# Reward Design

# The Perils of Trial-and-Error Reward Design: Misdesign through Overfitting and Invalid Task Specifications

**Serena Booth**[1,2,3], **W. Bradley Knox**[1,2,5], **Julie Shah**[3],
**Scott Niekum**[2,4], **Peter Stone**[2,6], **Alessandro Allievi**[1,2]

[1]Bosch, [2]The University of Texas at Austin, [3]MIT CSAIL,
[4]The University of Massachusetts Amherst, [5]Google Research, [6]Sony AI
{serenabooth, julie_a_shah}@csail.mit.edu, {bradknox,pstone}@cs.utexas.edu,
sniekum@cs.umass.edu, alessandro.allievi@us.bosch.com

## Abstract

In reinforcement learning (RL), a reward function that aligns exactly with a task's true performance metric is often sparse. For example, a true task metric might encode a reward of 1 upon success and 0 otherwise. These sparse task metrics can be hard to learn from, so in practice they are often replaced with alternative dense reward functions. These dense reward functions are typically designed by experts through an ad hoc process of trial and error. In this process, experts manually search for a reward function that improves performance with

the practice of reward design seldom adheres to this adage.[1] Instead, reward functions are typically designed through an ad hoc process of trial and error. In a survey of 24 expert RL practitioners, we found that 92% reported using trial and error to design their most recent reward function (Apdx. A). This finding echos the literature: Knox et al. (2021) found that, in a survey of RL for autonomous driving, all of the surveyed publications reported designing reward functions by trial and error. Despite the prevalence of trial-and-error reward design, the consequences of this process remain al-

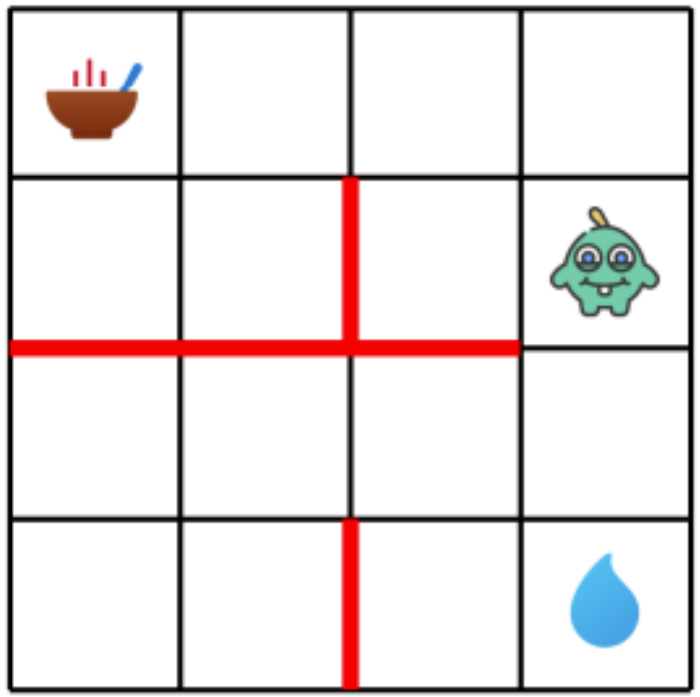
# Discussion

- Answer D

# Findings

- Shaping
- Overfitting
- RL Evaluation
- How to interpret human-specified rewards
- Implications for RL, neuroscience, …

$$r(\mathtt{H} \wedge \mathtt{T}) = a \qquad r(\mathtt{H} \wedge \neg\mathtt{T}) = b$$
$$r(\neg\mathtt{H} \wedge \mathtt{T}) = c \qquad r(\neg\mathtt{H} \wedge \neg\mathtt{T}) = d$$



"I am hungry and not thirsty."

Figure 1: Hungry Thirsty ($4 \times 4$ grid). Food and water are each located in a corner. Red walls are impassable. The current reward-relevant state is abbreviated as $\mathtt{H} \wedge \neg\mathtt{T}$, which corresponds to the agent being hungry and not thirsty. The $6 \times 6$ grid variant is depicted in Singh, Lewis, and Barto (2009).

