Supplementary Material for "LLMs as MDP Designers: A Dependency-Aware Agentic Framework for Automated Robot Policy Generation"

This document provides raw supplementary artifacts used in the ARPG experiments, including full task descriptions, intermediate ARPG traces, representative agent prompt templates, and simulator-specific documentation excerpts. These materials are intended to support reproducibility and auditability rather than to serve as a concise narrative appendix.

A Natural Language Descriptions of Main Benchmark Tasks

This section provides the full descriptions of the main benchmark tasks.

CartPole

A robot with an attached pole is placed on a cart that can move left and right. There's a joint between the cart and the pole, but it's a structure that rotates freely. We can apply force only to the cart. The cart is initialized within ±0.5 meters from the center, and the pole angle is initialized within ±0.05 radians. Both the cart and the pole start with zero velocity. The cart can move up to 3 meters to the left or right, and the pole is considered fallen if it tilts more than 0.5 radians in either direction The goal is to determine how to move the cart so that the pole remains upright for as long as possible.

Ant

The ant robot operates in a physics-based simulation with a fixed integration time step of 1/120 seconds. Control commands are applied through joint-level effort inputs at a lower effective frequency due to action decimation, resulting in a control interval of approximately 0.0167 seconds. Each episode spans a finite horizon of 16 seconds and terminates either when the time limit is reached or when the robot's base height falls below a minimum threshold (0.31 m), which indicates a loss of stable contact with the ground.

At each time step, the ant selects continuous control commands that actuate its leg joints, causing the articulated body to evolve under gravity, contact forces, and actuation limits. The task is defined relative to a fixed global reference direction: forward motion corresponds to movement along the positive X-axis, which can equivalently be interpreted as making progress toward a distant reference point placed far ahead in the scene. Consistent advancement in this direction reflects effective coordination among the legs and successful locomotion behavior.

While moving forward is essential, it is not sufficient on its own. The ant is implicitly encouraged to maintain a stable body orientation and continuous ground contact, as excessive tilting, rolling, or erratic leg motions tend to disrupt forward progress and lead to premature termination. In practice, effective policies discover gaits that balance propulsion with stability, keeping the body reasonably level while generating smooth, rhythmic leg movements.

The robot interacts with a flat ground surface configured with unit static and dynamic friction and zero restitution, providing consistent contact conditions across episodes. At the beginning of each episode, the initial state is randomized: the base pose and velocity are perturbed within bounded ranges, and joint positions and velocities are offset from their nominal configuration. As a result, the ant must repeatedly recover coordinated locomotion from a variety of nearby initial configurations rather than relying on a single fixed starting posture.

Actuation is applied uniformly across all joints with a moderate scaling factor, requiring the policy to learn how to distribute effort efficiently across the legs. Overly aggressive or abrupt

commands may lead to unnecessary energy expenditure or unstable motions, whereas smoother and more coordinated actuation tends to support sustained forward travel.
Overall, the objective is to learn a control policy that enables the ant to move forward steadily along the positive X-axis, maintain sufficient body stability to avoid collapse, and coordinate its legs in a way that supports continuous and efficient locomotion over the duration of each episode.

