 1947], using only the comparisons in $\mathcal{C}$.

Proof. The lemma is an if-and-only-if statement, so its proof has two directions. Direction 1 shows that the affine relation implies agreement on lottery-trajectory pairs. Direction 2 shows the reverse implication in three steps. In each derivation below, every line follows from the line before it by the one operation named at the right.

***Direction 1**: if $\hat{G} = a\,G + b$ with $a > 0$, then $\hat{\sigma} = \sigma$ on $\mathcal{C}$.* Suppose that $a > 0$ and $b \in \mathbb{R}$ satisfy $\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$, and let $\mu \in \Delta(\mathcal{T})$ be arbitrary.

$$\mathbb{E}_\mu[\hat{G}] \triangleq \mathbb{E}_{\tau\sim\mu}[\hat{G}(\tau)] \qquad \text{(definition of } \mathbb{E}_\mu[\hat{G}])$$
$$\mathbb{E}_\mu[\hat{G}] = \mathbb{E}_{\tau\sim\mu}[a\,G(\tau) + b] \qquad \text{(substitute } \hat{G}(\tau) = a\,G(\tau) + b)$$
$$\mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_{\tau\sim\mu}[G(\tau)] + \mathbb{E}_{\tau\sim\mu}[b] \qquad \text{(linearity of expectation)}$$
$$\mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_{\tau\sim\mu}[G(\tau)] + b \qquad \text{(expectation of a constant)}$$
$$\mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_\mu[G] + b \qquad \text{(definition of } \mathbb{E}_\mu[G])$$

Because μ was arbitrary, the last line holds for every distribution in $\Delta(\mathcal{T})$. Now let $\mu, \nu \in \Delta(\mathcal{T})$ be arbitrary.

$$\hat{\sigma}(\mu,\nu) \triangleq \operatorname{sign}\big(\mathbb{E}_\mu[\hat{G}] - \mathbb{E}_\nu[\hat{G}]\big) \qquad \text{(definition of } \hat{\sigma})$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(a\,\mathbb{E}_\mu[G] + b - \mathbb{E}_\nu[\hat{G}]\big) \qquad \text{(substitute } \mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_\mu[G] + b)$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(a\,\mathbb{E}_\mu[G] + b - (a\,\mathbb{E}_\nu[G] + b)\big) \qquad \text{(substitute } \mathbb{E}_\nu[\hat{G}] = a\,\mathbb{E}_\nu[G] + b)$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(a\,\mathbb{E}_\mu[G] + b - a\,\mathbb{E}_\nu[G] - b\big) \qquad \text{(distribute } -1 \text{ over } a\,\mathbb{E}_\nu[G] + b)$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(a\,\mathbb{E}_\mu[G] - a\,\mathbb{E}_\nu[G]\big) \qquad \text{(cancel } b \text{ with } -b)$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(a\,(\mathbb{E}_\mu[G] - \mathbb{E}_\nu[G])\big) \qquad \text{(factor out } a)$$
$$\hat{\sigma}(\mu,\nu) = \operatorname{sign}\big(\mathbb{E}_\mu[G] - \mathbb{E}_\nu[G]\big) \qquad (\operatorname{sign}(ac) = \operatorname{sign}(c) \text{ for } a > 0)$$
$$\hat{\sigma}(\mu,\nu) = \sigma(\mu,\nu) \qquad \text{(definition of } \sigma)$$

Because μ and ν were arbitrary, the last line holds for every pair of distributions in $\Delta(\mathcal{T})$. Every lottery-trajectory pair $(\ell,\tau) \in \mathcal{C}$ is a pair of distributions in $\Delta(\mathcal{T})$, so $\hat{\sigma}(\mu,\nu) = \sigma(\mu,\nu)$ implies that $\hat{\sigma}(\ell,\tau) = \sigma(\ell,\tau)$ for every $(\ell,\tau) \in \mathcal{C}$. This completes Direction 1.

***Direction 2**: if $\hat{\sigma} = \sigma$ on $\mathcal{C}$, then $\hat{G} = a\,G + b$ for some $a > 0$ and b.* Suppose $\hat{\sigma}(\ell,\tau) = \sigma(\ell,\tau)$ for every $(\ell,\tau) \in \mathcal{C}$, satisfying the antecedent in Direction 2. We refer to this supposition as agreement on $\mathcal{C}$. Throughout this direction, x, y, and z denote trajectories in $\mathcal{T}$.

We first derive a formula for the preference between a lottery and a trajectory, which Steps (i) and (ii) below both use. Let $x, y, z \in \mathcal{T}$ and $p \in (0,1)$ be arbitrary, and let $\ell \triangleq \langle p, y, x\rangle$.

$$\sigma(\ell,z) \triangleq \operatorname{sign}\big(\mathbb{E}_\ell[G] - \mathbb{E}_z[G]\big) \qquad \text{(definition of } \sigma)$$
$$\sigma(\ell,z) = \operatorname{sign}\big(\mathbb{E}_\ell[G] - G(z)\big) \qquad \text{(substitute } \mathbb{E}_z[G] = G(z))$$
$$\sigma(\ell,z) = \operatorname{sign}\big(p\,G(y) + (1-p)\,G(x) - G(z)\big) \qquad \text{(substitute the expected return of } \ell)$$
$$\sigma(\ell,z) = \operatorname{sign}\big(p\,G(y) + G(x) - p\,G(x) - G(z)\big) \qquad \text{(distribute } G(x) \text{ over } 1-p)$$
$$\sigma(\ell,z) = \operatorname{sign}\big(p\,G(y) - p\,G(x) + G(x) - G(z)\big) \qquad \text{(swap the second and third terms)}$$
$$\sigma(\ell,z) = \operatorname{sign}\big(p\,(G(y) - G(x)) + G(x) - G(z)\big) \qquad \text{(factor } p \text{ out of the first two terms)}$$
$$\sigma(\ell,z) = \operatorname{sign}\big(p\,(G(y) - G(x)) - (G(z) - G(x))\big) \qquad \text{(factor } -1 \text{ out of the last two terms)}$$

