

# PrefCLM: Enhancing Preference-based Reinforcement Learning with Crowdsourced Large Language Models

Ruiqi Wang<sup>1†</sup>, Dezhong Zhao<sup>1,2†</sup>, Ziqin Yuan<sup>1</sup>, Ike Obi<sup>1</sup>, and Byung-Cheol Min<sup>1</sup>

**Abstract**—Preference-based reinforcement learning (PbRL) is emerging as a promising approach to teaching robots through human comparative feedback, sidestepping the need for complex reward engineering. However, the substantial volume of feedback required in existing PbRL methods often lead to reliance on synthetic feedback generated by scripted teachers. This approach necessitates intricate reward engineering again and struggles to adapt to the nuanced preferences particular to human-robot interaction (HRI) scenarios, where users may have unique expectations toward the same task. To address these challenges, we introduce PrefCLM, a novel framework that utilizes crowdsourced large language models (LLMs) as simulated teachers in PbRL. We utilize Dempster-Shafer Theory to fuse individual preferences from multiple LLM agents at the score level, efficiently leveraging their diversity and collective intelligence. We also introduce a human-in-the-loop pipeline that facilitates collective refinements based on user interactive feedback. Experimental results across various general RL tasks show that PrefCLM achieves competitive performance compared to traditional scripted teachers and excels in facilitating more natural and efficient behaviors. A real-world user study ( $N=10$ ) further demonstrates its capability to tailor robot behaviors to individual user preferences, significantly enhancing user satisfaction in HRI scenarios. Videos, prompts, example evaluation functions, and other supplementary details can be found at <https://prefclm.github.io>.

## I. INTRODUCTION

Reinforcement learning (RL) has demonstrated remarkable success in the field of robot learning. However, the effectiveness of learned robot policies heavily hinges on the design of reward functions [1]. Crafting an informative and dense reward function is intricate, necessitating substantial effort and domain expertise, especially for tasks with long horizons [2] or those involving subtle social interactions [3]. Moreover, the issue of reward hacking that leads to unexpected robot behaviors remains a potential concern [4].

In light of these challenges, preference-based reinforcement learning (PbRL) [3], [5]–[13] has emerged as a promising solution by eliminating the intricate process of reward engineering. In the PbRL paradigm, a human teacher delivers continuous comparative feedback, expressing preferences between two distinct robot trajectories, each consisting of multiple state-action pairs across time steps. A reward model is then learned to align with the given preferences,

<sup>1</sup>SMART Laboratory, Department of Computer and Information Technology, Purdue University, West Lafayette, IN, USA. [wang5357, yuan460, obi, minb]@purdue.edu.

<sup>2</sup>College of Mechanical and Electrical Engineering, Beijing University of Chemical Technology, Beijing, China. DZ\_Zhao@buct.edu.cn.

This work involved human subjects or animals in its research. The authors confirm that all human/animal subject research procedures and protocols are exempt from review board approval.

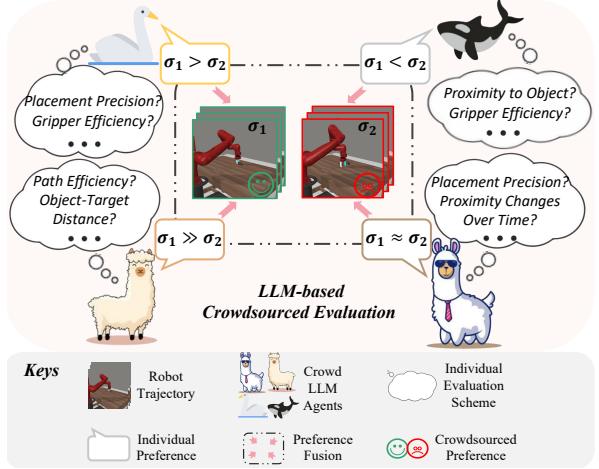


Fig. 1. Conceptual illustration of the LLM-based crowdsourced evaluation. LLM instructors are symbolized by animal icons, with distinct species presenting varied LLM architectures. These crowd instructors, through their unique evaluative criteria and reasoning, determine individual preferences for robot trajectories, which are fused to formulate a unified crowdsourced preference used for PbRL process.

guiding robot behaviors to more closely match human expectations through subsequent RL training. Despite recent advancements aimed at enhancing feedback efficiency, the requirement for a large volume of human feedback to achieve satisfactory performance still restricts the widespread application of PbRL. As a workaround, most studies [5]–[11] tend to adopt synthetic preferences from a scripted teacher for training effective robot policies. In practice, such a teacher bases preferences on the cumulative reward values derived from a meticulously designed reward function for each robot trajectory in the comparison [7].

While leveraging handcrafted reward functions indirectly through scripted teachers may mitigate reward hacking [6], this strategy inadvertently reintroduces the need for reward engineering, contradicting the original purpose of PbRL. Moreover, the Markovian nature of these reward functions limits the evaluation of scripted teachers to immediate state-action pairs within the robot trajectory, potentially neglecting crucial holistic information over the trajectory and thus leading to a high misalignment with actual human preferences [13]. Moreover, the reliance on scripted teachers with static reward functions hinders the flexibility to capture and reflect the individual preferences of users, which is crucial for human-robot interaction (HRI) task scenarios [14].

Meanwhile, large language models (LLMs) have demonstrated promising in-context reasoning capabilities across various areas of robotics, functioning as task planners [15], reward designers [16], [17], and communication intermediates [18].

aries in HRI [18]. This success inspires us to consider LLM agents as alternative synthetic teachers in PbRL, particularly for evaluating robot trajectories through more prolonged and adaptable patterns. However, the reasoning proficiency of an individual LLM, characterized by inherent uncertainty and variability [19], [20], may not always match the expertise and consistency of human instructors or even engineered scripted teachers across varied task scenarios. This challenge mirrors a familiar predicament in human society, where complex, multifaceted problems often surpass the expertise of any individual. In such situations, crowdsourcing has emerged as an efficient solution [21]. By harnessing the group wisdom of a distributed network of individuals, each contributing their unique knowledge and perspectives, crowdsourcing significantly enhances the utility of contributions from a vast pool of non-experts [22].

Drawing from this parallel, we introduce PrefCLM, a novel framework that leverages crowdsourced LLMs as simulated teachers in PbRL. As illustrated in Fig. 1, our main idea is to pivot from the engineered scripted teacher prevalent in prior research to an LLM-based crowdsourced evaluation paradigm, which harnesses the diversity and collective intelligence of multiple LLM agents, in evaluating robot behaviors. To efficiently manage the uncertainties and conflicts within the crowdsourcing, we further integrate the Dempster-Shafer Theory (DST) [23] to seamlessly fuse individual evaluative feedback from LLM instructors as a crowdsourced preference used in the PbRL process. Moreover, we introduce a human-in-the-loop (HITL) module that enables PrefCLM to incorporate language-based user feedback on iteratively learned robot policies during the PbRL process and accordingly adapt the evaluation patterns, leading to synthetic crowdsourced feedback that more closely aligns with unique user preferences in HRI scenarios.

The overview of the proposed PrefCLM with four main components is depicted in Fig. 2. Our key contributions in this letter can be summarized as:

- We propose PrefCLM as a novel solution to enhancing the feedback efficiency in PbRL with synthetic preferences from crowdsourced LLM agents, eliminating the need of intricate reward engineering for creating scripted teachers and enabling a more dynamic and nuanced evaluation pattern. PrefCLM can also refine synthetic preferences to match unique and subtle user expectations more closely through interactive user inputs.
- We introduce crowdsourcing and DST principles to efficiently leverage the collective intelligence and variegation of multiple LLM agents in evaluating robot behaviors and reflecting user individual expectations, elevating the quality and relevance of the crowdsourced preference feedback.
- Our extensive experiments demonstrate that PrefCLM can achieve competitive or superior task performance compared to expert-tuned scripted teachers across various benchmark general RL tasks. As supported by a user study (N=10), PrefCLM with user interactive inputs can facilitate more personalized robot behaviors, significantly boosting user satisfaction in real-world HRI scenarios.

## II. BACKGROUND AND PRELIMINARY

### A. Preference-based RL and Scripted Teachers.

PbRL has emerged as a promising approach to circumvent the complexities with reward function engineering in traditional RL [6]. Its goal is to learn a preference-aligned reward model  $\hat{R}_\psi$ , typically a neural network parameterized by  $\psi$ , based on comparative feedback between segments of robot trajectories (or behaviors) provided by a human instructor.

A robot trajectory segment  $\sigma$  is defined as a time-indexed sequence of states and actions within a specified length  $L$ :  $\{(s_1, a_1), \dots, (s_L, a_L)\}$ . Then a human teacher is periodically asked to express their preference  $\Lambda \in \{0, 1, 0.5\}$  between a pair of trajectory segments  $\sigma^0, \sigma^1$ , where  $\Lambda = 1$  indicates  $\sigma^1$  is preferred,  $\Lambda = 0$  denotes the opposite, and  $\Lambda = 0.5$  means equally preferred. Each preference query is stored in a replay buffer  $\mathcal{B}$  as  $(\sigma^0, \sigma^1, \Lambda)$ . Then, a Bradley-Terry model-based preference predictor [5] is employed to estimate the preference probabilities as:

$$\mathcal{P}_\psi [\sigma^1 \succ \sigma^0] = \frac{\exp\left(\sum_t \hat{R}_\psi(s_t^1, a_t^1)\right)}{\sum_{i \in \{0,1\}} \exp\left(\sum_t \hat{R}_\psi(s_t^i, a_t^i)\right)} \quad (1)$$

where  $\mathcal{P}_\psi [\sigma^1 \succ \sigma^0]$  represents the probability that trajectory segment  $\sigma^1$  is preferred over  $\sigma^0$ .

The reward learning problem is then regarded as a supervised learning problem, with the objective of minimizing the binary cross-entropy loss between the preferences provided by humans  $\Lambda$  and those by the reward model  $\hat{R}_\psi$  as:

$$\mathcal{L}_\psi = - \sum_{(\sigma^1, \sigma^0, \Lambda) \in \mathcal{B}} \Lambda(1) \log \mathcal{P}_\psi [\sigma^1 \succ \sigma^0] + \Lambda(0) \log \mathcal{P}_\psi [\sigma^0 \succ \sigma^1] \quad (2)$$

However, achieving robust performance typically necessitates an impractical amount of human feedback. To mitigate this, most existing works [5]–[11] have adopted a scripted teacher that provides synthetic preferences derived from an engineered reward function  $R$  for training. Formally, a scripted teacher determines synthetic preferences by comparing the cumulative reward values of each trajectory as:

$$\Lambda = \begin{cases} 0 & \text{If } \sum_{t=1}^L R(s_t^0, a_t^0) > \sum_{t=1}^L R(s_t^1, a_t^1) \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

Additionally, incorporating the stochastic labeling strategies proposed by [7], a scripted teacher can offer more realistic feedback with irrationalities, e.g., equally preferred or indecisions. Nevertheless, this approach leads back to the necessity for sophisticated reward engineering to implement such a teacher, which contradicts the original intention of PbRL. Moreover, such a scripted teacher lacks flexibility and may fail to capture long-term information in trajectories due to the static and Markovian nature of the underlying reward function [13]. In contrast, PrefCLM harnesses the collective intelligence of multiple LLMs to evaluate robot behaviors, excelling at providing a more nuanced evaluation pattern without the need for reward engineering and adapting flexibly to unique user expectations in HRI scenarios.