# Inverse Reward Design

**Dylan Hadfield-Menell** **Smitha Milli** **Pieter Abbeel*** **Stuart Russell** **Anca D. Dragan**
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94709
{dhm, smilli, pabbeel, russell, anca}@cs.berkeley.edu

## Abstract

Autonomous agents optimize the reward function we give them. Wha
know is how hard it is for us to design a reward function that actua
what we want. When designing the reward, we might think of so
training scenarios, and make sure that the reward will lead to the rig
in *those* scenarios. Inevitably, agents encounter *new* scenarios (e.g., r
terrain) where optimizing that same reward may lead to undesired be

# Assisted Robust Reward Design

**Jerry Zhi-Yang He**
Electrical Engineering and Computer Science
University of California Berkeley
hzyjerry@berkeley.edu

**Anca D. Dragan**
Electrical Engineering and Computer Science
University of California, Berkeley
anca@berkeley.edu

**Abstract:** Real-world robotic tasks require complex reward functions. When we define the problem the robot needs to solve, we pretend that a designer specifies this complex reward exactly, and it is set in stone from then on. In practice, however, reward design is an *iterative* process: the designer chooses a reward, eventually encounters an "edge-case" environment where the reward incentivizes the wrong behavior, revises the reward, and repeats. What would it mean to rethink robotics problems to formally account for this iterative nature of reward design? We propose that the robot not take the specified reward for granted, but rather have *uncertainty* about it, and account for the future design iterations as

# EUREKA: HUMAN-LEVEL REWARD DESIGN VIA CODING LARGE LANGUAGE MODELS

**Yecheng Jason Ma**[1 2 ✉] , **William Liang**[2], **Guanzhi Wang**[1 3], **De-An Huang**[1], **Osbert Bastani**[2], **Dinesh Jayaraman**[2], **Yuke Zhu**[1 4], **Linxi "Jim" Fan**[1 ✉†], **Anima Anandkumar**[1 3 †]

