TABLE III
IMPLEMENTATION DETAILS

Hyperparameter	Value
Discount Factor γ	0.99 (0.95 for PushChair)
Adam β_1	0.9
Adam β_2	0.99
Policy learning rate λ_π	0.0003
Q-function learning rate λ_Q	0.0003
Mini-batch size	256 (1024 for PushChair)
Replay buffer size	1M
Num. of retraining steps (Ant, HalfCheetah, Hopper, Walker2D)	1M
Num. of retraining steps (Humanoid)	4M
Num. of retraining steps (PushChair)	2M
Num. of evaluation episodes	5
Evaluation interval	5000
Dropout rate	0.1
Temperature of LVLMs (LaMOuR)	0.0

TABLE IV
ENVIRONMENTAL DETAILS

Environment	Termination Condition	OOD State	Scaling Coef. (λ)
Humanoid	not ($1.0 < h_{torso} < 2.0$)	$h_{torso} = 0.105$	0.05
PushChair	$\theta_{chair} > \pi/5$	$\theta_{chair} = \pi/2$	0.01

SUPPLEMENTARY MATERIAL

A. Implementation Details

In this subsection, we describe the implementation details of our method and baselines. For all environments, the policies and Q-functions of all methods are implemented as two-layer multi-layer perceptrons (MLPs) with 256 hidden units per layer. Each layer employs ReLU activation. All methods are optimized using the Adam optimizer [38]. The hyperparameters used for the experiments are listed in Table IV. We employed the automated entropy adjustment proposed in [39] for the entropy coefficient α. Note that the same set of hyperparameters is applied across all methods.

B. Environmental Details

1) DeepMind Control Suite Humanoid Environment: The original task of the Humanoid environment is to control a 3D bipedal robot, to walk forward as quickly as possible without falling over. During learning the original task, the episode is terminated when the agent falls over. This termination condition is implemented by checking whether the height of the torso, h_{torso}, is within a healthy range.

In our experiment, we retrain the policy in a state where the agent is lying on the floor. This state is an OOD state that the agent has never encountered while learning the original task, as the state lies outside the termination condition. The expected recovery behavior of the Humanoid in this OOD state is to stand up and return to an upright position.

2) ManiSkill2 PushChair Environment: The original task of the PushChair environment is to control the dual-arm Panda robot to push the chair to the goal position. We made a slight modification to the environment, terminating the episode when the chair begins to fall over. This termination condition is implemented by checking whether the tilt angle of the chair, θ_{chair}, exceeds a certain threshold.

Accordingly, we modified the reward function of the original environment. The original reward, designed to be negative to encourage faster success, leads to agent suicide when the agent can terminate the episode by tipping the chair over, resulting in a higher cumulative reward compared to completing the task properly. Therefore, we modified the stage reward to be positive and introduced a large penalty to discourage the agent from tipping the chair over. However, making the stage reward positive no longer incentivizes fast success. Since reward design in the original environment is beyond the scope of this research, we leave it as future work.

Moreover, we modified the state information provided to the generated reward code c_{reward} because the original state from the environment lacks sufficient information for accurate reward calculation. For instance, The original state only includes the chair's position, orientation, and velocity as its state information. However, to evaluate whether the dual arm successfully grabs the chair, it is more relevant to use the distance between the dual arm and the nearest point on the chair, rather than the distance to the chair's center. To address this, we augment the state only when it is passed to the generated code c_{reward}, while the agent's policy continues to use the original state.

In our experiment, we retrain the policy in a state where the chair has already fallen to the floor. This state is an OOD state that the agent has never encountered while learning the original task, as the state lies outside the termination condition. The expected recovery behavior of the agent is to use its dual arms to grab the chair and lift it back to an upright position. The few-shot example used for *Code Generation* is illustrated in Fig. 12.

C. Environment Description

As outlined in the original paper, we provide environment descriptions containing state and action information to the Code Generator to generate environment-specific recovery reward code. An example of the environment description for the MuJoCo Ant environment used in our system is shown in Fig. 13.

D. Full Prompts

We include all the prompts used for our system in Fig. 14–17.