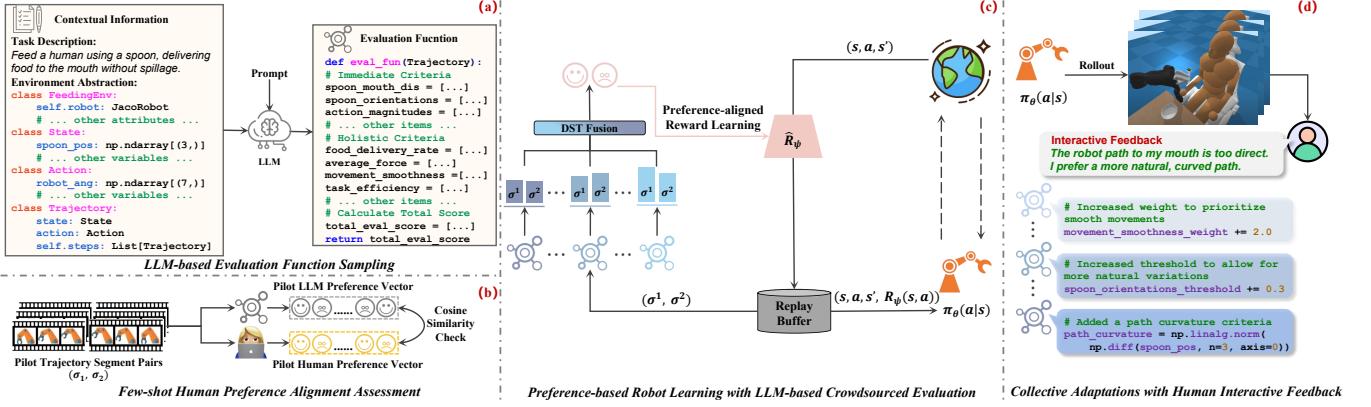


Fig. 2. Overview of the PrefCLM framework. (a) Given task-specific contextual information and prompts, multiple code-based evaluation functions are sampled from crowd LLM agents (Section III-A). (b) A cosine similarity check module then filters the sampled evaluation functions, selecting those that align with few-shot expert preferences within a specified tolerance (optional, Section III-B). (c) Evaluative scores are continuously assigned by these selected evaluation functions to pairs of robot trajectories. These scores are aggregated through Dempster-Shafer Theory (DST) fusion to form crowdsourced preferences, which are used for the reward learning in PbRL (Section III-C). (d) Crowd LLM agents can also collectively adapt and refine their evaluation functions based on user interactive inputs given periodically in HRI scenarios (optional, Section III-D).

### B. Coding LLMs for Robotics.

To effectively integrate LLMs into robot systems or learning frameworks, recent works [15]–[18] tend to utilize the programmatic output from LLMs rather than high-level language-based instructions, improving interpretability and enhancing control over robot behaviors. In line with this trend, our approach in PrefCLM shifts from having LLMs provide direct preferences in the interaction-intensive PbRL process to generating evaluation functions in code. This ensures a consistent and transparent evaluative rationale, and reduces the LLM usage costs. The most relevant to ours are recent studies [16], [17] that employ an LLM agent to assist reward design in RL. Different from their works, we prompt each LLM agent to generate non-Markovian evaluation functions that consider not only the current state-action pair but also the contextual information over the entire trajectory in PbRL. Furthermore, we employ a crowdsourcing strategy and strategic fusion methods to capitalize on the collective intelligence of multiple LLM agents whereas their works rely on a single one.

### C. Compound LLM Systems and Crowdsourcing.

Single LLM agent often faces reasoning deficiencies, such as hallucinations and limited perspectives, making them less effective for complex tasks [20]. To address this, compound LLM systems have been proposed, wherein multiple LLMs act in different roles to handle various aspects of the problem [19] or engage in debates to produce a more comprehensive output [20]. While we follow this general approach, our method differs in that we incorporate the repeated labeling concept from crowdsourcing [22], where each LLM independently handles the same role of generating evaluation functions for robot trajectories. Furthermore, differing from the prevalent majority voting method [22] used to aggregate decisions (e.g., preferences) in crowdsourcing or compound systems, we employ DST [23] for integrating the outputs at the score level from each LLM (e.g., evaluative scores) in evaluating robot behaviors. This approach enables a more nuanced and informative fusion of the crowd preferences from

LLMs, considering the confidence of a given preference and addressing associated indecision or conflicts more efficiently.

## III. METHODOLOGY

In this section, we present PrefCLM, a framework utilizing crowdsourced LLM agents to enable zero-shot or few-shot synthetic preference generation within PbRL. Furthermore, we introduce a HITL module in PrefCLM that can incorporate human interactive inputs to dynamically adapt and refine the evaluation mechanisms, leading to more personalized robot behaviors in HRI scenarios.

### A. LLM-based Evaluation Function Sampling

Assuming  $n$  LLM agents comprise the crowd, which can be either homogeneous or heterogeneous, the initial step in PrefCLM involves sampling multiple code-based evaluation functions from these agents. Each function, represented as  $\mathcal{F} : \dot{\sigma} \rightarrow \dot{\rho}$ , calculates an evaluative score  $\dot{\rho}$  for a robot trajectory segment  $\dot{\sigma}$  within a given task scenario. We prompt LLM agents to produce functions that regard the whole robot trajectory as the evaluative object, instead of single state-action pairs as considered in scripted teachers. Such functions aim to evaluate the holistic patterns and changes across time-steps within the entire trajectory in addition to the immediate effectiveness of each state-action pair, ensuring a more nuanced evaluation akin to humans.

To this end, contextual information along with prompts  $p$  (Appx. I) is provided to LLM agents. As shown in Fig. 2, the contextual information includes a concise task description  $l$ , which delineates the objectives of the task (Appx. II), and an environment abstraction  $e$ , structured as a hierarchy of Pythonic classes following [17] (Appx. III). This Pythonic representation provides a high-level and essential overview of the task environment, with particular emphasis on states, actions, and trajectories. Utilizing this information, the crowd LLM agents individually engage in-context reasoning to generate  $n$  evaluation functions as:

$$\mathcal{F}_1, \dots, \mathcal{F}_n \sim \text{LLMs}(l, e, p) \quad (4)$$

Empirically, we observe that the evaluation functions, even those generated by homogeneous agents, exhibit diversity (Appx. IV). This variation manifests in several ways, such as differing task-related criteria, assorted definitions for the same criteria, and varying priorities assigned to these criteria (e.g., different weighting schemes). Our PrefCLM capitalizes on this diversity, leveraging unique understanding that each LLM agent brings to the task and leading to a richer and more comprehensive evaluation process.

### B. Few-shot Expert Preference Alignment

Few-shot expert involvement has proven to enhance the performance of LLM-based systems [15], [18]. In line with this, our method integrates a few-shot generation pattern in addition to the zero-shot evaluation function generation previously described. This is facilitated by a few-shot expert preference alignment module. Specifically, an expert preference vector,  $\Upsilon_{expert} = [\Lambda^1, \dots, \Lambda^j]$ , is constructed by a human expert giving preferences  $\Lambda$  for a set of  $j$  pilot trajectory segment pairs. Similarly, for each evaluation function  $\mathcal{F}$  sampled, we can derive an LLM preference vector,  $\Upsilon_{\mathcal{F}} = [\hat{\Lambda}^1, \dots, \hat{\Lambda}^j]$ , where  $\hat{\Lambda}$  represents a pseudo preference for a segment pair, determined by the relative magnitude of evaluative scores that  $\mathcal{F}$  assigns to each segment in the pair. Then we can calculate the cosine similarity score, denoted as  $\vartheta \in [-1, 1]$ , between the  $\Upsilon_{expert}$  and  $\Upsilon_{\mathcal{F}}$  as:

$$\vartheta = \frac{\Upsilon_{expert} \cdot \Upsilon_{\mathcal{F}}}{\|\Upsilon_{expert}\| \times \|\Upsilon_{\mathcal{F}}\|} \quad (5)$$

Cosine similarity quantifies the angular closeness between these two vectors irrespective of their magnitude, effectively capturing the alignment in preference orientation between the expert and the LLM agent in high-dimensional spaces. Evaluation functions that achieve a similarity score exceeding a threshold  $\hat{\vartheta}$  are selected for further use. This ensures that selected functions are closely aligned with expert preferences while maintaining a diverse set of evaluative perspectives.

### C. Crowdsourced Evaluation and Preference Fusion

The subsequent phase entails employing the zero-shot generated or few-shot selected evaluation functions for the crowdsourced evaluation. This involves aggregating individual preferences  $\hat{\Lambda}$  from  $n$  crowd LLM agents to form a unified crowdsourced preference  $\bar{\Lambda}$ , which is then utilized for the reward learning process as described in Eqs. 1 and 2. For this purpose, we adopt DST [23] as the fusion strategy at the level of evaluative scores.

Consider a pair of robot trajectory segments,  $\sigma^0$  and  $\sigma^1$ , to be evaluated by the LLM crowdsourcing. We define the frame of discernment in DST as  $\Theta = \{\sigma^0, \sigma^1, \{\sigma^0, \sigma^1\}\}$ , which represents the possible decisions of preferring  $\sigma^0$ , preferring  $\sigma^1$ , or having an equal or indeterminate preference between them, respectively. For the  $k^{th}$  LLM agent, we first normalize its assigned scores  $\rho_k^0$  and  $\rho_k^1$  for  $\sigma^0$  and  $\sigma^1$  as:

$$\hat{\rho}_k^0 = \frac{\rho_k^0}{\rho_k^0 + \rho_k^1}; \hat{\rho}_k^1 = \frac{\rho_k^1}{\rho_k^0 + \rho_k^1} \quad (6)$$

Normalization is crucial here since the scale of the evaluative score from each agent is unknown and variable. We then

calculate the mass function, which reflects the  $k^{th}$  agent's belief toward each decision in  $\Theta$  as:

$$\begin{aligned} m_k(\{\sigma^0, \sigma^1\}) &= \varphi \times (1 - |\hat{\rho}_k^0 - \hat{\rho}_k^1|) \\ m_k(\sigma^0) &= \hat{\rho}_k^0 \times (1 - m_k(\{\sigma^0, \sigma^1\})) \\ m_k(\sigma^1) &= \hat{\rho}_k^1 \times (1 - m_k(\{\sigma^0, \sigma^1\})) \end{aligned} \quad (7)$$

Here,  $\varphi \in [0, 1]$  is a parameter in DST that determines the maximum level of indecision assignable by any agent. Then the fused mass function for any decision  $X$  within  $\Theta$  is calculated as:

$$\bar{m}(X) = \bigoplus_{k=1}^n m_k(X) \quad (8)$$

where  $\bigoplus$  denotes the operation of Dempster's rule of combination that fuses individual belief iteratively as:

$$m_{1:k}(X) = \frac{1}{1 - K_{1:k}} \sum_{A \cap B = X} m_{1:k-1}(A) \cdot m_k(B) \quad (9)$$

$$K_{1:k} = \sum_{A \cap B = \emptyset} m_{1:k-1}(A) \cdot m_k(B) \quad (10)$$

where  $K_{1:k}$  represents the conflict measure after combining up to the  $k^{th}$  LLM agent.  $A$  and  $B$  are subsets of the possible preference decisions within  $\Theta$ .