We refer to the last line as **the lottery formula**. The seven lines above use only the definitions of σ and of expected return, which are the same for $\hat{\sigma}$ and $\hat{G}$. The last line therefore also holds with $\hat{\sigma}$ and $\hat{G}$ in place of σ and G, and we refer to that version as the lottery formula for $\hat{G}$.

Step (i). We derive two equalities, both used by Step (iii). First, $\hat{G}$ and G rank every trajectory pair identically. Second, in the special case of a tie, two trajectories with equal ground-truth returns have equal candidate returns. Let $x \neq y$ and $p \in (0,1)$ be arbitrary, and let $\ell \triangleq \langle p, y, x\rangle$, so that $(\ell, x) \in \mathcal{C}$.

$$\sigma(\ell,x) = \operatorname{sign}\big(p\,(G(y) - G(x)) - (G(x) - G(x))\big) \qquad \text{(lottery formula with } z = x)$$
$$\sigma(\ell,x) = \operatorname{sign}\big(p\,(G(y) - G(x))\big) \qquad \text{(evaluate } c - c = 0)$$
$$\sigma(\ell,x) = \operatorname{sign}\big(G(y) - G(x)\big) \qquad (\operatorname{sign}(pc) = \operatorname{sign}(c) \text{ for } p > 0)$$
$$\hat{\sigma}(\ell,x) = \operatorname{sign}\big(\hat{G}(y) - \hat{G}(x)\big) \qquad \text{(repeat last 3 lines with } \hat{G} \text{ and } \hat{\sigma})$$
$$\sigma(\ell,x) = \operatorname{sign}\big(\hat{G}(y) - \hat{G}(x)\big) \qquad \text{(substitute by agreement on } \mathcal{C})$$
$$\operatorname{sign}\big(G(y) - G(x)\big) = \operatorname{sign}\big(\hat{G}(y) - \hat{G}(x)\big) \qquad \text{(substitute from } \sigma(\ell,x) = \operatorname{sign}\big(G(y) - G(x)\big))$$

The last line also holds when $x = y$, since both sides equal $\mathrm{sign}(0)$. Therefore it holds for all $x, y \in \mathcal{T}$. It states that $\hat{G}$ and G rank every trajectory pair identically, which is the first claim of this step. For the second claim, suppose $G(x) = G(y)$.

$$\begin{aligned}
\mathrm{sign}\big(G(y) - G(x)\big) &= \mathrm{sign}\big(\hat{G}(y) - \hat{G}(x)\big) && \text{(last line of the previous derivation)}\\
\mathrm{sign}\big(G(y) - G(y)\big) &= \mathrm{sign}\big(\hat{G}(y) - \hat{G}(x)\big) && \text{(substitute } G(x) = G(y)\text{)}\\
0 &= \mathrm{sign}\big(\hat{G}(y) - \hat{G}(x)\big) && \text{(evaluate } c - c = 0 \text{) and } \mathrm{sign}(0) = 0\text{)}\\
0 &= \hat{G}(y) - \hat{G}(x) && \text{(}\mathrm{sign}(c) = 0 \text{ only if } c = 0\text{)}\\
\hat{G}(x) &= \hat{G}(y) && \text{(add } \hat{G}(x) \text{ to both sides)}
\end{aligned}$$

Hence $G(x) = G(y)$ implies $\hat{G}(x) = \hat{G}(y)$, which is the second claim of this step.

Step (ii). We show that for three trajectories with distinct ground-truth returns, the points $\big(G(\cdot), \hat{G}(\cdot)\big)$ are collinear. For arbitrary trajectories $x, y \in \mathcal{T}$ with $G(x) \neq G(y)$, let $\mathrm{slope}(x, y) \triangleq \frac{\hat{G}(x) - \hat{G}(y)}{G(x) - G(y)}$ denote the slope of the line through the points $\big(G(x), \hat{G}(x)\big)$ and $\big(G(y), \hat{G}(y)\big)$.

$$\begin{aligned}
\mathrm{slope}(y, x) &\triangleq \frac{\hat{G}(y) - \hat{G}(x)}{G(y) - G(x)} && \text{(definition of slope)}\\
\mathrm{slope}(y, x) &= \frac{\hat{G}(x) - \hat{G}(y)}{G(x) - G(y)} && \text{(multiply numerator and denominator by } -1\text{)}\\
\mathrm{slope}(y, x) &= \mathrm{slope}(x, y) && \text{(definition of slope)}
\end{aligned}$$

The last line says that slope does not depend on the order of its two arguments. Now let $x, y, z \in \mathcal{T}$ be such that $G(z)$ lies strictly between $G(x)$ and $G(y)$, i.e., either $G(x) < G(z) < G(y)$ or $G(y) < G(z) < G(x)$. Let $p^\star \triangleq \frac{G(z) - G(x)}{G(y) - G(x)}$. The numerator and the denominator of $p^\star$ are nonzero and have the same sign, and the numerator is smaller in absolute value, so $p^\star \in (0, 1)$. Let $\ell^\star \triangleq \langle p^\star, y, x \rangle$. Since $G(x) \neq G(y)$, the trajectories x and y are distinct, so $(\ell^\star, z) \in \mathcal{C}$. We first evaluate $\sigma(\ell^\star, z)$ with the lottery formula, which states that $\sigma(\ell, z) = \mathrm{sign}\big(p\,(G(y) - G(x)) - (G(z) - G(x))\big)$ for $\ell = \langle p, y, x \rangle$.