Humanoid

The humanoid operates in a physics-based simulation with a fixed integration time step of 1/120 seconds. Control commands are applied at a lower frequency through action decimation, resulting in an effective control interval of approximately 0.0167 seconds. Each episode runs for a finite horizon of 16 seconds and terminates either upon reaching the time limit or when the humanoid's base height drops below a minimum threshold (0.6 m), which corresponds to a loss of balance in the simulated environment.
Within each episode, the humanoid repeatedly selects joint-level control commands that determine how its articulated body evolves under gravity, contact forces, and actuation limits. The task is defined with respect to a fixed global reference direction: forward motion corresponds to movement along the positive X-axis, which can equivalently be interpreted as progression toward a distant reference location placed far ahead in the scene. In addition to advancing forward, the humanoid is expected to maintain an upright posture and a forward-facing orientation, avoiding excessive leaning, twisting, or lateral deviation while walking. Sustained advancement along the positive X-axis while remaining upright and aligned with the forward direction constitutes successful locomotion behavior.
As the simulation unfolds, the humanoid's base position and orientation continuously change as a consequence of its chosen actions. Effective policies therefore require not only generating forward motion, but also actively stabilizing the torso and coordinating limb movements to keep the body facing forward and vertically aligned under gravity.
The robot interacts with a flat ground surface configured with unit static and dynamic friction and zero restitution, defining consistent contact conditions throughout training. Initial conditions are randomized at the beginning of each episode: the root pose and velocity are perturbed within bounded ranges, and joint positions and velocities are offset from their nominal configuration, introducing variability into the starting state distribution. As a result, the humanoid encounters a range of nearby initial configurations and must repeatedly recover stable, upright walking motion under slightly different conditions.
Actuation properties are non-uniform across the humanoid's joints. Joints associated with the torso and pelvis operate with higher effective transmission ratios (on the order of 60–70), reflecting their role in supporting body weight and maintaining balance. Major leg joints exhibit a broader range of actuation strength, with certain thigh components configured at higher ratios (around 130) and others at moderate levels (around 45), while shin joints are assigned intermediate values (around 90). In contrast, foot joints operate under comparatively lower actuation scaling (around 20–25), allowing more compliant interaction with the ground. This heterogeneous actuation profile reflects the differing mechanical demands placed on various joints during forward, upright locomotion and shapes how balance and stability emerge over time.

Underspecified Humanoid

So, yeah… this is just a humanoid in simulation and we want it to walk forward without falling over. When I say "forward," I don't mean it in a super strict, mathematical sense. Most of the time it's along the world +X direction, but depending on how the scene is set up, it could also just mean "the direction the robot is generally moving or facing." So I wouldn't lock that down too hard.
An episode runs for some fixed amount of time — usually something like ten-ish seconds, maybe a bit more — but honestly the exact duration isn't that important here and it has changed across different runs. Episodes also end when the robot is basically falling, but that's not always decided by one clean number. Sometimes it's about the base getting too low, other times it's more of

a general “this posture is clearly unstable now” kind of check. I don’t think there’s a single termination rule that applies everywhere.
What we care about is that the motion looks like reasonable walking. But what “reasonable” means isn’t fully pinned down yet. In some experiments we cared more about moving faster, in others we mostly cared about not wobbling or collapsing, and sometimes energy use or how smooth the motion looks mattered more. There’s definitely a trade-off there, and we haven’t fixed the priorities in advance. The idea was to adjust that later once we saw how training behaved.
The policy sends joint-level commands, but the exact interpretation of those commands depends on the actuator setup. In some cases it behaves more like sending target joint positions, in other cases it’s closer to effort or drive-style commands. So I wouldn’t assume a single, fixed action type or scale here — that really depends on how the actuators are configured for a given run.
As for observations, the policy gets the robot’s pose and joint-related information, but not necessarily everything you might want. Sometimes velocities are given directly, sometimes they’re inferred indirectly, and things like foot contact or heading information may or may not be present. There could also be some noise or delay in the signals, but the exact details weren’t always consistent across experiments.
The ground is flat, but physical properties like friction or damping weren’t always locked to one exact value. They were usually around the standard settings, but small variations or randomization may have been applied. Initial conditions are randomized too, though exactly which variables are randomized and by how much is something that changed between runs.
Overall, the goal is just to get the humanoid to keep moving forward in a way that doesn’t obviously fall apart. The precise definition of the objective, the termination conditions, and even the exact observation and action structure were intentionally left flexible and refined later during training.

Reach

This is a reach-style end-effector pose tracking task in a manager-based Isaac Lab environment. The simulator runs with a physics timestep of 1/60 s and control decimation = 2 (effective control period ≈ 0.033 s). Each episode lasts exactly 12 seconds and terminates only by timeout.
At each control step, the policy outputs continuous commands for the 7 arm joints (the gripper remains fixed). The task is defined by a time-varying end-effector pose command (position + orientation) expressed in the robot base frame. The target pose is resampled every 4 seconds and held constant between resampling events.
Target positions are sampled within a bounded, reachable workspace above the table, roughly $x \in [0.35, 0.65]$, $y \in [-0.20, 0.20]$, $z \in [0.15, 0.50]$ in the base frame. To avoid degenerate behaviors such as reaching under the table or colliding with the tabletop, the commanded targets are restricted to remain above the tabletop by a safety margin; the target z is always strictly above the table surface height (in the same base frame) by at least a small clearance, and targets are sampled such that they are feasible without requiring self-collision or contact with the table.
The commanded orientation is constrained to match a typical reaching convention: roll is fixed, pitch is kept within a limited range, and yaw is allowed to rotate freely. The goal is to minimize both position and orientation tracking error between the commanded end-effector pose and the actual end-effector pose over time, including immediately after each target change.
In addition to tracking accuracy, smooth and stable motion is preferred: large joint velocities and abrupt changes in control commands are undesirable because they tend to destabilize tracking near the target. At the start of each episode, joint positions are randomized within reasonable bounds (consistent with joint limits) so the policy must generalize across different initial configurations while still tracking targets safely above the table.