Subsequently, the decision in  $\Theta$  with the highest aggregated belief after considering all  $n$  LLM agents is determined as the final decision,  $\bar{X}$ , as:

$$\bar{X} = \arg\max_{X \in \Theta} \bar{m}(X) \quad (11)$$

Finally, we can obtain the crowdsourced preference  $\bar{\Lambda}$  as:

$$\bar{\Lambda} = \begin{cases} 0 & \text{if } \bar{X} = \sigma^0 \\ 1 & \text{if } \bar{X} = \sigma^1 \\ 0.5 & \text{otherwise} \end{cases} \quad (12)$$

The crowdsourced preference is generated each time the PbRL algorithm queries for a preference between a pair of trajectory segments rolled out from the current robot policy. This synthetic preference adheres to the same format as those obtained from human teachers, denoted as  $\Lambda$  in Section II. As a result, the PrefCLM framework can be seamlessly integrated into the reward learning phase in any PbRL frameworks, offering a plug-and-play solution that harnesses the collective intelligence of multiple LLM agents in evaluating robot behaviors. This integration enhances the feedback efficiency of the PbRL process, eliminating the need for extensive human efforts in providing feedback or engineering scripted teachers.

### D. Collective Adaptations with Interactive Inputs for HRI

The crowdsourced preference aggregated from crowd LLM agents can be semantically consistent with the objectives of general RL tasks, yet it may be *misspecific* in practice, especially in HRI scenarios where expectations of the same task may vary across users [14]. For instance, when a robotic arm is feeding food to a patient, some may expect the robot to move slowly and cautiously, while others may prefer a more efficient motion to minimize disruptions.

To better adapt to such unique user expectations, we propose an extension of PrefCLM tailored for HRI scenarios. The first intuitive step is to incorporate specific user

expectations, such as “I prefer the robot to move slowly and cautiously when feeding me”, into the prompts for the evaluation function sampling stage in PrefCLM as additional contextual information. This allows LLM agents to integrate specific evaluation criteria that reflect user preferences, such as motion caution, into their evaluation functions.

However, humans rarely articulate their intentions clearly or comprehensively in a single or few instances, and their expectations may change as the interaction evolves [24]. To address this, as illustrated in Fig. 2d, we introduce an additional HITL module in PrefCLM capable of dynamically incorporating interactive inputs from users into the evaluation paradigms of the LLM crowd, thus fostering more personalized robot behaviors. In addition to initializing PrefCLM with user expectations as described above, users can observe the robot trajectory segments rolled out from the iterative robot policies during training sessions and offer real-time interactive feedback, such as “The robot’s path to my mouth is too direct. I prefer a more natural, curved path.”

This interactive feedback is conveyed to the crowd of LLM agents, triggering a collective refinement of their evaluation functions to better align with the given user preferences. Each LLM agent may interpret and incorporate the user feedback differently based on their current evaluation functions. For instance, as shown in Fig. 2d, one agent might focus on adjusting existing related criteria, such as increasing the importance of *motion smoothness* to encourage gentle and continuous movements or modifying the threshold for acceptable *end-effector orientation* to allow for more variations in the angle of the spoon. Another agent might introduce a new evaluation criterion, such as *path curvature*, which assigns higher scores to paths that exhibit a more fluid and curved shape, if such criteria were previously lacking in their evaluation function. This diversity in adjustments with the DST fusion could ensure that the refined pattern of the LLM-based crowdsourced evaluation capture a broad spectrum of the given interactive feedback.

The updated evaluation functions are then used to re-score the robot trajectory segments  $\sigma$ , both those stored in the replay buffer and newly generated ones, producing new preference labels  $\bar{\Lambda}$  that better align with the expressed feedback of the user. These updated preference labels are subsequently used to retrain the reward model  $\hat{R}_\psi$ , which in turn guides the robot learning process towards generating behaviors that more closely match the user expectations.

## IV. EXPERIMENTS AND RESULTS

### A. Experiments on General RL Tasks

We first designed experiments to evaluate the performance of PrefCLM under zero-shot and few-shot generation conditions on general RL tasks across various simulation benchmarks. Extensive ablation studies were also conducted.

1) *Environments*: The general RL tasks we consider include locomotion tasks from the DeepMind Control Suite benchmark [25], specifically the Walker, Cheetah, and Quadruped tasks, as well as manipulation tasks from the

Meta-World benchmark [26], specifically the Button Press, Door Unlock, and Drawer Open tasks.

2) *Baselines*: We compared our method against two baselines for generating synthetic preferences in PbRL:

- Scripted Teachers. This baseline determines preferences towards robot trajectories based on expert-tuned reward functions, representing a common practice in existing PbRL works [5]–[11].
- PrefEVO. We built this baseline by adapting the Eureka framework [16], initially designed for reward design with an LLM agent, to PbRL scenarios. The framework uses evolutionary search to enhance LLM outputs by sampling and evaluating multiple reward functions, selecting the one achieving highest scores from pre-defined fitness functions as the final reward function. The LLM can also refine the final reward function by self-reflecting on the policy performance metrics, e.g., the success rate and its changes on manipulation tasks.

To adapt it to PbRL settings, we replace the prompts and contextual information with those used in PrefCLM, and substitute the fitness functions with the cosine similarity scores mentioned in Section III-B as the search criterion, since such functions are not available in our setup. This represents a state-of-the-art single-LLM-based baseline.

3) *Ablation Studies*: To further investigate the impact of crowdsourcing and DST fusion mechanisms within our framework, we conducted additional ablation studies. Specifically, we aimed to assess how the number and composition (homogeneous or heterogeneous) of the LLM agents in the crowdsourcing affect the performance, and how DST fusion benefits PrefCLM under these conditions.

To this end, we implemented an ablation model named MajCLM, which uses majority voting, a common method in compound AI systems, to fuse individual preferences from crowd LLM agents instead of using DST fusion. We tested PrefCLM and MajCLM on the Walker and Button Press tasks with different numbers of homogeneous LLM agents (3, 10, 20) under zero-shot settings, using GPT-4 [27] as the LLM backbone. Results of a single LLM agent are also reported. We further tested them under heterogeneous settings with a combination of GPT-4, Claude-Opus, and Llama-3-70B.

4) *Implementation Details*: All methods were implemented using PEBBLE [6], a benchmark PbRL algorithm, as the backbone network with consistent hyperparameters, and disagreement sampling was used for feedback querying. Feedback volume was determined in line with prior research [8], [12], assigning 600 queries for tasks such as Walker and Cheetah, 1200 for Quadruped, 2500 for Button Press and Door Unlock, and 5000 for Drawer Open.

For the Scripted Teachers baseline, benchmark-provided reward functions were used, and we adopted the optimal labeling strategies for each task as reported in [7]. For all LLM-based methods, GPT-4, specifically the gpt-4o model, was employed as the LLM backbone, unless stated otherwise. The crowd size for PrefCLM was set at 10 when compared to other baselines. In implementing the

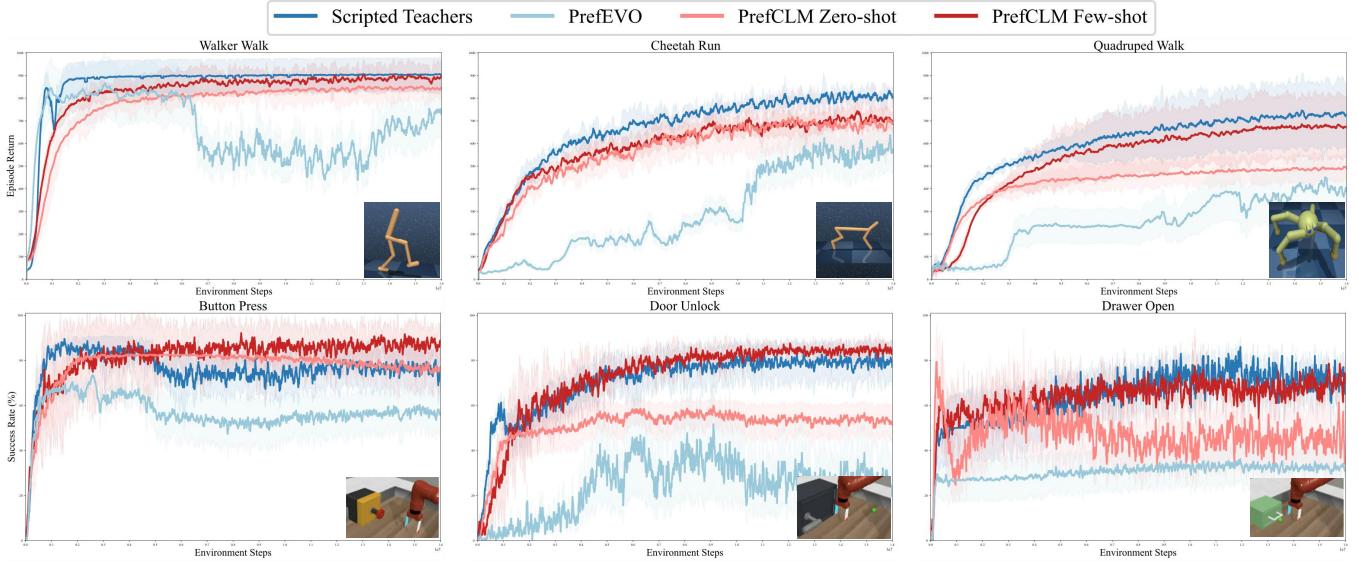


Fig. 3. Learning curves on general RL tasks, measured in episode returns for locomotion tasks and success rates for manipulation tasks. The solid line represents the mean, while the shaded area indicates the standard deviation across five runs.

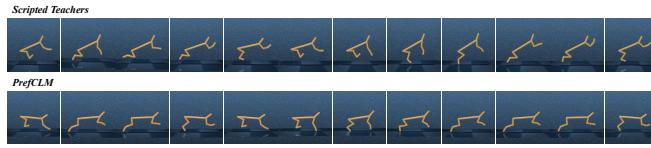


Fig. 4. Locomotion behaviors learned by the Scripted Teachers (top) and PrefCLM (bottom) on the Cheetah Run task.

few-shot expert preference alignment in PrefCLM and the evolutionary search in PrefEVO, we provided 15 pilot preferences for trajectories rolled out from the policy during the unsupervised learning phase in PEBBLE for each task. For PrefEVO, following the original work [16], we conducted 5 rounds of evolutionary search with a sampling size of 16 each time. The threshold  $\hat{\vartheta}$  of cosine similarly in PrefCLM was set to 0.5 for locomotion tasks and 0.6 for manipulation tasks. Additionally, the indecision parameter  $\varphi$  in DST was set to 0.3 for all tasks.