$$\begin{aligned}
\sigma(\ell^\star, z) &= \mathrm{sign}\big(p^\star(G(y) - G(x)) - (G(z) - G(x))\big) && \text{(lottery formula with } p = p^\star\text{)}\\
\sigma(\ell^\star, z) &= \mathrm{sign}\Big(\tfrac{G(z) - G(x)}{G(y) - G(x)}\,(G(y) - G(x)) - (G(z) - G(x))\Big) && \text{(substitute the definition of } p^\star\text{)}\\
\sigma(\ell^\star, z) &= \mathrm{sign}\big((G(z) - G(x)) - (G(z) - G(x))\big) && \text{(cancel the factor } G(y) - G(x)\text{)}\\
\sigma(\ell^\star, z) &= \mathrm{sign}(0) && \text{(evaluate } c - c = 0\text{)}\\
\sigma(\ell^\star, z) &= 0 && \text{(evaluate } \mathrm{sign}(0) = 0\text{)}
\end{aligned}$$

We then use agreement on $\mathcal{C}$ to obtain the corresponding equation for $\hat{G}$.

$$\begin{aligned}
\hat{\sigma}(\ell^\star, z) &= \mathrm{sign}\big(p^\star(\hat{G}(y) - \hat{G}(x)) - (\hat{G}(z) - \hat{G}(x))\big) && \text{(lottery formula for } \hat{G}\text{, with } p = p^\star\text{)}\\
\sigma(\ell^\star, z) &= \mathrm{sign}\big(p^\star(\hat{G}(y) - \hat{G}(x)) - (\hat{G}(z) - \hat{G}(x))\big) && \text{(substitute by agreement on } \mathcal{C}\text{)}\\
0 &= \mathrm{sign}\big(p^\star(\hat{G}(y) - \hat{G}(x)) - (\hat{G}(z) - \hat{G}(x))\big) && \text{(substitute } \sigma(\ell^\star, z) = 0\text{)}\\
0 &= p^\star(\hat{G}(y) - \hat{G}(x)) - (\hat{G}(z) - \hat{G}(x)) && \text{(}\mathrm{sign}(c) = 0 \text{ only if } c = 0\text{)}\\
\hat{G}(z) - \hat{G}(x) &= p^\star(\hat{G}(y) - \hat{G}(x)) && \text{(add } \hat{G}(z) - \hat{G}(x) \text{ to both sides)}\\
\hat{G}(z) - \hat{G}(x) &= \frac{G(z) - G(x)}{G(y) - G(x)}\,(\hat{G}(y) - \hat{G}(x)) && \text{(substitute the definition of } p^\star\text{)}\\
\frac{\hat{G}(z) - \hat{G}(x)}{G(z) - G(x)} &= \frac{G(z) - G(x)}{G(y) - G(x)} \cdot \frac{\hat{G}(y) - \hat{G}(x)}{G(z) - G(x)} && \text{(divide both sides by } G(z) - G(x)\text{)}\\
\frac{\hat{G}(z) - \hat{G}(x)}{G(z) - G(x)} &= \frac{\hat{G}(y) - \hat{G}(x)}{G(y) - G(x)} && \text{(cancel the factor } G(z) - G(x)\text{)}\\
\mathrm{slope}(z, x) &= \mathrm{slope}(y, x) && \text{(definition of slope, on both sides)}
\end{aligned}$$

The last line says that the line through the points for x and z and the line through the points for x and y have the same slope, so the points for x, y, and z are collinear.

Step (iii). We now complete the reverse implication by showing that $\hat{G}$ is a positive affine transformation of G, meaning that for every $\tau \in \mathcal{T}$ there exist constants $a > 0$ and b such that $\hat{G}(\tau) = a\,G(\tau) + b$. There are two cases.

Case A: G is constant on $\mathcal{T}$. Choose any $\tau_0 \in \mathcal{T}$, and let $a \triangleq 1$ and $b \triangleq \hat{G}(\tau_0) - G(\tau_0)$. Let $\tau \in \mathcal{T}$ be arbitrary. Since G is constant, $G(\tau) = G(\tau_0)$, so Step (i) gives $\hat{G}(\tau) = \hat{G}(\tau_0)$.

$a\,G(\tau) + b = a \cdot G(\tau) + b$	()
$a\,G(\tau) + b = G(\tau) + b$	(substitute $a = 1$ and simplify)
$a\,G(\tau) + b = G(\tau) + \hat{G}(\tau_0) - G(\tau_0)$	(definition of b)
$a\,G(\tau) + b = G(\tau_0) + \hat{G}(\tau_0) - G(\tau_0)$	(substitute $G(\tau) = G(\tau_0)$)
$a\,G(\tau) + b = \hat{G}(\tau_0)$	(cancel $G(\tau_0)$ with $-G(\tau_0)$)
$a\,G(\tau) + b = \hat{G}(\tau)$	(substitute $\hat{G}(\tau) = \hat{G}(\tau_0)$)

Case B: G is not constant on $\mathcal{T}$. Step (iii) seeks constants $a > 0$ and b for which $\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$, i.e., for which the points $\big(G(\tau), \hat{G}(\tau)\big)$ for all $\tau \in \mathcal{T}$ lie on the line with slope a through $(0, b)$.