Lift

The task is set in a fixed-base manipulation environment built with Isaac Lab (v2.1) on top of Isaac Sim (v4.5). A Franka Emika Panda robotic arm with a parallel gripper is loaded from the standard USD asset and operates on a tabletop scene.
The simulation evolves at a physics integration rate of 100 Hz, with control commands applied every other physics step, resulting in an effective control frequency of 50 Hz. Each episode lasts for approximately five seconds of simulated time and terminates either when this time limit is

reached or when the manipulated object falls below a height of about -0.05 m, which is treated as a loss of control.
At the beginning of each episode, a small rigid object is placed on the table roughly half a meter in front of the robot. Its initial horizontal position is randomized within a limited range, introducing variation in the relative geometry, while its initial vertical position is fixed at the table height.
Throughout an episode, pose commands are sampled within a bounded workspace above the table, with target positions typically lying between about 0.4 and 0.6 meters in front of the robot, within ±0.25 meters laterally, and between roughly 0.25 and 0.5 meters in height. These commands are expressed relative to the robot base and remain constant over the duration of an episode, serving as soft references for guiding motion rather than hard success constraints.
As the arm moves, the distance between the gripper and the object gradually decreases, eventually allowing contact to occur. Meaningful interaction emerges when the gripper approaches the object, establishes control through grasping, and lifts it above approximately 0.04 m. Once lifted, the object can be transported within the workspace while remaining stably coupled to the arm.
The robot itself consists of a seven-degree-of-freedom arm and a two-finger parallel gripper. Successful behavior involves positioning the object between the gripper fingers, closing the gripper to achieve a stable grasp, maintaining that grasp without excessive motion, and lifting the object in a smooth and controlled manner.
At each control step, the agent observes the robot's joint positions and velocities, the object's position relative to the robot, the currently commanded target pose, and the most recent action. These signals are combined into a single observation vector used for decision making.
Because interaction unfolds continuously over time, effective strategies rely on sequences of moderate, coordinated actions rather than abrupt corrections. The learning process therefore favors smooth progress accumulated across successive steps, while discouraging excessive joint motion and rapid changes in control inputs.
The overall goal is to learn robust object manipulation behaviors under these physical, temporal, and geometric constraints using reinforcement learning, without explicitly hard-coding task stages or success conditions.

B ARPG Logbook Traces

The examples presented below are intermediate artifacts obtained from the ARPG logbook, with only minimal editorial modifications applied for publication.

B.1 Cartpole

Parameters

- **CartInitPosAbsMax**: Maximum absolute initial cart displacement from the center (meters) (float).
- **PoleInitAngleAbsMax**: Maximum absolute initial pole angle from upright (radians) (float).
- **CartInitVelocity**: Initial linear velocity of the cart (m/s) (float).
- **PoleInitAngularVelocity**: Initial angular velocity of the pole (rad/s) (float).
- **CartPositionLimitAbsMax**: Maximum absolute allowed cart displacement before termination (meters) (float).
- **FailureAngleAbsMax**: Maximum absolute allowed pole angle before termination (radians) (float).
- **RootInitPos**: Initial world position of the articulation root (x, y, z) in meters (array of size 3, float).
- **SliderToCartInitPos**: Initial position of the prismatic joint `slider_to_cart` (meters) (float).
- **CartToPoleInitPos**: Initial angle of the revolute joint `cart_to_pole` (radians) (float).
- **RigidBodyEnabled**: Whether rigid body simulation is enabled (binary).