Following [12], we used ground-truth reward returns as the performance metric for locomotion tasks, while for manipulation tasks, we reported the success rate as provided by the benchmark. Each model underwent five independent runs on each task, with results reported as an average accompanied by the standard deviation. All experiments were conducted on a workstation with three NVIDIA RTX 4090 GPUs.

**5) Results and Analysis:** Figure 3 illustrates the learning curves of PrefCLM in comparison to various baselines. We can observe that PrefCLM, under zero-shot or few-shot generation modes, achieves performance comparable to expert-tuned Scripted Teachers across most locomotion and manipulation tasks in terms of episode returns or final success rates and convergence speed. Notably, PrefCLM even outperforms the baseline on the Button Press and Door Unlock tasks. Although PrefCLM does not surpass the Scripted Teachers in locomotion tasks, it leads to more natural and efficient behaviors, as illustrated in Figure 4, where the robot behaviors from PrefCLM use two legs to run like a real animal, whereas those from Scripted Teachers only use the back leg to jump. This advantage is attributed

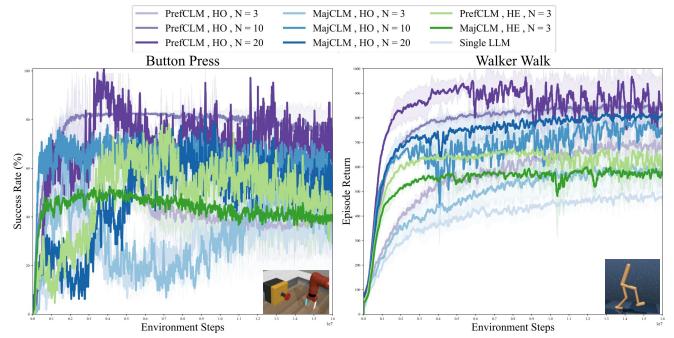


Fig. 5. Results of ablation studies in terms of learning curves following Fig. 3. HO: homogeneous setting; HE: heterogeneous setting; N: number of LLM agents in the crowd.

to the LLM-generated evaluation functions which consider a broader range of criteria, such as motion efficiency, in addition to the basic success criteria typically considered by reward functions in Scripted Teachers. Additionally, the Markovian nature of reward functions in Scripted Teachers may neglect historical information that is critical for capturing the full context of robot behaviors. Overall, these observations suggest that PrefCLM can generate high-quality and more comprehensive evaluation patterns without the need for expert engineering, showcasing its potential as an efficient simulated teacher in PbRL.

Furthermore, while the zero-shot mode of PrefCLM can achieve satisfactory results, a notable performance gap is observed in incompletely solved tasks such as Quadruped and Drawer Open. This highlights the benefits of the few-shot expert alignment process, especially when the task demands nuanced understanding.

On the other hand, we note a significant performance disparity between PrefEVO and PrefCLM, with PrefEVO failing to surpass even the zero-shot mode of PrefCLM. This discrepancy can be attributed to the limitations of the evolutionary search method in the context of PbRL. Unlike traditional RL, where reward outputs directly dictate robot policy, PbRL involves a two-tiered process: evaluative scores



Fig. 6. Illustration of the user study environment. A participant seated in a chair interacts with a Jaco assistive robot arm that feeds using a spoon.

are first used to learn a reward model, which then informs the policy. Consequently, the self-reflection metrics employed in PrefEVO may not effectively represent the quality of the evaluative functions, as these metrics are not directly linked to the overall effectiveness of the feedback mechanism in PbRL. Therefore, the evolved evaluation functions may not translate into improved performance in the PbRL setting. Additionally, the time and resource-intensive nature of evolutionary search makes it less efficient. These demonstrate that the crowdsourcing approach in PrefCLM is a more fitting and efficient method for enhancing LLM-based evaluations in PbRL, leveraging the collective wisdom of crowd LLMs.

Moreover, as depicted in Fig. 5, we observe that the PrefCLM (purples) outperforms MajCLM (blues) under different crowd numbers in homogeneous settings and in the heterogeneous setting. This can be attributed to the fact that the majority voting fusion method in MajCLM may not efficiently handle conflicts and uncertainties, which increase as the size and heterogeneity of the crowd agents grow. Majority voting at the decision level may overlook the nuances and disagreements among the agents. In contrast, the DST fusion in PrefCLM effectively manages these complexities by handling the diverse beliefs and resolving conflicts at the score level, offering a more robust fusion mechanism.

Additionally, as shown in Fig. 5, we can observe the effects of crowd LLM agent size. For both models, performance generally increases with the crowd size. However, this performance enhancement diminishes as the number continues to grow. For example, moving from  $n=1$  (single LLM) to  $n=3$  and then to  $n=10$  shows continuous improvement, but the differences between  $n=10$  and  $n=20$  are minimal, with performance remaining nearly the same. This trend aligns with findings in compound inference systems [28].

### B. Real-world User Study

To evaluate the personalization and user satisfaction enabled by PrefCLM with the HITL module in realistic HRI tasks, we further carried out a real-world user study.

1) *Setup:* The HRI task we selected is the Feeding task from the Assistive Gym benchmark [29]. In this task, a robot arm is tasked with delivering a spoon holding food, represented as small spheres, to the mouth of a human seated in a chair without spilling. For the real-world setup, as illustrated in Fig. 6, we employed the Kinova Jaco assistive robotic arm and a RealSense D435 camera, integrated with

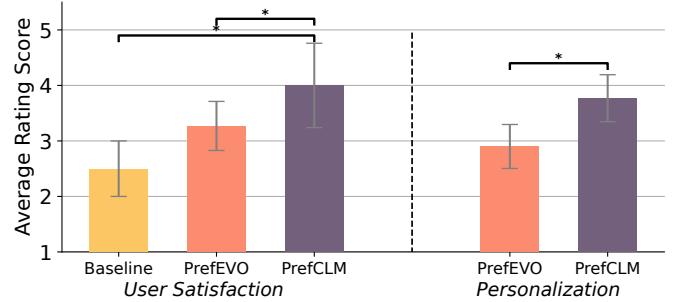


Fig. 7. Average rating scores from participants in terms of satisfaction and personalization. Baseline refers to the pre-trained policy.  $*: p < 0.01$ .

a face landmark detection algorithm [30] and YOLOv7 [31] to track the human's head and mouth and the position of the spoon respectively. An emergency switch is conveniently located and easily accessible to investigators, ensuring safety throughout the experiment.

2) *Baselines:* To guarantee the safety of participants in our user study, we initially pre-trained a policy following the guidance from the benchmark [29] in simulation. This step is to ensure the basic functionality, i.e., successfully reaching the human mouth. Additionally, we configured the simulation environment to closely mimic the real-world setup, including the position of the chair, human shape, and the mounting of the robotic arm, in order to minimize the sim-to-real gap. Following this, we fine-tuned the policy using synthetic feedback from PrefCLM (few-shot,  $n=10$ ) equipped with the HITL module. Additionally, PrefEVO, also equipped with the same module, served as another baseline representing state-of-the-art single-LLM-based HITL adaptation approaches [17].

3) *Experimental Design:* We recruited 10 participants (3 females, 7 males) from faculty/staff and students at BUCT, with an average age of 24.5 years ( $SD = 2.78$ ). All participants reviewed and signed a consent form, which outlined the study objectives, procedures, potential risks, and their rights as participants. Initially, participants were introduced to the Feeding task and asked to verbally express their general expectations. These expectations, along with other contextual information and prompts, were used to generate the initial evaluation functions by PrefCLM and PrefEVO for the initial fine-tuning training of the pre-trained policy respectively. For each model, we periodically (every  $4 \times 10^6$  environment steps, approximately 2 hours) rolled out the learned robot policy to the physical Kinova Jaco robotic arm, for a total of three times. Each time, each participant provided their interactive feedback, which was utilized to refine the evaluation functions.

After all the training, participants engaged in physical interactions with the robot policies fine-tuned by PrefCLM and PrefEVO, as well as with the pre-trained baseline policy. Each participant interacted with each policy three times in a randomized sequence and was not informed about which policy was active to prevent any bias in their responses. Following each interaction, participants were asked to rate the robot behaviors of the three policies in terms of satisfaction, using a Likert scale ranging from 1 (strongly disagree) to 5 (strongly agree). They were also asked to rate the level of

personalization resulting from PrefCLM and PrefEVO using the same scale. To evaluate statistical differences between our model and others, two-sample t-tests were conducted.

*4) Results and Analysis:* As shown in Fig 7, we can observe that the policies fine-tuned by PrefEVO and PrefCLM achieve higher ratings in terms of overall user satisfaction compared to the pre-trained policy. This demonstrates that static reward functions, which serve as the basis of Scripted Teachers, lack the flexibility to capture subtle and unique user expectations in customizable HRI tasks. In contrast, the LLM-based approaches can more efficiently adapt to user expectations by incorporating interactive user verbal inputs and adjusting evaluation patterns in PbRL accordingly. This highlights the potential of LLM agents to bridge the gap between human intentions and robot behaviors.

More importantly, PrefCLM significantly outperforms PrefEVO in both satisfaction and personalization. This can be attributed to the fact that PrefEVO relies on the adaptation of the evaluation function by a single LLM agent, whereas PrefCLM adopts a collective adaptation pattern within the crowdsourcing framework. Each LLM agent refines its evaluation criteria based on user interactive inputs, covering a broader spectrum of user-specific feedback. This demonstrates that the crowdsourcing and fusion strategies proposed in PrefCLM effectively harness collective intelligence to better align robot behaviors with unique user preferences.

## V. CONCLUSION

In this letter, we present PrefCLM that leverages crowdsourced LLMs for generating synthetic feedback in PbRL. PrefCLM is also capable of capturing the subtle intentions of users through interactive feedback. By leveraging the diversity and collective intelligence of multiple LLM agents with crowdsourcing and DST fusion methods, PrefCLM facilitates a dynamic and nuanced evaluation of robot behaviors, catering specifically to the unique expectations of users in personalized HRI scenarios. Our experimental results demonstrate that PrefCLM not only competes with expert-tuned scripted teachers in terms of performance in general RL tasks but also offers a more natural and intuitive method for specifying preferences in personalized HRI scenarios.