Choose $\tau_{\text{lo}}, \tau_{\text{hi}} \in \mathcal{T}$ with $G(\tau_{\text{lo}}) < G(\tau_{\text{hi}})$. Because $G(\tau_{\text{lo}}) < G(\tau_{\text{hi}})$, the point $\big(G(\tau_{\text{lo}}), \hat{G}(\tau_{\text{lo}})\big)$ and the point $\big(G(\tau_{\text{hi}}), \hat{G}(\tau_{\text{hi}})\big)$ are distinct, so exactly one line passes through both. To characterize that line, we let $a \triangleq \text{slope}(\tau_{\text{hi}}, \tau_{\text{lo}})$, which is the slope of that line, and $b \triangleq \hat{G}(\tau_{\text{lo}}) - a\,G(\tau_{\text{lo}})$, so that the line with slope a through $(0, b)$ is that same line. To complete Case B, it remains to show that $a > 0$ and that, for every $\tau \in \mathcal{T}$, the point $\big(G(\tau), \hat{G}(\tau)\big)$ is on this same line, i.e., $a\,G(\tau) + b = \hat{G}(\tau)$.

We first show that $a > 0$, starting from the result of Step (i) that $\hat{G}$ and G rank every trajectory pair identically.

$\text{sign}\big(G(\tau_{\text{hi}}) - G(\tau_{\text{lo}})\big) = \text{sign}\big(\hat{G}(\tau_{\text{hi}}) - \hat{G}(\tau_{\text{lo}})\big)$	(first claim of Step (i) with $x = \tau_{\text{lo}}$, $y = \tau_{\text{hi}}$)
$1 = \text{sign}\big(\hat{G}(\tau_{\text{hi}}) - \hat{G}(\tau_{\text{lo}})\big)$	($\text{sign}(c) = 1$ for $c = G(\tau_{\text{hi}}) - G(\tau_{\text{lo}}) > 0$)
$\hat{G}(\tau_{\text{hi}}) - \hat{G}(\tau_{\text{lo}}) > 0$	($\text{sign}(c) = 1$ only if $c > 0$)
$\dfrac{\hat{G}(\tau_{\text{hi}}) - \hat{G}(\tau_{\text{lo}})}{G(\tau_{\text{hi}}) - G(\tau_{\text{lo}})} > 0$	(divide both sides by $G(\tau_{\text{hi}}) - G(\tau_{\text{lo}})$, which is positive)
$\text{slope}(\tau_{\text{hi}}, \tau_{\text{lo}}) > 0$	(definition of slope)
$a > 0$	(definition of a)

Now let $\tau \in \mathcal{T}$ be arbitrary. We show $a\,G(\tau) + b = \hat{G}(\tau)$ in five sub-cases: $G(\tau) < G(\tau_{\text{lo}}) < G(\tau_{\text{hi}})$, $G(\tau) = G(\tau_{\text{lo}})$, $G(\tau_{\text{lo}}) < G(\tau) < G(\tau_{\text{hi}})$, $G(\tau) = G(\tau_{\text{hi}})$, and $G(\tau_{\text{lo}}) < G(\tau_{\text{hi}}) < G(\tau)$. The sub-cases are however presented in an order that groups them by proof technique.

Sub-case B1: $G(\tau) = G(\tau_{\text{lo}})$**.** In this sub-case, the second claim of Step (i) gives $\hat{G}(\tau) = \hat{G}(\tau_{\text{lo}})$.

$b \triangleq \hat{G}(\tau_{\text{lo}}) - a\,G(\tau_{\text{lo}})$	(definition of b)
$a\,G(\tau_{\text{lo}}) + b = \hat{G}(\tau_{\text{lo}})$	(add $a\,G(\tau_{\text{lo}})$ to both sides)
$a\,G(\tau) + b = \hat{G}(\tau_{\text{lo}})$	(substitute $G(\tau) = G(\tau_{\text{lo}})$)
$a\,G(\tau) + b = \hat{G}(\tau)$	(substitute $\hat{G}(\tau) = \hat{G}(\tau_{\text{lo}})$)

In each of Sub-cases B2 to B5, $G(\tau) \neq G(\tau_{\text{lo}})$, so $\text{slope}(\tau, \tau_{\text{lo}})$ is defined, and each of these sub-cases shows that $a = \text{slope}(\tau, \tau_{\text{lo}})$; the derivation immediately after Sub-case B5 then shows that $a = \text{slope}(\tau, \tau_{\text{lo}})$ implies $a\,G(\tau) + b = \hat{G}(\tau)$.

Sub-case B2: $G(\tau) = G(\tau_{\mathrm{hi}})$**.** Step (i) established that $G(x) = G(y)$ implies $\hat{G}(x) = \hat{G}(y)$, so $\hat{G}(\tau) = \hat{G}(\tau_{\mathrm{hi}})$, which the following derivation uses.

$$
\begin{aligned}
a &\triangleq \mathrm{slope}(\tau_{\mathrm{hi}}, \tau_{\mathrm{lo}}) && \text{(definition of } a\text{)} \\
a &= \frac{\hat{G}(\tau_{\mathrm{hi}}) - \hat{G}(\tau_{\mathrm{lo}})}{G(\tau_{\mathrm{hi}}) - G(\tau_{\mathrm{lo}})} && \text{(definition of slope)} \\
a &= \frac{\hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}})}{G(\tau_{\mathrm{hi}}) - G(\tau_{\mathrm{lo}})} && \text{(substitute } \hat{G}(\tau) = \hat{G}(\tau_{\mathrm{hi}})\text{)} \\
a &= \frac{\hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}})}{G(\tau) - G(\tau_{\mathrm{lo}})} && \text{(substitute } G(\tau) = G(\tau_{\mathrm{hi}})\text{)} \\
a &= \mathrm{slope}(\tau, \tau_{\mathrm{lo}}) && \text{(definition of slope)}
\end{aligned}
$$

Sub-cases B3 to B5 use the result of Step (ii) that $\mathrm{slope}(z, x) = \mathrm{slope}(y, x)$ whenever $G(z)$ lies strictly between $G(x)$ and $G(y)$, and Sub-case B5 also uses the result of Step (ii) that $\mathrm{slope}(y, x) = \mathrm{slope}(x, y)$.