- **EnableGyroscopicForces**: Whether gyroscopic forces are enabled (binary).
- **EnabledSelfCollisions**: Whether self-collisions are enabled for the articulation (binary).
- **MaxLinearVelocity**: Maximum linear velocity of rigid bodies (m/s) (float).
- **MaxAngularVelocity**: Maximum angular velocity of rigid bodies (rad/s) (float).
- **MaxDepenetrationVelocity**: Maximum depenetration velocity for contact resolution (m/s) (float).
- **SolverPositionIterationCount**: Number of solver position iterations (int).
- **SolverVelocityIterationCount**: Number of solver velocity iterations (int).
- **SleepThreshold**: Linear/angular velocity threshold below which bodies can go to sleep (float).
- **StabilizationThreshold**: Threshold for articulation stabilization (float).
- **CartEffortLimit**: Effort limit of the cart prismatic joint actuator (N) (float).
- **CartVelocityLimit**: Velocity limit of the cart prismatic joint (m/s) (float).
- **CartStiffness**: Spring stiffness of the cart prismatic actuator (float).
- **CartDamping**: Damping of the cart prismatic actuator (float).
- **PoleEffortLimit**: Effort limit of the pole revolute joint actuator (N·m) (float).
- **PoleVelocityLimit**: Velocity limit of the pole revolute joint (rad/s) (float).
- **PoleStiffness**: Spring stiffness of the pole revolute actuator (float).
- **PoleDamping**: Damping of the pole revolute actuator (float).
- **NumJoints**: Total number of articulated joints in the robot (int).
- **NumActuators**: Total number of actuators defined for the robot (int).

Variables

- **CartPosition**: Cart displacement along the track, corresponding to the prismatic joint `slider_to_cart` position (meters).
- **CartVelocity**: Cart linear velocity along the track, corresponding to the prismatic joint `slider_to_cart` velocity (m/s).
- **PoleAngle**: Pole angle relative to the upright configuration, corresponding to the revolute joint `cart_to_pole` position (radians), where 0 indicates the upright pose.
- **PoleAngularVelocity**: Angular velocity of the pole about the hinge, corresponding to the revolute joint `cart_to_pole` velocity (rad/s).
- **CartForceCommand**: Control variable representing the horizontal force applied by the cart actuator; the value is saturated by `CartEffortLimit`.

Objective

- **Objective**: Maximize the expected cumulative survival time of the system by learning a policy π that keeps the cart and pole within their valid operating limits:

$$\max_{\pi} \quad \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \mathbb{I}\left\{\begin{array}{l} |\mathrm{CartPosition}_t| \leq \mathrm{CartPositionLimitAbsMax} \\ \wedge\, |\mathrm{PoleAngle}_t| < \mathrm{FailureAngleAbsMax} \end{array}\right\}\right].$$

  The objective rewards each timestep in which both the cart position and pole angle remain within their respective safety thresholds.

Constraints

- **Initial State Constraints**:
  - $|\mathrm{CartPosition}(0)| \leq \mathrm{CartInitPosAbsMax}$.

  - $|\text{PoleAngle}(0)| \leq \text{PoleInitAngleAbsMax}$.
  - $\text{CartVelocity}(0) = \text{CartInitVelocity}$.
  - $\text{PoleAngularVelocity}(0) = \text{PoleInitAngularVelocity}$.
- **State Safety Constraints**:
  - $\forall t \in \mathbb{N},\ |\text{CartPosition}(t)| \leq \text{CartPositionLimitAbsMax}$.
  - $\forall t \in \mathbb{N},\ |\text{PoleAngle}(t)| \leq \text{FailureAngleAbsMax}$.
- **Velocity Constraints**:
  - $\forall t \in \mathbb{N},\ |\text{CartVelocity}(t)| \leq \text{CartVelocityLimit}$.
  - $\forall t \in \mathbb{N},\ |\text{PoleAngularVelocity}(t)| \leq \text{PoleVelocityLimit}$.
  - $\forall t \in \mathbb{N},\ |\text{CartVelocity}(t)| \leq \text{MaxLinearVelocity}$.
  - $\forall t \in \mathbb{N},\ |\text{PoleAngularVelocity}(t)| \leq \text{MaxAngularVelocity}$.
- **Actuation Constraint**:
  - $\forall t \in \mathbb{N},\ |\text{CartForceCommand}(t)| \leq \text{CartEffortLimit}$.