## REFERENCES

- [1] H. Zhu *et al.*, “The ingredients of real world robotic reinforcement learning,” in *Int. Conf. on Learning Representations*, 2019.
- [2] R. Wang, D. Zhao *et al.*, “Initial task allocation in multi-human multi-robot teams: An attention-enhanced hierarchical reinforcement learning approach,” *IEEE Robotics and Automation Letters*, 2024.
- [3] R. Wang *et al.*, “Feedback-efficient active preference learning for socially aware robot navigation,” in *2022 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*. IEEE, 2022, pp. 11 336–11 343.
- [4] D. Hadfield-Menell, S. Milli *et al.*, “Inverse reward design,” *Advances in neural information processing systems*, vol. 30, 2017.
- [5] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in neural information processing systems*, vol. 30, 2017.
- [6] K. Lee, L. M. Smith, and P. Abbeel, “Pebble: Feedback-efficient interactive reinforcement learning via relabeling experience and unsupervised pre-training,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 6152–6163.
- [7] K. Lee, L. Smith, A. Dragan, and P. Abbeel, “B-pref: Benchmarking preference-based reinforcement learning,” *Neural Information Processing Systems (NeurIPS)*, 2021.
- [8] J. Park, Y. Seo, J. Shin, H. Lee, P. Abbeel, and K. Lee, “Surf: Semi-supervised reward learning with data augmentation for feedback-efficient preference-based reinforcement learning,” in *International Conference on Learning Representations*, 2021.
- [9] A. Hiranaka, M. Hwang, S. Lee, C. Wang, L. Fei-Fei, J. Wu, and R. Zhang, “Primitive skill-based robot learning from human evaluative feedback,” in *2023 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*. IEEE, 2023, pp. 7817–7824.
- [10] K. Metcalf, M. Sarabia, N. Mackraz, and B.-J. Theobald, “Sample-efficient preference-based reinforcement learning with dynamics aware rewards,” in *7th Annual Conference on Robot Learning*, 2023.
- [11] M. Liu *et al.*, “Task decoupling in preference-based reinforcement learning for personalized human-robot interaction,” in *2022 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2022, pp. 848–855.
- [12] R. Liu, F. Bai, Y. Du, and Y. Yang, “Meta-reward-net: Implicitly differentiable reward learning for preference-based reinforcement learning,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 22 270–22 284, 2022.
- [13] C. Kim, J. Park *et al.*, “Preference transformer: Modeling human preferences using transformers for rl,” in *The Eleventh International Conference on Learning Representations*, 2022.
- [14] B. Irfan, A. Ramachandran *et al.*, “Personalization in long-term human-robot interaction,” in *2019 14th ACM/IEEE International Conference on Human-Robot Interaction*. IEEE, 2019, pp. 685–686.
- [15] I. Singh *et al.*, “Progprompt: Generating situated robot task plans using large language models,” in *2023 IEEE Int. Conf. Robotics and Automation*. IEEE, 2023, pp. 11 523–11 530.
- [16] Y. J. Ma *et al.*, “Eureka: Human-level reward design via coding large language models,” in *International Conference on Learning Representations*, 2024.
- [17] T. Xie *et al.*, “Text2reward: Reward shaping with language models for reinforcement learning,” in *International Conference on Learning Representations*, 2024.
- [18] J. Wu, R. Antonova *et al.*, “Tidybot: Personalized robot assistance with large language models,” in *2023 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2023, pp. 3546–3553.
- [19] S. Hong *et al.*, “Metagpt: Meta programming for multi-agent collaborative framework,” in *The Twelfth International Conference on Learning Representations*, 2023.
- [20] C.-M. Chan *et al.*, “Chateval: Towards better llm-based evaluators through multi-agent debate,” in *The Twelfth International Conference on Learning Representations*, 2023.
- [21] J. Howe *et al.*, “The rise of crowdsourcing,” *Wired magazine*, vol. 14, no. 6, pp. 176–183, 2006.
- [22] V. S. Sheng and J. Zhang, “Machine learning with crowdsourcing: A brief summary of the past research and future directions,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 9837–9843.
- [23] G. Shafer, “Dempster-shafer theory,” *Encyclopedia of artificial intelligence*, vol. 1, pp. 330–331, 1992.
- [24] N. Gasteiger *et al.*, “Factors for personalization and localization to optimize human–robot interaction: A literature review,” *International Journal of Social Robotics*, vol. 15, no. 4, pp. 689–701, 2023.
- [25] Y. Tassa, *et al.*, “Deepmind control suite,” *arXiv preprint arXiv:1801.00690*, 2018.
- [26] T. Yu *et al.*, “Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning,” in *Conference on robot learning*. PMLR, 2020, pp. 1094–1100.
- [27] J. Achiam, S. Adler *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [28] L. Chen *et al.*, “Are more llm calls all you need? towards scaling laws of compound inference systems,” *arXiv:2403.02419*, 2024.
- [29] Z. Erickson *et al.*, “Assistive gym: A physics simulation framework for assistive robotics,” in *2020 IEEE Int. Conf. Robotics and Automation*. IEEE, 2020, pp. 10 169–10 176.
- [30] C. Lugaressi *et al.*, “Mediapipe: A framework for building perception pipelines,” *arXiv preprint arXiv:1906.08172*, 2019.
- [31] C.-Y. Wang *et al.*, “Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2023, pp. 7464–7475.

# APPENDIX

In the appendix, we provide details about various aspects of our work, including the prompts used for generating evaluation functions, details of task descriptions and environment abstractions, example evaluation functions for the general RL tasks, and details about the pre-training and fine-tuning process in our user study.

## TABLE OF CONTENTS

I Prompts Used for Generating Evaluation Functions .....	1
• I.1 Initial System Prompt	
• I.2 Initial User Prompt	
• I.3 Code Error Feedback Prompt	
II Task Descriptions .....	2
• II.1 Description of Tasks Used in Benchmark	
III Environment Abstraction .....	2
• III.1 Example Environment Abstraction for Walker Task	
• III.2 Example Environment Abstraction for Button Press Task	
IV Example of LLM-based Evaluation Functions .....	4
• IV.1 Expert-engineered Reward Function for Walker Task	
• IV.2 LLM-based Evaluation Functions for Walker Task	
• IV.3 Expert-engineered Reward Function for Button Press Task	
• IV.4 LLM-based Evaluation Functions for Button Press Task	
V The Feeding Task in the User Study .....	14
• V.1 Pre-trained Robot Policy	
• V.2 Fine-tuning Process	

## APPENDIX I PROMPTS USED FOR GENERATING EVALUATION FUNCTIONS

In this section, we present the prompts utilized for LLM-based evaluation functions sampling in the PrefCLM. We begin by initializing the LLM agent with a specific role and job description:

### Prompt 1: Initial System

You are an expert evaluator specializing in preference-based reinforcement learning for robots. Your task is to design a sophisticated Python evaluation function that accurately scores robot trajectories within a specific reinforcement learning environment. This function is critical for guiding the robot's learning process and optimizing its task performance.

Your evaluation function should:

- Use only the variables available in the robot's trajectory, which consists of multiple state-action pairs across different time steps.
- Return a single float value as the overall score, where higher scores indicate better performance.
- Incorporate two key components:
  - *Immediate evaluation*: Assess individual state-action pairs at each time step.
  - *Holistic evaluation*: Analyze patterns and trends across the entire trajectory.

Next, we provide specific contextual information, including the task description (Appendix II) and environment abstraction (Appendix III), along with additional requirements for generating evaluation functions:

### Prompt 2: Initial User

I need you to generate the evaluation function for the following task: {Task Description}. The Pythonic class-like environment abstraction is {Environment Abstraction}.

Proceed as follows:

- Analyze the task requirements and environment step-by-step.
- Develop a function with the signature `def evaluate_trajectory(trajectory: Trajectory) -> float` that returns only the `final_score`.
- Include comments in your code to explain your reasoning and design choices.

Additional Requirements:

- The evaluation function must be a standalone function, suitable for integration into a class in another Python file.
- It must not contain any intra-class calls.
- Provide concrete, well-reasoned initial threshold values and weights. Avoid placeholders.

Additionally, in practice, although not frequent, sometimes the LLM agent may generate code with errors such as syntax errors or runtime issues (e.g., shape mismatch). In line with previous works [16], [17], we utilize the traceback message from code execution to prompt the LLM agent to fix the bug and provide an executable evaluation function if errors occur. The prompt for handling code errors is shown below:

**Prompt 3: Code Error Feedback**

Executing the evaluation function code you generated above has the following error: {traceback\_msg}. Please fix the bug and provide a new, evaluation function.

## APPENDIX II TASK DESCRIPTIONS

Following [16], [17], we use the task descriptions provided by the benchmark environments as the *{Task Description}* in the prompts. These are summarized in Table I.

TABLE I  
DESCRIPTION OF EACH TASK.

Task	Descriptions
Walker Walk	Control the Walker robot to walk steadily in the forward direction, maintaining balance and speed.
Cheetah Run	Control the Cheetah robot to run swiftly in the forward direction, optimizing for speed and stability.
Quadruped Walk	Control the Quadruped robot to walk in the forward direction, ensuring coordination among all four legs for smooth movement.
Button Press	Instruct the robot to press a button located along the y-axis, requiring precise positioning and force application.
Door Unlock	Instruct the robot to unlock a door by rotating the lock mechanism counter-clockwise, requiring fine motor skills and dexterity.
Drawer Open	Instruct the robot to open a drawer by pulling its handle, requiring a firm grip and controlled pulling force.

## APPENDIX III ENVIRONMENT ABSTRACTION

To effectively generate evaluation functions within a task environment, LLM agents must understand how attributes of the robot and environment are represented, including the configuration of robots and objects, trajectory information, and available functions. To this end, following [17], we employ a compact representation in Pythonic style, which utilizes Python classes, typing, and comments. This approach offers a higher level of abstraction compared to listing all environment-specific information in a list or table format, enabling the creation of general, reusable prompts across different environments. Additionally, Pythonic representation is prevalent in the pre-training data of LLMs, facilitating the LLM's understanding of the environment. Example environment abstractions for the Walker and Button Press tasks are provided below.

**Example Environment Abstraction for the Walker Task**

```
class WalkerEnv:
    physics: Physics
    task: PlanarWalker
    control_timestep: float = 0.025 # Time interval for each control update.
    time_limit: float = 25 # Maximum duration for each episode in seconds.

    def step(self, action: np.ndarray):
        """Executes one timestep of the environment's dynamics with the given action and updates
        the trajectory."""