Sub-case B3: $G(\tau_{\mathrm{lo}}) < G(\tau) < G(\tau_{\mathrm{hi}})$**.**

$$
\begin{aligned}
a &\triangleq \mathrm{slope}(\tau_{\mathrm{hi}}, \tau_{\mathrm{lo}}) && \text{(definition of } a\text{)} \\
a &= \mathrm{slope}(\tau, \tau_{\mathrm{lo}}) && (\mathrm{slope}(z, x) = \mathrm{slope}(y, x) \text{ from Step (ii)})
\end{aligned}
$$

Sub-case B4: $G(\tau_{\mathrm{lo}}) < G(\tau_{\mathrm{hi}}) < G(\tau)$**.**

$$
\begin{aligned}
a &\triangleq \mathrm{slope}(\tau_{\mathrm{hi}}, \tau_{\mathrm{lo}}) && \text{(definition of } a\text{)} \\
a &= \mathrm{slope}(\tau, \tau_{\mathrm{lo}}) && (\mathrm{slope}(z, x) = \mathrm{slope}(y, x) \text{ from Step (ii)})
\end{aligned}
$$

Sub-case B5: $G(\tau) < G(\tau_{\mathrm{lo}}) < G(\tau_{\mathrm{hi}})$**.**

$$
\begin{aligned}
a &\triangleq \mathrm{slope}(\tau_{\mathrm{hi}}, \tau_{\mathrm{lo}}) && \text{(definition of } a\text{)} \\
a &= \mathrm{slope}(\tau_{\mathrm{lo}}, \tau_{\mathrm{hi}}) && (\mathrm{slope}(y, x) = \mathrm{slope}(x, y) \text{ from Step (ii)}) \\
a &= \mathrm{slope}(\tau, \tau_{\mathrm{hi}}) && (\mathrm{slope}(z, x) = \mathrm{slope}(y, x) \text{ from Step (ii)}) \\
a &= \mathrm{slope}(\tau_{\mathrm{hi}}, \tau) && (\mathrm{slope}(y, x) = \mathrm{slope}(x, y) \text{ from Step (ii)}) \\
a &= \mathrm{slope}(\tau_{\mathrm{lo}}, \tau) && (\mathrm{slope}(z, x) = \mathrm{slope}(y, x) \text{ from Step (ii)}) \\
a &= \mathrm{slope}(\tau, \tau_{\mathrm{lo}}) && (\mathrm{slope}(y, x) = \mathrm{slope}(x, y) \text{ from Step (ii)})
\end{aligned}
$$

In Sub-cases B2 to B5, $a = \mathrm{slope}(\tau, \tau_{\mathrm{lo}})$.

$$
\begin{aligned}
\mathrm{slope}(\tau, \tau_{\mathrm{lo}}) &\triangleq \frac{\hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}})}{G(\tau) - G(\tau_{\mathrm{lo}})} && \text{(definition of slope)} \\
a &= \frac{\hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}})}{G(\tau) - G(\tau_{\mathrm{lo}})} && \text{(substitute } a = \mathrm{slope}(\tau, \tau_{\mathrm{lo}})\text{)} \\
a\,(G(\tau) - G(\tau_{\mathrm{lo}})) &= \hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}}) && \text{(multiply both sides by } G(\tau) - G(\tau_{\mathrm{lo}})\text{)} \\
a\,G(\tau) - a\,G(\tau_{\mathrm{lo}}) &= \hat{G}(\tau) - \hat{G}(\tau_{\mathrm{lo}}) && \text{(distribute } a \text{ over } G(\tau) - G(\tau_{\mathrm{lo}})\text{)} \\
a\,G(\tau) - a\,G(\tau_{\mathrm{lo}}) + \hat{G}(\tau_{\mathrm{lo}}) &= \hat{G}(\tau) && \text{(add } \hat{G}(\tau_{\mathrm{lo}}) \text{ to both sides)} \\
a\,G(\tau) + \hat{G}(\tau_{\mathrm{lo}}) - a\,G(\tau_{\mathrm{lo}}) &= \hat{G}(\tau) && \text{(swap the second and third terms)} \\
a\,G(\tau) + b &= \hat{G}(\tau) && \text{(definition of } b\text{)}
\end{aligned}
$$

In Case A, and in Case B across its five sub-cases, the a and b defined in that case satisfy $a > 0$ and $a\,G(\tau) + b = \hat{G}(\tau)$ for every $\tau \in \mathcal{T}$. The lemma states this equation with $\hat{G}(\tau)$ on the left, so we exchange its two sides: $\hat{G}(\tau) = a\,G(\tau) + b$.