State

- **Description**: The minimal Markov state describing the cart–pole system at a given timestep, consisting of the cart position and velocity along the slider, and the pole angle (about upright) and angular velocity at the hinge.
- **Variables**:
  - `CartPosition`
  - `CartVelocity`
  - `PoleAngle`
  - `PoleAngularVelocity`
- **Shape**: $[4]$

Action

- **Description**: A continuous control input specifying the horizontal force applied to the cart actuator.
- **Variables**:
  - `CartForceCommand`
- **Type**: Continuous.
- **Shape**: $[1]$.
- **Bounds**:

$$-\text{CartEffortLimit} \ \leq \ \text{CartForceCommand} \ \leq \ \text{CartEffortLimit}.$$

Reward

- **Description**: Alive bonus that returns a reward of 1 at each timestep while both the cart position and pole angle remain within their allowed limits, and 0 otherwise.
- **Formula**:

$$R_t = \mathbb{I}\left\{ \begin{array}{l} |\text{CartPosition}_t| \leq \text{CartPositionLimitAbsMax} \\ \wedge\, |\text{PoleAngle}_t| < \text{FailureAngleAbsMax} \end{array} \right\}.$$

- **Components**:
  - R_t: Reward at time t.
  - CartPosition_t: Cart position along the track at time t (meters).
  - $\text{CartPositionLimitAbsMax}$: Maximum allowed absolute cart position (meters).
  - PoleAngle_t: Pole angle from upright at time t (radians).
  - $\text{FailureAngleAbsMax}$: Maximum allowed absolute pole angle from upright (radians).

– $\mathbb{I}\{\cdot\}$: Indicator function equal to 1 if the condition holds, 0 otherwise.

B.2 Ant

Parameters

- **PhysicsTimeStep**: Fixed physics integration time step in seconds (e.g., $1/120$ s) (float).
- **ControlInterval**: Effective control interval in seconds due to action decimation (approximately $1/60$ s) (float).
- **ActionDecimation**: Number of physics steps per action application (int).
- **EpisodeHorizonSeconds**: Maximum episode duration in seconds (float).
- **EpisodeHorizonSteps**: Maximum episode duration in simulation steps (int).
- **TerminationMinBaseHeight**: Minimum allowed base height before early termination (meters) (float).
- **GroundStaticFriction**: Static friction coefficient of the ground plane (float).
- **GroundDynamicFriction**: Dynamic friction coefficient of the ground plane (float).
- **GroundRestitution**: Coefficient of restitution of the ground plane (float).
- **GravityDisabled**: Whether gravity is disabled for the robot rigid bodies (binary).
- **MaxDepenetrationVelocity**: Maximum depenetration velocity for contact resolution (float).
- **EnableGyroscopicForces**: Whether gyroscopic forces are enabled (binary).
- **EnabledSelfCollisions**: Whether self-collisions are enabled for the articulation (binary).
- **SolverPositionIterationCount**: Number of solver position iterations for the articulation (int).
- **SolverVelocityIterationCount**: Number of solver velocity iterations for the articulation (int).
- **SleepThreshold**: Linear/angular velocity threshold below which bodies may go to sleep (float).
- **StabilizationThreshold**: Threshold for solver stabilization (float).
- **CopyFromSource**: Whether simulation properties are copied from the USD source (binary).
- **InitBasePos**: Initial base position (x, y, z) in meters (array of size 3, float).
- **InitLegJointsDefaultPos**: Default initial joint position (radians) for all joints matching `.*_leg` (float).
- **InitFrontLeftFootPos**: Initial joint position for `front_left_foot` (radians) (float).
- **InitFrontRightFootPos**: Initial joint position for `front_right_foot` (radians) (float).
- **InitLeftBackFootPos**: Initial joint position for `left_back_foot` (radians) (float).
- **InitRightBackFootPos**: Initial joint position for `right_back_foot` (radians) (float).
- **ActuatorStiffness**: Implicit actuator spring stiffness applied to all joints (float).
- **ActuatorDamping**: Implicit actuator damping applied to all joints (float).

Variables

- **JointEffortCommands**: Agent-selected continuous torque/effort commands applied to the eight actuated joints (`front_left_leg`, `front_left_foot`, `front_right_leg`, `front_right_foot`, `left_back_leg`, `left_back_foot`, `right_back_leg`, `right_back_foot`) at each control interval; commands are held constant between updates due to action decimation (shape $[8]$).
- **JointPositions**: Current joint angles of the eight actuated joints (radians) (shape $[8]$).