[1]NVIDIA, [2]UPenn, [3]Caltech, [4]UT Austin; [†]Equal advising

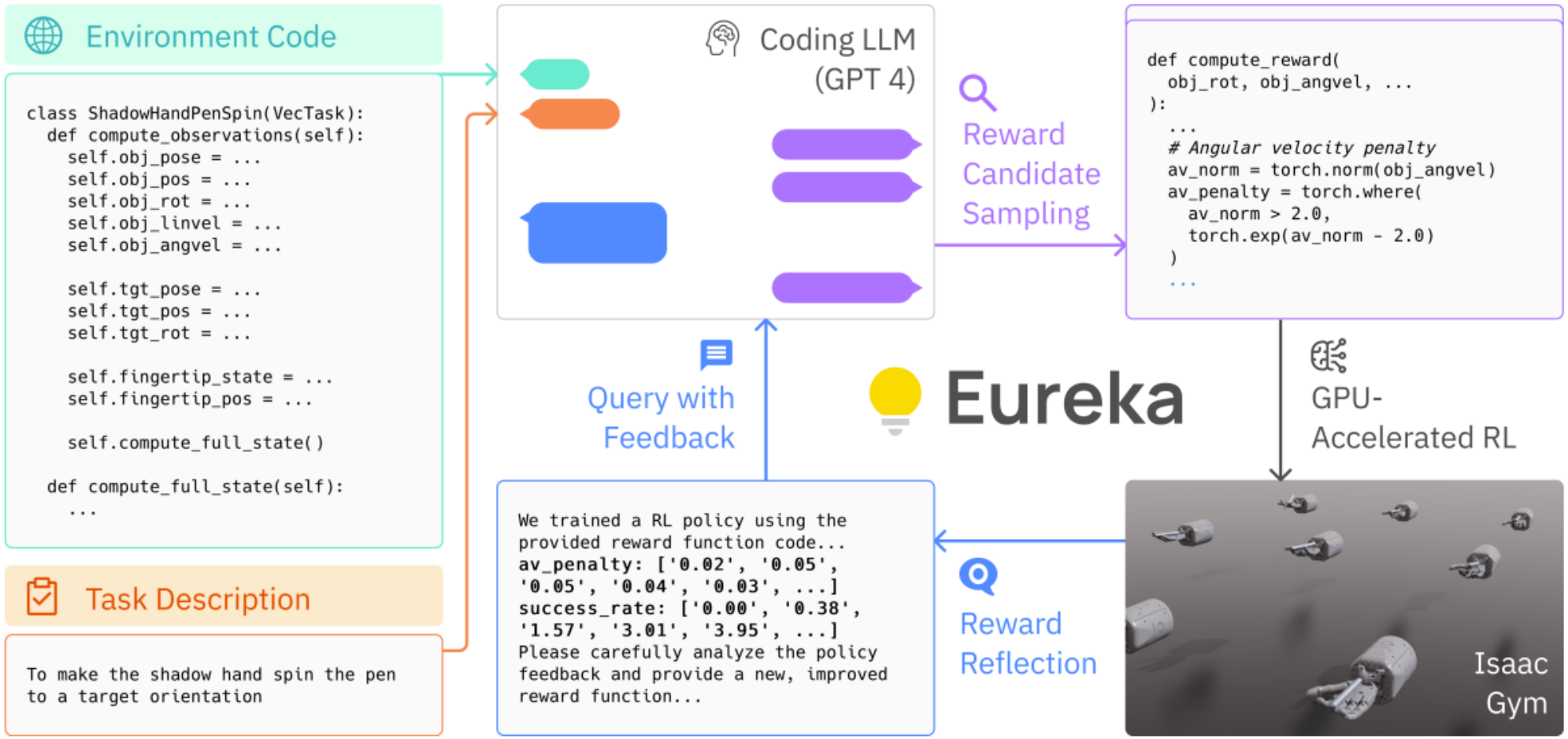
https://eureka-research.github.io

https://eureka-research.github.io/

## ABSTRACT

Large Language Models (LLMs) have excelled as high-level semantic planners for sequential decision-making tasks. However, harnessing them to learn complex low-level manipulation tasks, such as dexterous pen spinning, remains an open problem. We bridge this fundamental gap and present EUREKA, a human-level reward design algorithm powered by LLMs. EUREKA exploits the remarkable zero-shot generation, code-writing, and in-context improvement capabilities of state-of-the-

# Discussion


Environment Code
class ShadowHandPenSpin(VecTask):
def compute_observations(self):
self.obj_pose = ...
self.obj_pos = ...
self.obj_rot = ...
self.obj_linvel = ...
self.obj_angvel = ...
self.tgt_pose = ...
self.tgt_pos = ...
self.tgt_rot = ...
self.fingertip_state = ...
self.fingertip_pos = ...
self.compute_full_state()
def compute_full_state(self):
...
Task Description
To make the shadow hand spin the pen to a target orientation
Coding LLM (GPT 4)
Reward Candidate Sampling
def compute_reward(
obj_rot, obj_angvel, ...
):
...
# Angular velocity penalty
av_norm = torch.norm(obj_angvel)
av_penalty = torch.where(
av_norm > 2.0,
torch.exp(av_norm - 2.0)
)
...
GPU-Accelerated RL
Isaac Gym
Reward Reflection
We trained a RL policy using the provided reward function code...
av_penalty: ['0.02', '0.05', '0.05', '0.04', '0.03', ...]
success_rate: ['0.00', '0.38', '1.57', '3.01', '3.95', ...]
Please carefully analyze the policy feedback and provide a new, improved reward function...
Query with Feedback
Eureka

```
def compute_reward(object_rot, goal_rot, object_angvel, object_pos, fingertip_pos):
    # Rotation reward
    rot_diff = torch.abs(torch.sum(object_rot * goal_rot, dim=1) - 1) / 2
-   rotation_reward_temp = 20.0
+   rotation_reward_temp = 30.0                                   Changing hyperparameter
    rotation_reward = torch.exp(-rotation_reward_temp * rot_diff)

    # Distance reward
+   min_distance_temp = 10.0
    min_distance = torch.min(torch.norm(fingertip_pos - object_pos[:, None], dim=2), dim=1).values
-   distance_reward = min_distance
+   uncapped_distance_reward = torch.exp(-min_distance_temp * min_distance)
+   distance_reward = torch.clamp(uncapped_distance_reward, 0.0, 1.0)   Changing functional form

-   total_reward = rotation_reward + distance_reward
+   # Angular velocity penalty                                    Adding new component
+   angvel_norm = torch.norm(object_angvel, dim=1)
+   angvel_threshold = 0.5
+   angvel_penalty_temp = 5.0
+   angular_velocity_penalty = torch.where(angvel_norm > angvel_threshold,
+       torch.exp(-angvel_penalty_temp * (angvel_norm - angvel_threshold)), torch.zeros_like(angvel_norm))
+
+   total_reward = 0.5 * rotation_reward + 0.3 * distance_reward - 0.2 * angular_velocity_penalty

    reward_components = {
        "rotation_reward": rotation_reward,
        "distance_reward": distance_reward,
+       "angular_velocity_penalty": angular_velocity_penalty,
    }

    return total_reward, reward_components
```

# Findings, Results, and Conclusions

- Poor correlation between human and LLM rewards
- Evolutionary search seems to work really well
- Curriculum learning
- Human feedback and “warm starting” with human reward
- Prompting magic ☺
- Interpretable code