This completes Direction 2. □

Proposition 1 (Lottery-trajectory pairs suffice to detect any misalignment)**.** *If no lottery-trajectory pair is discordant, then no pair $(\mu, \nu) \in \Delta(\mathcal{T}) \times \Delta(\mathcal{T})$ is discordant, i.e., $\hat{G}$ is not misaligned with G. Hence $\hat{G}$ is misaligned with G if and only if some lottery-trajectory pair is discordant.*

Proof. The proof has two parts. Part 1 shows that if no lottery-trajectory pair is discordant, then $\hat{G}$ is not misaligned with G. Part 2 shows the converse, that if some lottery-trajectory pair is discordant, then $\hat{G}$ is misaligned with G. The two parts together give the final sentence of the proposition. In each derivation, every line follows from the line before it by the reason named at the right.

Part 1.

No lottery-trajectory pair is discordant.	(supposition)
$\hat{\sigma}(\ell, \tau) = \sigma(\ell, \tau)$ for every $(\ell, \tau) \in \mathcal{C}$.	(definition of discordant)
$\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$, for some $a > 0$ and $b \in \mathbb{R}$.	(Lemma 1)
$\hat{\sigma}(\mu, \nu) = \sigma(\mu, \nu)$ for every $\mu, \nu \in \Delta(\mathcal{T})$.	(Direction 1, proof of Lemma 1)
No pair $(\mu, \nu) \in \Delta(\mathcal{T}) \times \Delta(\mathcal{T})$ is discordant.	(definition of discordant)
$\hat{G}$ is not misaligned with G.	(Definition 1)

Part 2.

Some lottery-trajectory pair $(\ell, \tau) \in \mathcal{C}$ is discordant.	(supposition)
Some pair in $\Delta(\mathcal{T}) \times \Delta(\mathcal{T})$ is discordant.	(ℓ and τ are distributions in $\Delta(\mathcal{T})$)
$\hat{G}$ is misaligned with G.	(Definition 1)

□

For an initial-state distribution D_0', a transition function T', and a policy π that together generate only trajectories in $\mathcal{T}$, let $\mu_\pi^{D_0', T'} \in \Delta(\mathcal{T})$ denote the distribution over the trajectories they generate. Writing $\mu \triangleq \mu_\pi^{D_0', T'}$, let $J^{D_0', T'}(\pi) \triangleq \mathbb{E}_\mu[G]$ denote the ground-truth expected return of π and $\hat{J}^{D_0', T'}(\pi) \triangleq \mathbb{E}_\mu[\hat{G}]$ the candidate expected return of π, and let $\Pi^\star(D_0', T')$ and $\hat{\Pi}^\star(D_0', T')$ denote the sets of policies that maximize $J^{D_0', T'}$ and $\hat{J}^{D_0', T'}$, respectively.

Corollary 1 (Identical policy rankings under any transition function)**.** *If no lottery-trajectory pair is discordant, then there exist $a > 0$ and $b \in \mathbb{R}$ such that $\hat{J}^{D_0', T'}(\pi) = a\,J^{D_0', T'}(\pi) + b$ for every initial-state distribution D_0', transition function T', and policy π that together generate only trajectories in $\mathcal{T}$. Hence, under each such D_0' and T', the candidate expected return and the ground-truth expected return rank all such policies identically, and in particular $\hat{\Pi}^\star(D_0', T') = \Pi^\star(D_0', T')$. When $\mathcal{T}$ contains all full trajectories, these conclusions hold under every transition function, including ones other than the T of $\mathcal{M}$.*

Proof. Suppose no lottery-trajectory pair is discordant. By the first three lines of Part 1 of the proof of Proposition 1, there exist $a > 0$ and $b \in \mathbb{R}$ such that $\hat{G}(\tau) = a\,G(\tau) + b$ for every $\tau \in \mathcal{T}$. Direction 1 of the proof of Lemma 1 therefore gives $\mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_\mu[G] + b$ for every $\mu \in \Delta(\mathcal{T})$ and $\hat{\sigma}(\mu, \nu) = \sigma(\mu, \nu)$ for every $\mu, \nu \in \Delta(\mathcal{T})$. These constants come from Lemma 1, which involves only $\mathcal{T}$, G, and $\hat{G}$, so they do not depend on D_0', T', or π. Let D_0' and T' be arbitrary. To shorten the derivations, in this proof we write J, $\hat{J}$, $\Pi^\star$, and $\hat{\Pi}^\star$ without the superscript or argument D_0', T'. The proof has three parts, one for each conclusion of the corollary.

Part 1: $\hat{J}(\pi) = a\,J(\pi) + b$. Let π be an arbitrary policy that, with D_0' and T', generates only trajectories in $\mathcal{T}$, and write $\mu \triangleq \mu_\pi^{D_0', T'}$.

$\hat{J}(\pi) \triangleq \mathbb{E}_\mu[\hat{G}]$	(definition of $\hat{J}$)
$\hat{J}(\pi) = a\,\mathbb{E}_\mu[G] + b$	(substitute $\mathbb{E}_\mu[\hat{G}] = a\,\mathbb{E}_\mu[G] + b$)
$\hat{J}(\pi) = a\,J(\pi) + b$	(definition of J)

Part 2: $\hat{J}$ and J rank policies identically. Let π and π' be two such policies, and write $\mu \triangleq \mu_\pi^{D_0', T'}$ and $\mu' \triangleq \mu_{\pi'}^{D_0', T'}$.