```

```

        pass

    def reset(self):
        """Resets the environment to an initial state and returns the first observation."""
        pass

    def get_trajectory(self) -> Trajectory:
        """Returns the trajectory data collected during an episode, including states, actions, and
        observations."""
        pass

class Physics:
    def torso_upright(self) -> float:
        """Calculates the cosine of the angle between the torso's z-axis and the vertical,
        indicating how upright the torso is."""
        pass

    def torso_height(self) -> float:
        """Returns the vertical position of the torso in meters, which helps monitor the walker's
        balance."""
        pass

    def horizontal_velocity(self) -> float:
        """Measures the horizontal speed of the walker's center of mass, reflecting movement
        efficiency."""
        pass

    def orientations(self) -> np.ndarray:
        """Returns an array of planar orientations for body segments, aiding in posture analysis.
        """
        pass

    def velocity(self) -> np.ndarray:
        """Returns a comprehensive velocity vector for all body parts, including both linear and
        angular velocities."""
        pass

class PlanarWalker:
    trajectory: Trajectory
    _move_speed: float # Desired movement speed, varies with the task ('stand', 'walk', 'run').

    def get_observation(self, physics: Physics) -> collections.OrderedDict:
        """Compiles observational data from physics simulations, crucial for real-time decision-
        making."""
        pass

    def get_state(self, physics: Physics) -> dict:
        """Aggregates current state information from physics, providing a detailed snapshot of
        dynamic conditions."""
        pass

class Trajectory:
    def __init__(self, max_length=time_limit):
        self.states: deque # queue of states, max length 25
        self.actions: deque # queue of actions, max length 25
        self.observations: deque # queue of observations, max length 25

    def add_step(self, state: dict, action: np.ndarray, observation: np.ndarray):
        # Add a step to the trajectory

    def __len__(self) -> int:
        # Return the number of steps in the trajectory

```

### Example Environment Abstraction for the Button Press Task

```

class SawyerButtonPressEnvV2(gym.Env):
    def __init__(self):
        self.robot: Robot # the Sawyer robot in the environment
        self.button: RigidObject # the button object in the environment
        self.goal_position: np.ndarray[(3,)] # 3D position of the goal (button pressed position)

```

```

        self.trajectory: Trajectory # stores the trajectory of the episode

    def reset(self) -> np.ndarray:
        # Reset the environment and return initial observation

    def step(self, action: np.ndarray) -> tuple:
        # Perform one step and return (observation, reward, done, info)

    def get_trajectory(self) -> Trajectory:
        # Return the recorded trajectory

class Robot:
    def __init__(self):
        self.ee_position: np.ndarray[(3,)] # 3D position of the end-effector
        self.joint_positions: np.ndarray[(7,)] # 7 joint positions of Sawyer robot
        self.joint_velocities: np.ndarray[(7,)] # 7 joint velocities of Sawyer robot

class RigidObject:
    def __init__(self):
        self.position: np.ndarray[(3,)] # 3D position of the object (button)
        self.quaternion: np.ndarray[(4,)] # quaternion of the object (button)

class Trajectory:
    def __init__(self, max_length=25):
        self.states: deque # queue of states, max length 25
        self.actions: deque # queue of actions, max length 25
        self.observations: deque # queue of observations, max length 25

    def add_step(self, state: dict, action: np.ndarray, observation: np.ndarray):
        # Add a step to the trajectory

    def __len__(self) -> int:
        # Return the number of steps in the trajectory

class State:
    def __init__(self):
        self.robot: Robot # state of the robot
        self.button: RigidObject # state of the button

```

## APPENDIX IV

### EXAMPLE OF LLM-BASED EVALUATION FUNCTIONS

In this section, we demonstrate example LLM-based evaluation functions during experiments, especially for the Walker and Button Press Tasks. Note that all example evaluation functions were sampled from multiple independent calls from the gpt-4 model. For comparison, we also provide the expert-engineered reward functions for these tasks, which serve as the evaluation basis of the Scripted Teachers baseline. The expert-tuned reward functions and sample functions are shown below.

#### Expert-engineered Reward Function for the Walker Task

```

def get_reward(self, physics):
    """Returns a reward to the agent."""
    standing = rewards.tolerance(physics.torso_height(),
                                 bounds=(_STAND_HEIGHT, float('inf')),
                                 margin=_STAND_HEIGHT/2)
    upright = (1 + physics.torso_upright()) / 2
    stand_reward = (3*standing + upright) / 4
    if self._move_speed == 0:
        return stand_reward
    else:
        move_reward = rewards.tolerance(physics.horizontal_velocity(),
                                         bounds=(self._move_speed, float('inf')),
                                         margin=self._move_speed/2,
                                         value_at_margin=0.5,
                                         sigmoid='linear')
    return stand_reward * (5*move_reward + 1) / 6

```

## Example LLM-based Evalution Functions for the Walker Task

```
####Evalution Function Sample 1####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    def gaussian(x, mu, sigma):
        return np.exp(-0.5 * ((x - mu) / sigma) ** 2)

    def sigmoid(x, k=1):
        return 1 / (1 + np.exp(-k * x))

    if len(trajectory.states) == 25:
        a = trajectory.actions[0] is None
        if not a:
            immediate_scores = []
            stability_scores = []
            efficiency_scores = []
            goal_achievement_scores = []

            for i, (state, action) in enumerate(zip(trajectory.states, trajectory.actions)):
                # Immediate evaluation
                upright_score = (1 + state['torso_upright']) / 2
                height_score = gaussian(state['torso_height'], mu=trajectory.task._STAND_HEIGHT, sigma=0.1)
                action_magnitude = np.linalg.norm(action)
                action_score = gaussian(action_magnitude, mu=0, sigma=0.5)

                immediate_score = (
                    0.4 * upright_score +
                    0.4 * height_score +
                    0.2 * action_score
                )
                immediate_scores.append(immediate_score)

            # Stability evaluation
            if i >= 2:
                height_changes = [trajectory.states[j+1]['torso_height'] - trajectory.states[j]['torso_height'] for j in range(i-2, i)]
                upright_changes = [trajectory.states[j+1]['torso_upright'] - trajectory.states[j]['torso_upright'] for j in range(i-2, i)]
                height_stability = 1 - abs(height_changes[1] - height_changes[0])
                upright_stability = 1 - abs(upright_changes[1] - upright_changes[0])
                stability_scores.append((height_stability + upright_stability) / 2)

            # Efficiency evaluation
            if i >= 1:
                velocity_change = state['horizontal_velocity'] - trajectory.states[i-1]['horizontal_velocity']
                efficiency_scores.append(gaussian(velocity_change, mu=0, sigma=0.1))

            # Goal achievement evaluation
            target_speed = trajectory.task._move_speed
            if target_speed == 0:
                stand_score = gaussian(state['torso_height'], mu=trajectory.task._STAND_HEIGHT, sigma=0.1)
                upright_score = (1 + state['torso_upright']) / 2
                goal_achievement_scores.append((stand_score + upright_score) / 2)
            else:
                goal_achievement_scores.append(gaussian(state['horizontal_velocity'], mu=target_speed, sigma=target_speed / 4))

            # Holistic evaluation
            overall_stability = np.mean(stability_scores)
            overall_efficiency = np.mean(efficiency_scores)
            goal_progression = np.polyfit(range(len(goal_achievement_scores)), goal_achievement_scores, 1)[0]
            goal_progression_score = sigmoid(goal_progression, k=10)
            motion_consistency = 1 - np.std(efficiency_scores)
            task_completion = np.mean(goal_achievement_scores[-10:])

            holistic_score = (
                0.2 * overall_stability +
                0.2 * overall_efficiency +
                0.2 * goal_progression_score +
                0.2 * motion_consistency +
                0.2 * task_completion
            )

        else:
            holistic_score = 0.5

    else:
        holistic_score = 0.0

    return holistic_score
```

```

        0.2 * goal_progression_score +
        0.2 * motion_consistency +
        0.2 * task_completion
    )

    # Combine immediate and holistic scores
    final_score = 0.4 * np.mean(immediate_scores) + 0.6 * holistic_score
else:
    final_score = 0
else:
    final_score = 0

return final_score

####Evaluation Function Sample 2####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    """
    Evaluate a given robot trajectory and return an overall score.
    Higher scores indicate better performance.

    :param trajectory: The Trajectory object containing states, actions, and observations.
    :return: A single float value representing the overall score.
    """
    # Define weight constants
    WEIGHT_UPRIGHT = 1.0
    WEIGHT_HEIGHT = 0.5
    WEIGHT_VELOCITY = 1.0
    WEIGHT_ENERGY = 0.2
    WEIGHT_STABILITY = 0.8
    WEIGHT_PROGRESS = 0.5

    # Immediate evaluation variables
    upright_scores = []
    height_scores = []
    velocity_scores = []
    energy_scores = []

    # Holistic evaluation variables
    total_distance = 0
    height_variation = []
    upright_variation = []

    target_speed = trajectory.task._move_speed
    stand_height = _STAND_HEIGHT
    max_height_threshold = 0.8 * stand_height # Threshold below which walker is considered fallen

    previous_state = None
    if len(trajectory.states) == 25:
        a = trajectory.actions[0] == None
        if type(a) != bool:
            for step in range(len(trajectory)):
                state = trajectory.states[step]
                action = trajectory.actions[step]
                observation = trajectory.observations[step]

                # Immediate evaluations
                torso_upright = state['torso_upright']
                torso_height = state['torso_height']
                horizontal_velocity = state['horizontal_velocity']

                # Upright score: closer to 1 is better
                upright_scores.append(torso_upright)

                # Height score: closer to stand height is better
                height_scores.append(1 - abs(torso_height - stand_height) / stand_height)

                # Velocity score: closer to target speed is better
                velocity_scores.append(1 - abs(horizontal_velocity - target_speed) / target_speed)

```

```

# Energy score: lower action magnitudes are better
energy_scores.append(1 - np.linalg.norm(action) / np.sqrt(len(action)))

# Holistic evaluations
if previous_state is not None:
    distance_travelled = abs(state['horizontal_velocity']) * _CONTROL_TIMESTEP
    total_distance += distance_travelled

height_variation.append(torso_height)
upright_variation.append(torso_upright)

previous_state = state

# Compute immediate evaluation scores
mean_upright_score = np.mean(upright_scores)
mean_height_score = np.mean(height_scores)
mean_velocity_score = np.mean(velocity_scores)
mean_energy_score = np.mean(energy_scores)

# Compute holistic evaluation scores
stability_score = 1 - (np.std(height_variation) / stand_height + np.std(
    upright_variation)) / 2
progress_score = total_distance / (len(trajectory) * _CONTROL_TIMESTEP *
    target_speed)

# Calculate overall score with weights
final_score = (WEIGHT_UPRIGHT * mean_upright_score +
               WEIGHT_HEIGHT * mean_height_score +
               WEIGHT_VELOCITY * mean_velocity_score +
               WEIGHT_ENERGY * mean_energy_score +
               WEIGHT_STABILITY * stability_score +
               WEIGHT_PROGRESS * progress_score)
else:
    final_score = 0
else:
    final_score = 0
return final_score

####Evaluation Function Sample 3####

def evaluate_trajectory(trajectory: 'Trajectory') -> float:
    """
    Evaluate the robot's trajectory and return an overall score.

    Args:
        trajectory (Trajectory): The trajectory to be evaluated.

    Returns:
        float: The overall score of the trajectory.
    """

    # Initialize score components
    immediate_scores = []
    total_distance = 0.0
    total_effort = 0.0
    last_velocity = None
    smoothness_penalty = 0.0
    consistency_penalty = 0.0

    # Define constants for evaluation thresholds and weights
    TOROS_UPRIGHT_THRESHOLD = 0.8 # Close to fully upright
    TOROS_HEIGHT_THRESHOLD = 1.0 # Close to the stand height
    TARGET_VELOCITY = 1.0 # Target walking speed
    EFFICIENCY_WEIGHT = 0.1 # Weight for efficiency in the overall score
    SMOOTHNESS_WEIGHT = 0.2 # Weight for smoothness in the overall score
    FALL_PENALTY = -100.0 # Penalty for falling

    for step in range(len(trajectory)):
        state = trajectory.states[step]
        action = trajectory.actions[step]
        observation = trajectory.observations[step]