- ***OOD Description***: Fig. 14
- ***Behavior Reasoning***: Fig. 15
- ***Code Generation***: Fig. 16, 17

````
Implement staged rewards for the recovery behavior using a nested if-else structure.

Here is an example of how to generate code for staged reward:

Recovery behavior: Lift a cube from a table to a specified height (0.5 meters).

Corresponding reward code:
```python
import numpy as np
from typing import Optional

def calculate_reward(state: np.ndarray, action: np.ndarray) -> float:
    # Initialize base reward
    reward = 0.0

    # Extract state components
    dist_to_cube = state[0]  # Distance to cube
    gripper_force = state[1]  # Force applied by gripper
    cube_height = state[2]  # Cube's current height
    cube_tilt = state[3]  # Cube's tilt angle
    cube_velocity = state[4]  # Cube's velocity

    target_height = 0.5  # Desired cube height

    # Logarithmic distance reward (encourages closer approach)
    log_dist_to_cube = np.log(dist_to_cube + 1e-5)
    reward += -dist_to_cube - np.clip(log_dist_to_cube, -10, 0)

    # Height-based reward (minimizing deviation from target)
    height_diff = np.abs(cube_height - target_height)
    reward += -height_diff * 0.2

    # Action penalty (encourages smooth control)
    action_norm = np.linalg.norm(action)
    reward -= action_norm * 1e-6

    # Stage-based reward
    stage_reward = -10

    # Stage 1: Approach cube
    if dist_to_cube < 0.1:
        stage_reward += 2  # Reward for reaching the cube

        # Stage 2: Establish grip
        if gripper_force > 0.1:
            stage_reward += 2
            reward += 2.0 * gripper_force  # Extra reward for grip strength

            # Stage 3: Lift cube
            if height_diff < 0.1:
                stage_reward += 2
                reward -= cube_tilt * 0.2  # Penalize excessive tilting

    # Add stage-based reward to final reward
    reward += stage_reward

    return reward
```
````

Fig. 12. Few-shot example utilized in the PushChair environment.

```
# MuJoCo Ant Environment Documentation

## Environment Overview
The ant is a 3D quadruped robot consisting of a torso (free rotational body) with four
legs attached to it, where each leg has two body parts.
The ant is capable of performing various locomotion tasks by applying torque to the eight
hinges connecting the two body parts of each leg and the torso (nine body parts and eight
hinges).

## Technical Specifications

### State Space
The environment state is represented as a 1-dimensional NumPy array of shape `(111,)`,
containing comprehensive information about the ant's current configuration:

#### 1. Position and Orientation
- `state[0:5]`: Core position (state[0]) and orientation (state[1:5]: quaternion) values
  - `state[0]`: z-coordinate of the torso (center of mass) in meters (0.55 when spawned).
  - `state[1]`: w-orientation of the torso in radians.
  - `state[2]`: x-orientation of the torso in radians.
  - `state[3]`: y-orientation of the torso in radians.
  - `state[4]`: z-orientation of the torso in radians.

#### 2. Joint Angles
- `state[5:13]`: Joint angle information in radians
  - `state[5]`: angle between torso and first link on front left in radians.
  - `state[6]`: angle between the two links on the front left in radians.
  - `state[7]`: angle between torso and first link on front right in radians.
  - `state[8]`: angle between the two links on the front right in radians.
  - `state[9]`: angle between torso and first link on back left in radians.
  - `state[10]`: angle between the two links on the back left in radians.
  - `state[11]`: angle between torso and first link on back right in radians.
  - `state[12]`: angle between the two links on the back right in radians.

### Action Space
The action space consists of a 1-dimensional NumPy array of shape `(8,)`, controlling the
torques applied to each of the ant's actuated joints.

**Range**: All actions are bounded between [-1, 1]

**Control mapping**:
- `action[0]`: Torque applied on the rotor between the torso and back right hip.
- `action[1]`: Torque applied on the rotor between the back right two links.
- `action[2]`: Torque applied on the rotor between the torso and front left hip.
- `action[3]`: Torque applied on the rotor between the front left two links.
- `action[4]`: Torque applied on the rotor between the torso and front right hip.
- `action[5]`: Torque applied on the rotor between the front right two links.
- `action[6]`: Torque applied on the rotor between the torso and back left hip.
- `action[7]`: Torque applied on the rotor between the back left two links.
```

Fig. 13. Environment description for the MuJoCo Ant environment.

```
The input image illustrates an out-of-distribution scenario involving a reinforcement
learning agent trained in {env_name}.

Your task is to analyze and describe the robot's (agent's) state in the image, focusing
specifically on its position, orientation, actions, and any observable anomalies or
deviations from expected behavior.

If other objects are present in the image, provide concise descriptions of their states,
either in relation to the robot or independently. Do not mention the absence of
additional objects if none are present.

Disregard the background completely.

Provide your response as concise, standalone sentences: one describing the robot in
{env_name} and one for each additional object (if present).
```