$\operatorname{sign}\big(\hat{J}(\pi) - \hat{J}(\pi')\big) = \operatorname{sign}\big(\mathbb{E}_\mu[\hat{G}] - \mathbb{E}_{\mu'}[\hat{G}]\big)$	(definition of $\hat{J}$, for π and π')
$\operatorname{sign}\big(\hat{J}(\pi) - \hat{J}(\pi')\big) = \hat{\sigma}(\mu, \mu')$	(definition of $\hat{\sigma}$)
$\operatorname{sign}\big(\hat{J}(\pi) - \hat{J}(\pi')\big) = \sigma(\mu, \mu')$	(substitute $\hat{\sigma}(\mu, \mu') = \sigma(\mu, \mu')$)
$\operatorname{sign}\big(\hat{J}(\pi) - \hat{J}(\pi')\big) = \operatorname{sign}\big(\mathbb{E}_\mu[G] - \mathbb{E}_{\mu'}[G]\big)$	(definition of σ)
$\operatorname{sign}\big(\hat{J}(\pi) - \hat{J}(\pi')\big) = \operatorname{sign}\big(J(\pi) - J(\pi')\big)$	(definition of J, for π and π')

Part 3: $\hat{\Pi}^\star = \Pi^\star$. Let π be such a policy. Below, "for every π'" ranges over all such policies.

$\pi \in \hat{\Pi}^\star \iff \hat{J}(\pi) \geq \hat{J}(\pi')$ for every π' (definition of $\hat{\Pi}^\star$)

$\pi \in \hat{\Pi}^\star \iff \hat{J}(\pi) - \hat{J}(\pi') \geq 0$ for every π' (subtract $\hat{J}(\pi')$ from both sides)

$\pi \in \hat{\Pi}^\star \iff \operatorname{sign}\left(\hat{J}(\pi) - \hat{J}(\pi')\right) \geq 0$ for every π' ($c \geq 0 \iff \operatorname{sign}(c) \geq 0$)

$\pi \in \hat{\Pi}^\star \iff \operatorname{sign}\left(J(\pi) - J(\pi')\right) \geq 0$ for every π' (substitute the last line of Part 2)

$\pi \in \hat{\Pi}^\star \iff J(\pi) - J(\pi') \geq 0$ for every π' ($c \geq 0 \iff \operatorname{sign}(c) \geq 0$)

$\pi \in \hat{\Pi}^\star \iff J(\pi) \geq J(\pi')$ for every π' (add $J(\pi')$ to both sides)

$\pi \in \hat{\Pi}^\star \iff \pi \in \Pi^\star$ (definition of $\Pi^\star$)

Finally, when $\mathcal{T}$ contains all full trajectories, every D_0', T', and π generate only trajectories in $\mathcal{T}$, so the three conclusions hold under every transition function T'. □

As noted in Section 4.1, two clarifications apply to Corollary 1. First, the initial-state distribution matters for the ranking of policies by expected return, but it matters little for the set of optimal policies. In an MDP, a policy that is optimal from every state maximizes expected return under every initial-state distribution [Sutton and Barto, 2018], so the set of policies that maximize expected return depends on the initial-state distribution only through which states can be reached from it. Second, $\hat{J}$ is computed with the candidate discount factor $\hat{\gamma}$, so Corollary 1 does not cover a policy that is trained with $\hat{r}$ under a discount factor other than $\hat{\gamma}$.

Remarks. The following remarks support three statements made in the main text.

Trajectory pairs alone do not suffice (Section 4.1). Step (i) in the proof of Lemma 1 concludes that $\hat{G}$ and G rank every trajectory pair identically. That property alone does not imply that $\hat{G}$ is a positive affine transformation of G. For example, suppose three trajectories τ_1, τ_2, τ_3 have ground-truth returns $0, 1, 10$ and candidate returns $0, 1, 2$, so that $\hat{G}$ and G rank the three trajectories identically. Let $\ell \triangleq \langle 1/2, \tau_3, \tau_1 \rangle$.

$\mathbb{E}_\ell[G] = \frac{1}{2}\, G(\tau_3) + \frac{1}{2}\, G(\tau_1)$ (expected return of the lottery ℓ)

$\mathbb{E}_\ell[G] = \frac{1}{2} \cdot 10 + \frac{1}{2}\, G(\tau_1)$ (substitute $G(\tau_3) = 10$)

$\mathbb{E}_\ell[G] = \frac{1}{2} \cdot 10 + \frac{1}{2} \cdot 0$ (substitute $G(\tau_1) = 0$)

$\mathbb{E}_\ell[G] = 5$ (evaluate $\frac{1}{2} \cdot 10 + \frac{1}{2} \cdot 0$)

The same four lines with $\hat{G}(\tau_3) = 2$ and $\hat{G}(\tau_1) = 0$ give $\mathbb{E}_\ell[\hat{G}] = 1$. Hence G ranks ℓ above τ_2, because $5 > 1 = G(\tau_2)$, whereas $\hat{G}$ ranks ℓ and τ_2 equally, because $1 = \hat{G}(\tau_2)$. The pair (ℓ, τ_2) is therefore discordant. Whether trajectory pairs alone determine G up to a positive affine transformation differs across MDPs [Skalse et al., 2023, Theorem 3.10].

The choice of lottery probability (Section 4.2). The lottery formula from the proof of Lemma 1 states that $\sigma(\ell, z) = \operatorname{sign}\left(p\,(G(y) - G(x)) - (G(z) - G(x))\right)$ for $\ell = \langle p, y, x \rangle$. It explains why the scaffolded reward debugger sets p near the value at which $\hat{G}$ ranks the lottery and the trajectory equally. Let τ_1, τ_2, τ_3 satisfy $G(\tau_1) < G(\tau_3)$ and $\hat{G}(\tau_1) < \hat{G}(\tau_3)$, let $p^\star \triangleq \frac{G(\tau_2) - G(\tau_1)}{G(\tau_3) - G(\tau_1)}$ and $\hat{p}^\star \triangleq \frac{\hat{G}(\tau_2) - \hat{G}(\tau_1)}{\hat{G}(\tau_3) - \hat{G}(\tau_1)}$, and let $\ell \triangleq \langle p, \tau_3, \tau_1 \rangle$ for an arbitrary $p \in (0, 1)$.