```

```

# Immediate evaluation
upright_score = max(state['torso_upright'], 0) # Prefer upright posture
height_score = max(0, 1 - abs(state['torso_height'] - trajectory.task._STAND_HEIGHT))
speed_score = max(0, 1 - abs(state['horizontal_velocity']) - trajectory.task._move_speed)
immediate_scores.append(immediate_score)

immediate_score = (upright_score + height_score + speed_score) / 3
immediate_scores.append(immediate_score)

# Holistic evaluation components
total_distance += state['horizontal_velocity'] * trajectory.task.control_timestep
total_effort += np.sum(np.square(action))

if last_velocity is not None:
    smoothness_penalty += np.linalg.norm(state['velocity'] - last_velocity)
last_velocity = state['velocity']

# Consistency in joint orientations
consistency_penalty += np.var(observation['orientations'])

# Holistic evaluation
average_immediate_score = np.mean(immediate_scores)
efficiency_score = total_distance / (total_effort + 1e-6) # Avoid division by zero
smoothness_score = 1 / (smoothness_penalty + 1e-6) # Smoothness as inverse of penalty
consistency_score = 1 / (consistency_penalty + 1e-6) # Consistency as inverse of penalty

final_score = (
    average_immediate_score +
    EFFICIENCY_WEIGHT * efficiency_score +
    SMOOTHNESS_WEIGHT * smoothness_score +
    (1 - EFFICIENCY_WEIGHT - SMOOTHNESS_WEIGHT) * consistency_score
)

# Penalize for falling
if trajectory.states[-1]['torso_height'] < 0.8 * trajectory.task._STAND_HEIGHT:
    final_score += FALL_PENALTY

return final_score

####Evaluation Function Sample 4####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    """
    Evaluate the robot's trajectory and return an overall score.

    Args:
        trajectory (Trajectory): The trajectory to be evaluated.

    Returns:
        float: The overall score of the trajectory.
    """

    # Constants
    TARGET_HEIGHT = trajectory.task._STAND_HEIGHT # 1.2 meters
    TARGET_SPEED = trajectory.task._move_speed # Depends on the current task: 0, 1 m/s, or 8 m/s
    FALL_THRESHOLD = 0.8 * TARGET_HEIGHT # Around 0.96 meters

    # Initialize score components
    stability_score = 0
    speed_score = 0
    efficiency_score = 0
    smoothness_score = 0
    progress_score = 0
    consistency_score = 0

    # Initialize counters
    steps = len(trajectory)
    distance_covered = 0
    previous_velocity = None
    previous_orientations = None

```

```

# Immediate Evaluation
for i, (state, action, observation) in enumerate(zip(trajectory.states, trajectory.actions,
    trajectory.observations)):
    # Stability
    torso_upright = state['torso_upright']
    torso_height = state['torso_height']
    stability_score += max(0, torso_upright) * max(0, (torso_height - FALL_THRESHOLD) / (
        TARGET_HEIGHT - FALL_THRESHOLD))

    # Speed
    horizontal_velocity = state['horizontal_velocity']
    speed_score += max(0, 1 - abs(horizontal_velocity - TARGET_SPEED) / TARGET_SPEED)

    # Efficiency
    efficiency_score += 1 - np.linalg.norm(action) / np.sqrt(3) # Normalized to [0, 1]

    # Progress
    if i > 0:
        distance_covered += horizontal_velocity * trajectory.task.control_timestep

    # Smoothness (difference in orientations and velocities between consecutive steps)
    if previous_orientations is not None:
        orientation_diff = np.linalg.norm(state['orientations'] - previous_orientations)
        velocity_diff = np.linalg.norm(state['velocity'] - previous_velocity)
        smoothness_score += 1 / (1 + orientation_diff + velocity_diff)

    previous_velocity = state['velocity']
    previous_orientations = state['orientations']

# Normalize immediate scores
if steps > 0:
    stability_score /= steps
    speed_score /= steps
    efficiency_score /= steps
    smoothness_score /= (steps - 1) if steps > 1 else 1

# Holistic Evaluation
# Progress
progress_score = distance_covered

# Consistency (variation in orientations and velocities)
orientation_variation = np.var([state['orientations'] for state in trajectory.states],
    axis=0).mean()
velocity_variation = np.var([state['velocity'] for state in trajectory.states], axis=0).
    mean()
consistency_score = 1 / (1 + orientation_variation + velocity_variation)

# Combine scores
final_score = (
    0.3 * stability_score +
    0.3 * speed_score +
    0.1 * efficiency_score +
    0.1 * smoothness_score +
    0.1 * progress_score +
    0.1 * consistency_score
)

return final_score

```

## Expert-engineered Reward Function for the Button Press Task

```

def compute_reward(
    self, action: npt.NDArray[Any], obs: npt.NDArray[np.float64]
) -> tuple[float, float, float, float, float, float]:
    assert (
        self._target_pos is not None
    ), "'reset_model()' must be called before 'compute_reward()'."
    del action
    obj = obs[4:7]
    tcp = self.tcp_center

```

```

tcp_to_obj = float(np.linalg.norm(obj - tcp))
tcp_to_obj_init = float(np.linalg.norm(obj - self.init_tcp))
obj_to_target = abs(self._target_pos[1] - obj[1])

tcp_closed = max(obs[3], 0.0)
near_button = reward_utils.tolerance(
    tcp_to_obj,
    bounds=(0, 0.05),
    margin=tcp_to_obj_init,
    sigmoid="long_tail",
)
button_pressed = reward_utils.tolerance(
    obj_to_target,
    bounds=(0, 0.005),
    margin=self._obj_to_target_init,
    sigmoid="long_tail",
)

reward = 2 * reward_utils.hamacher_product(tcp_closed, near_button)
if tcp_to_obj <= 0.05:
    reward += 8 * button_pressed

return (reward, tcp_to_obj, obs[3], obj_to_target, near_button, button_pressed)

```

## Example LLM-based Evaluation Functions for the Button Press Task

```

#####Evaluation Function Sample 1#####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    # Weights for different components of the evaluation
    distance_weight = 0.5
    y_alignment_weight = 0.3
    trend_weight = 0.1
    smoothness_weight = 0.1

    distance_score = 0.0
    y_alignment_score = 0.0
    trend_score = 0.0
    smoothness_score = 0.0

    previous_ee_position = None
    previous_joint_positions = None
    previous_joint_velocities = None
    if len(trajectory.states) == 25:
        a = trajectory.actions[0] == None
        if type(a) != bool:
            for i, (state, action, observation) in enumerate(zip(trajectory.states, trajectory.actions, trajectory.observations)):
                robot = state.robot
                button = state.button

                # Immediate evaluation: Distance to button
                distance = np.linalg.norm(robot.ee_position - button.position)
                distance_score += 1.0 / (1.0 + distance) # Inverse to give higher scores for smaller distances

                # Immediate evaluation: Y-coordinate alignment
                y_alignment = 1.0 - abs(robot.ee_position[1] - button.position[1])
                y_alignment_score += y_alignment

                # Holistic evaluation: Trend analysis
                if previous_ee_position is not None:
                    if (robot.ee_position[1] - previous_ee_position[1]) * (button.position[1] - previous_ee_position[1]) > 0:
                        trend_score += 1.0

                # Holistic evaluation: Movement smoothness
                if previous_joint_positions is not None and previous_joint_velocities is not None:

```

```

        joint_position_diff = np.linalg.norm(robot.joint_positions -
                                              previous_joint_positions)
        joint_velocity_diff = np.linalg.norm(robot.joint_velocities -
                                              previous_joint_velocities)
        smoothness_score += 1.0 / (1.0 + joint_position_diff + joint_velocity_diff
                                    )

    previous_ee_position = robot.ee_position
    previous_joint_positions = robot.joint_positions
    previous_joint_velocities = robot.joint_velocities

    # Normalize scores
    num_steps = len(trajectory)
    if num_steps > 0:
        distance_score /= num_steps
        y_alignment_score /= num_steps
        trend_score /= num_steps
        smoothness_score /= num_steps

    # Final score calculation with weights
    final_score = (distance_weight * distance_score +
                   y_alignment_weight * y_alignment_score +
                   trend_weight * trend_score +
                   smoothness_weight * smoothness_score)
    else:
        final_score = 0
    else:
        final_score = 0
    return final_score

####Evaluation Function Sample 2####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    immediate_scores = []
    holistic_scores = []

    # Immediate evaluation weights
    weight_tcp_to_obj = 0.4
    weight_button_pressed = 0.6

    # Holistic evaluation weights
    weight_success_rate = 0.5
    weight_efficiency = 0.3
    weight_stability = 0.2

    # Collect immediate scores
    if len(trajectory.states) == 25 :
        a = trajectory.actions[0] == None
        if type(a) != bool:
            for state, action, observation in zip(trajectory.states, trajectory.actions,
                                                   trajectory.observations):
                tcp_to_obj = np.linalg.norm(observation[4:7] - observation[:3]) # Distance
                from hand to button
                button_pressed = state['obj_pos'][1] <= 0.78 # Check if the button is pressed

                immediate_score = (weight_tcp_to_obj * (1 - tcp_to_obj)) + (
                    weight_button_pressed * button_pressed)
                immediate_scores.append(immediate_score)

    # Calculate holistic scores
    total_steps = len(trajectory)
    successful_steps = sum(1 for state in trajectory.states if state['obj_pos'][1] <=
                           0.78)
    success_rate = successful_steps / total_steps if total_steps > 0 else 0

    # Efficiency: Inverse of the number of steps taken to complete the task
    efficiency = 1 / total_steps if total_steps > 0 else 0

    # Stability: Variability in the hand's position (lower variability means higher
    # stability)
    hand_positions = np.array([state['hand_pos'] for state in trajectory.states])
    stability = 1 / np.std(hand_positions) if np.std(hand_positions) > 0 else 0