Fig. 14. Prompt for the *OOD Description*.

```
Identify the specific physical behavior that an agent should execute to recover from an
out-of-distribution (OOD) state and return to a state where it can effectively perform
its original task.

The original task and the agent's OOD state are defined as follows:
- Original Task Description: {original_task}
- OOD State Description in {env_name}: {ood_description}

Based on the descriptions of the OOD state and the original task, determine the single
recovery behavior by following these steps:
1. Identify the state in which the agent can successfully perform the original task.
2. Determine the single specific behavior required to transition the agent from the OOD
state to the state where it can successfully perform the original task.

Avoid including unnecessary details that do not directly contribute to the recovery
process.

Follow these guidelines when formulating the recovery behavior:
1. Do not reference the original task directly in the recovery behavior.
2. Provide a single, clear, and concise sentence that describes the single recovery
behavior the agent should execute.

The output format should be: [A single, concise sentence explaining the single recovery
behavior]
```

Fig. 15. Prompt for the *Behavior Reasoning*.

````
Generate Python code for two functions:
1. is_recovered(state): An evaluation function that checks whether the agent is in a
state where it can successfully perform the original task.
2. calculate_reward(state, action): A reward function that evaluates the recovery
behavior of the agent in {env_name} based on the given state and action.
Ensure the code is clear, concise, and suitable for integration into the RL agent's
framework.

Recovery behavior refers to the actions taken by an agent to transition from an out-of-
distribution (OOD) state back to a state where it can effectively perform its original
task.

The agent's state, action information, current OOD state, and recovery behavior are
described as follows:
1. Description of Agent's State and Action: {environment_description}
2. Description of Recovery Behavior: {recovery_behavior}

The format of the evaluation function (is_recovered(state)) is defined as follows:

```
def is_recovered(state: np.ndarray) -> int:
    """
    Verify if agent has recovered to state where it can perform the original task.

    Args:
        state (np.ndarray): Environment state vector

    Returns:
        int: 1 if recovered, 0 if still recovering
    """
```

Follow these guidelines when writing the evaluation function:
1. Define clear criteria for evaluating the success of the specified recovery behavior.
2. **Guarantee that the criteria include only the most essential state components
necessary to evaluate the specified recovery behavior, avoiding any extraneous velocity-
based metrics or non-essential variables**.
3. Guarantee that the criteria are not too strict to avoid hindering the recovery process.
4. Return 1 if the state satisfies the criteria, and 0 otherwise.
5. If quaternions are used in the function, convert them to Euclidean angles and used
them instead of quaternions for better interpretability.
````

Fig. 16. Prompt for the *Code Generation* (1/2).

````
The format of the reward function (calculate_reward(state, action)) is defined as follows:

```
def calculate_reward(state: np.ndarray, action: np.ndarray) -> float:
    """
    Compute the reward for recovery behavior.

    Args:
        state (np.ndarray): Agent's state vector
        action (np.ndarray): Agent's action vector

    Returns:
        float: Reward value
    """
```

Follow these guidelines when designing the reward function:
1. Focus on the actions specified in the recovery behavior.
2. Use the same criteria defined in the evaluation function (is_recovered(state)) to
determine whether the recovery behavior has been achieved.
3. **Gradually increase the reward as the state approaches the defined criteria and
decrease it as the state moves further away. Carefully consider the direction of
approach.**
4. **Guarantee the code strictly aligns with the provided description of the recovery
behavior; avoid implementing unrelated functionality.**
5. Guarantee that the recovery reward is always a negative value.
6. Penalize large actions to promote efficiency and minimize unnecessary effort during
recovery. Use a coefficient to scale the penalty appropriately, accounting for the
dimensionality of the action space.
7. If quaternions are used in the function, convert them to Euclidean angles and used
them instead of quaternions for better interpretability.

{example}

Both functions should adhere to the following coding guidelines:
- Implement using pure Python and NumPy.
- Include type hints for all functions.
- Provide clear and comprehensive docstrings with Args and Returns sections.
- Follow PEP 8 coding standards.
- Prefer vectorized operations for efficiency.
- Handle edge cases, such as NaN or infinity in inputs.
- Ensure functions have no side effects and do not modify inputs.

The response should follow these guidelines:
- Provide only Python code without any additional explanations.
- Do not include any markdown symbols, such as ``` or ```python.
- Implement any necessary helper functions for calculations.
- Use clear and descriptive variable names for readability.
- Add inline comments to explain any complex or non-obvious logic.
````

Fig. 17. Prompt for the *Code Generation* (2/2).