$p^\star \triangleq \dfrac{G(\tau_2) - G(\tau_1)}{G(\tau_3) - G(\tau_1)}$ (definition of $p^\star$)

$p^\star\,(G(\tau_3) - G(\tau_1)) = G(\tau_2) - G(\tau_1)$ (multiply both sides by $G(\tau_3) - G(\tau_1)$)

$\sigma(\ell, \tau_2) = \operatorname{sign}\left(p\,(G(\tau_3) - G(\tau_1)) - (G(\tau_2) - G(\tau_1))\right)$ (lottery formula with $x = \tau_1$, $y = \tau_3$, $z = \tau_2$)

$\sigma(\ell, \tau_2) = \operatorname{sign}\left(p\,(G(\tau_3) - G(\tau_1)) - p^\star(G(\tau_3) - G(\tau_1))\right)$ (substitute the last line of the previous display)

$\sigma(\ell, \tau_2) = \operatorname{sign}\left((p - p^\star)(G(\tau_3) - G(\tau_1))\right)$ (factor out $G(\tau_3) - G(\tau_1)$)

$\sigma(\ell, \tau_2) = \operatorname{sign}(p - p^\star)$ ($\operatorname{sign}(cd) = \operatorname{sign}(c)$ for $d > 0$)

The same six lines, with $\hat{G}$, $\hat{\sigma}$, and $\hat{p}^\star$ in place of G, σ, and $p^\star$, give $\hat{\sigma}(\ell, \tau_2) = \operatorname{sign}(p - \hat{p}^\star)$. The pair (ℓ, τ_2) is discordant exactly when $\operatorname{sign}(p - p^\star) \neq \operatorname{sign}(p - \hat{p}^\star)$, which holds if and only if $p^\star \neq \hat{p}^\star$ and p lies in the closed interval between $p^\star$ and $\hat{p}^\star$. Setting $p = \hat{p}^\star$ therefore yields a discordant pair whenever $p^\star \neq \hat{p}^\star$, whereas any other p does so only if $p^\star$ lies on the same side of $\hat{p}^\star$ as p and at least as far from $\hat{p}^\star$ as p is.

Potential-based shaping (Section 4.1). Let $F(s, a, s') \triangleq \gamma\Phi(s') - \Phi(s)$ for a potential function Φ [Ng et al., 1999], and take the candidate reward function to be $r + F$ and the candidate discount factor to be γ. For infinite-horizon trajectories, the candidate return is then $\hat{G}(\tau) = G(\tau) - \Phi(s_0^\tau)$, where s_0^τ is the start state of τ [Skalse et al., 2023, Lemma B.1]. Suppose this $\hat{G}$ is a positive affine transformation of G, i.e., there exist $c > 0$ and $d \in \mathbb{R}$ with $\hat{G}(\tau) = c\,G(\tau) + d$ for every $\tau \in \mathcal{T}$. Let $\tau \in \mathcal{T}$ be arbitrary.

$$
\begin{aligned}
G(\tau) - \Phi(s_0^\tau) &= c\,G(\tau) + d && \text{(equate the two expressions for } \hat{G}(\tau)) \\
G(\tau) - c\,G(\tau) - \Phi(s_0^\tau) &= d && \text{(subtract } c\,G(\tau) \text{ from both sides)} \\
G(\tau) - c\,G(\tau) &= d + \Phi(s_0^\tau) && \text{(add } \Phi(s_0^\tau) \text{ to both sides)} \\
(1 - c)\,G(\tau) &= d + \Phi(s_0^\tau) && \text{(factor out } G(\tau))
\end{aligned}
$$

If $c = 1$:

$$
\begin{aligned}
(1 - 1)\,G(\tau) &= d + \Phi(s_0^\tau) && \text{(substitute } c = 1) \\
0 &= d + \Phi(s_0^\tau) && \text{(evaluate } (1 - 1)\,G(\tau)) \\
-d &= \Phi(s_0^\tau) && \text{(subtract } d \text{ from both sides)}
\end{aligned}
$$

so Φ takes the same value, $-d$, at the start state of every trajectory in $\mathcal{T}$. If $c \neq 1$:

$$
G(\tau) = \frac{d + \Phi(s_0^\tau)}{1 - c} \qquad \text{(divide both sides by } 1 - c)
$$

so $G(\tau)$ depends on τ only through its start state, i.e., G is constant on the trajectories from each start state. Hence $\hat{G}$ is a positive affine transformation of G only if Φ takes the same value at the start state of every trajectory in $\mathcal{T}$ or G is constant on the trajectories from each start state. Otherwise, by Lemma 1, some lottery-trajectory pair is discordant, so $(r + F, \gamma)$ is misaligned with (r, γ) under Definition 1, even though $r + F$ and r have the same optimal policies in every MDP [Ng et al., 1999]. The converse of Corollary 1 therefore does not hold, i.e., a candidate reward function can have the same optimal policies as the ground-truth reward function under every transition function while the candidate return function is misaligned with G. TAC [Muslimani et al., 2025] is invariant to potential-based shaping because it compares only trajectories that share a start state, whereas Definition 1 imposes no such restriction.