```

```

        holistic_score = (weight_success_rate * success_rate) + (weight_efficiency *
            efficiency) + (weight_stability * stability)

        # Combine immediate and holistic scores
        final_score = np.mean(immediate_scores) + holistic_score
    else:
        final_score = 0
    else:
        final_score = 0
    return final_score

#####Evaluation Function Sample 3#####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    # Initializing variables for evaluation
    total_steps = len(trajectory)
    if total_steps == 0:
        return 0.0

    proximity_weight = 0.3
    force_weight = 0.2
    smoothness_weight = 0.2
    completion_weight = 0.3

    total_proximity_score = 0.0
    total_force_score = 0.0
    total_smoothness_score = 0.0
    task_completed = False

    for i in range(total_steps):
        state = trajectory.states[i]
        action = trajectory.actions[i]
        observation = trajectory.observations[i]

        # Extract relevant data
        ee_position = state.robot.ee_position
        button_position = state.button.position
        joint_velocities = state.robot.joint_velocities
        goal_position = state.button.position

        # Calculate proximity score (closer is better)
        distance_to_button = np.linalg.norm(ee_position - button_position)
        proximity_score = max(0, 1 - distance_to_button) # Closer to button gives higher
            score
        total_proximity_score += proximity_score

        # Calculate force score (assume we can check the force applied in some way, here we
            use placeholder)
        force_applied = np.linalg.norm(action) # Assuming action contains force information
        ideal_force = 1.0 # Placeholder for ideal force value
        force_score = max(0, 1 - abs(force_applied - ideal_force))
        total_force_score += force_score

        # Calculate smoothness score (lower joint velocities indicate smoother movement)
        smoothness_score = max(0, 1 - np.mean(np.abs(joint_velocities)))
        total_smoothness_score += smoothness_score

        # Check if the task is completed
        if np.array_equal(ee_position, goal_position):
            task_completed = True

    # Normalize scores
    avg_proximity_score = total_proximity_score / total_steps
    avg_force_score = total_force_score / total_steps
    avg_smoothness_score = total_smoothness_score / total_steps
    completion_score = 1.0 if task_completed else 0.0

    # Calculate overall score
    final_score = (proximity_weight * avg_proximity_score +
                    force_weight * avg_force_score +
                    smoothness_weight * avg_smoothness_score +

```

```

        completion_weight * completion_score)

    return final_score

#####Evaluation Function Sample 4#####

def evaluate_trajectory(trajectory: Trajectory) -> float:
    """
    Evaluate the trajectory of the robot based on immediate and holistic criteria.

    Parameters:
    trajectory (Trajectory): The trajectory object containing states, actions, and
        observations.

    Returns:
    float: The final score representing the quality of the trajectory.
    """
    # Initialize scores
    proximity_score = 0.0
    action_effectiveness_score = 0.0
    button_press_success_score = 0.0
    consistency_score = 0.0
    efficiency_score = 0.0
    smoothness_score = 0.0

    # Immediate evaluation parameters
    proximity_threshold = 0.05 # Threshold for proximity to the button
    button_press_threshold = 0.02 # Threshold for button press success

    # Holistic evaluation parameters
    max_steps = 25 # Maximum number of steps in the trajectory
    smoothness_weight = 0.1 # Weight for smoothness in the overall score

    # Iterate over the trajectory
    if len(trajectory.states) == 25 :
        a = trajectory.actions[0] == None
        if type(a) != bool:
            for i in range(len(trajectory)):
                state = trajectory.states[i]
                action = trajectory.actions[i]
                observation = trajectory.observations[i]

                # Immediate evaluation
                tcp_to_obj = np.linalg.norm(state['hand_pos'] - state['obj_pos'])
                obj_to_target = abs(state['obj_pos'][1] - 0.78) # Goal y-coordinate is 0.78

                # Proximity to the button
                if tcp_to_obj <= proximity_threshold:
                    proximity_score += 1.0

                # Action effectiveness
                if i > 0:
                    prev_state = trajectory.states[i-1]
                    prev_tcp_to_obj = np.linalg.norm(prev_state['hand_pos'] - prev_state['obj_pos'])
                    if tcp_to_obj < prev_tcp_to_obj:
                        action_effectiveness_score += 1.0

                # Button press success
                if obj_to_target <= button_press_threshold:
                    button_press_success_score += 1.0

            # Holistic evaluation
            total_steps = len(trajectory)

            # Consistency: How often the robot moves closer to the goal
            for i in range(1, total_steps):
                prev_state = trajectory.states[i-1]
                curr_state = trajectory.states[i]
                prev_tcp_to_obj = np.linalg.norm(prev_state['hand_pos'] - prev_state['obj_pos'])


```

```

curr_tcp_to_obj = np.linalg.norm(curr_state['hand_pos'] - curr_state['obj_pos'])
)
if curr_tcp_to_obj < prev_tcp_to_obj:
    consistency_score += 1.0

# Efficiency: Reward quicker task completion
efficiency_score = max(0, max_steps - total_steps)

# Smoothness: Penalize erratic movements
for i in range(2, total_steps):
    prev_action = trajectory.actions[i-1]
    curr_action = trajectory.actions[i]
    action_diff = np.linalg.norm(curr_action - prev_action)
    smoothness_score -= smoothness_weight * action_diff

# Normalize scores
total_possible_steps = max_steps - 1
proximity_score /= total_possible_steps
action_effectiveness_score /= total_possible_steps
button_press_success_score /= total_possible_steps
consistency_score /= total_possible_steps

# Combine scores into a final score
final_score = (
    proximity_score * 0.3 +
    action_effectiveness_score * 0.2 +
    button_press_success_score * 0.3 +
    consistency_score * 0.1 +
    efficiency_score * 0.05 +
    smoothness_score * 0.05
)
else:
    final_score = 0
else:
    final_score = 0
return final_score

```

We can observe that compared to the expert-designed reward functions, the LLM-based evaluation functions cover more than just success-related criteria, providing a more nuanced evaluation pattern. Also, as required by the prompts, the LLM-based evaluation functions cover immediate state-action pairs as well as holistic evaluations.

For example, on the Walker task, the expert reward function is primarily focused on immediate task success, measured through Upright Posture, which rewards the walker for keeping its torso upright, and Torso Height, which ensures the torso height is within a certain range. On the other hand, the LLM-based evaluation function integrates these success-related criteria but also extends the evaluation to cover additional aspects, such as Energy Efficiency, measured by penalizing large action magnitudes to promote energy-efficient behavior, and Stability Over Time by evaluating changes in torso height and uprightness, ensuring stability throughout the trajectory. By incorporating broader criteria—both immediate and holistic—the LLM-based evaluation functions provide a more comprehensive and nuanced assessment of the robot trajectories. This ensures the walker not only completes the tasks successfully but also does so efficiently, stably, and consistently over time, leading to potentially more robust and effective reinforcement learning outcomes.

More importantly, we observe that the evaluation functions generated from the same gpt-4o agents, exhibit diversity. This variation manifests in several ways, such as differing task-related criteria, assorted definitions for the same criteria, and varying priorities assigned to these criteria (e.g., different weighting schemes). Our PrefCLM capitalizes on this diversity, leveraging the unique understanding that each LLM agent brings to the task and leading to a richer and more comprehensive evaluation process.

## APPENDIX V THE FEEDING TASK IN THE USER STUDY

We selected the Feeding task from the Assistive Gym: A Physics Simulation Framework for Assistive Robotics [29]. In this task, a robot arm is tasked with delivering a spoon holding food, represented as small spheres, to the mouth of a human seated in a chair without spilling.

### A. Pre-trained Robot Policy

To pre-train a robot policy, we utilized the ground-truth reward functions provided by the benchmark, which consist of several costs and penalties to differentiate:

- $C_d(s)$ : cost for long distance from the robot's end effector to the target assistance location (e.g., human mouth).
- $C_e(s)$ : reward for successfully feeding food to the human mouth.
- $C_v(s)$ : cost for high robot end effector velocities.
- $C_f(s)$ : cost for applying force away from the target assistance location.
- $C_{hf}(s)$ : cost for applying high forces near the target.
- $C_{fd}(s)$ : cost for spilling food on the human.
- $C_{fdv}(s)$ : cost for food entering the mouth at high velocities.

We selected the default weights for these criteria as in [29]. We trained the robot policy using Soft Actor-Critic (SAC) for a total of  $1.6 \times 10^7$  time steps, approximately 8 hours. SAC is also the RL training basis of PEBBLE [6], the PbRL backbone algorithm for our PrefCLM, ensuring smooth fine-tuning with PrefCLM. After pre-training, the robot policy is capable of basic functionality, i.e., successfully bringing the spoon close to a certain distance from the user's mouth.

Furthermore, the Assistive Gym allows for adjusting the human shape, location of the chair, mounting of the robotic arm, and other physical parameters, providing a good opportunity to mimic realistic settings during pre-training.

### B. Fine-tuning Process

During the user study, we aimed to fine-tune the pre-trained robot policy using PrefCLM (few-shot, n=10) by incorporating user interactive feedback and compare the resulting satisfaction and personalization against the baseline PrefEVO and pre-trained robot policy.

Each participant first expressed their initial expectations for the Feeding Task in natural language, e.g., "I want the robot to move carefully and slowly when feeding me." PrefCLM then generated initial evaluation functions based on these expectations. Using the crowdsourced evaluation functions, we fine-tuned the pre-trained policy.

For each model (PrefCLM and baseline PrefEVO), we periodically (every  $4 \times 10^6$  environment steps, approximately 2 hours) rolled out the learned robot policy to the physical Kinova Jaco robotic arm, for a total of three times. Each time, participants provided interactive feedback, which was utilized to refine the evaluation functions.

Specifically, we conducted the following steps:

- Fine-tuned the pre-trained policy for  $4 \times 10^6$  environment steps using the initial evaluation functions generated based on user expectations.
- Participants interacted with the first fine-tuned policy and provided the first interactive feedback.
- PrefCLM adapted the evaluation functions based on this feedback and fine-tuned the policy again for  $4 \times 10^6$  environment steps.
- Repeated the interaction and feedback process with the second fine-tuned policy.
- PrefCLM adapted the evaluation functions once more and fine-tuned the policy again for  $4 \times 10^6$  environment steps.
- Repeated the interaction and feedback process with the third fine-tuned policy.
- PrefCLM adapted the evaluation functions once more and conducted a final fine-tuning for  $4 \times 10^6$  environment steps.

It is worth noting that PEBBLE, the PbRL backbone algorithm for our PrefCLM, is an off-policy PbRL algorithm. This means that when the evaluation function is adapted and a new reward model is learned in the PbRL setting, PEBBLE can re-label all state-action pairs from the behavior of previous robot policies and reward models in the replay buffer. This ensures efficient use of previous experiences and accelerate the learning process.

For participants, they could choose to leave or stay during the fine-tuning process. If they decided to leave, we stored the fine-tuned robot policy after current round of training, and resumed with the next round of interaction, interactive feedback, evaluation function adaptations, and fine-tuning upon their return.

The final fine-tuned robot policy by PrefCLM is compared to the final fine-tuned one by PrefEVO and the pre-trained policy, as the final interaction policy. Each participant interacted with each policy three times in a randomized sequence and was not informed about which policy was active to prevent any bias in their responses. Following each interaction, participants were asked to rate the robot behaviors of the three policies in terms of satisfaction, using a Likert scale ranging from 1 (strongly disagree) to 5 (strongly agree). They were also asked to rate the level of personalization resulting from PrefCLM and PrefEVO using the same scale.