

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(trajecory: 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: RigidBody # 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 RigidBody:
    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: RigidBody # 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 Evaluation Functions for the Walker Task

```
###Evaluation 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
    )

    # 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(trajecory) * _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_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.
- $C_{h,f}(s)$: 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 1.6×10^7 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 4×10^6 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 4×10^6 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 4×10^6 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 4×10^6 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 4×10^6 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.