- **JointVelocities**: Current joint angular velocities of the eight actuated joints (rad/s) (shape $[8]$).
- **BasePosition**: World-frame position of the free-floating base (x, y, z) in meters (shape $[3]$).
- **BaseOrientationQuat**: World-frame orientation of the base represented as a unit quaternion (w, x, y, z) (shape $[4]$).
- **BaseLinearVelocity**: World-frame linear velocity of the base (m/s) (shape $[3]$).
- **BaseAngularVelocity**: World-frame angular velocity of the base (rad/s) (shape $[3]$).
- **FootContactStates**: Binary indicators denoting ground contact for each foot link (`front_left`, `front_right`, `left_back`, `right_back`) (shape $[4]$).
- **FootContactForces**: World-frame ground reaction forces applied at each foot link; forces are zero when the corresponding foot is not in contact (shape $[4 \times 3]$).
- **BaseHeight**: Current base height above the ground plane, corresponding to the z-component of `BasePosition` (meters).
- **ForwardVelocity**: Instantaneous base linear velocity along the $+X$ axis in the world frame (m/s).
- **ForwardDisplacement**: Cumulative displacement of the base along the $+X$ axis measured from the start of the episode (meters).
- **ElapsedTime**: Elapsed time since the beginning of the current episode (seconds).

Objective

- **Objective**: Learn a locomotion policy π that maximizes the expected cumulative reward over a finite horizon:

$$\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T-1} \mathbb{I}\{h_t \geq h_{\min}\}\left(r_t^{\text{prog}} - \lambda_{\text{tilt}}\, r_t^{\text{tilt}} - \lambda_{\omega}\, r_t^{\omega} - \lambda_u\, r_t^{u} - \lambda_{\Delta}\, r_t^{\Delta} - \lambda_{\perp}\, r_t^{\perp}\right)\right],$$

  where $T = \text{EpisodeHorizonSteps}$.
- **Reward Components**:
  - $r_t^{\text{prog}} = \text{ForwardVelocity}_t \cdot \text{PhysicsTimeStep}$ (forward progress).
  - $r_t^{\text{tilt}} = (\text{BaseOrientationQuat}_{t,x})^2 + (\text{BaseOrientationQuat}_{t,y})^2$ (penalty for body tilt).
  - $r_t^{\omega} = \|\text{BaseAngularVelocity}_t\|_2^2$ (penalty for angular motion).
  - $r_t^{u} = \|\text{JointEffortCommands}_t\|_2^2$ (control effort penalty).
  - $r_t^{\Delta} = \mathbb{I}\{t > 0\}\|\text{JointEffortCommands}_t - \text{JointEffortCommands}_{t-1}\|_2^2$ (action smoothness penalty).
  - $r_t^{\perp} = (\text{BaseLinearVelocity}_{t,y})^2$ (lateral drift penalty).
- **Survival Condition**:
  - $h_t = \text{BaseHeight}_t$ is the base height at time t.
  - $h_{\min} = \text{TerminationMinBaseHeight}$ is the minimum allowed base height.

Constraints

- **Time Discretization and Horizon**:

$$\Delta t = \text{PhysicsTimeStep}, \quad d = \text{ActionDecimation}, \quad T = \text{EpisodeHorizonSteps}.$$

$$\text{ControlInterval} = d\,\Delta t, \qquad T = \frac{\text{EpisodeHorizonSeconds}}{\Delta t}, \qquad d, T \in \mathbb{Z}_{\geq 1}.$$

- **Time Evolution**:

$$t_0 = 0, \qquad t_{k+1} = t_k + \Delta t, \qquad t_k \geq 0.$$

- **Termination Time**:

$$\tau = \inf\left\{k \geq 0 \;\middle|\; t_k \geq T\Delta t \;\vee\; h_k < h_{\min}\right\}.$$

- **State Definitions and Invariants**:

$$h_k = (\text{BasePosition}_k)_z \geq 0, \qquad \|\text{BaseOrientationQuat}_k\|_2 = 1.$$
$$v_k^x = (\text{BaseLinearVelocity}_k)_x.$$

- **Forward Displacement**:

$$s_0 = 0, \qquad s_{k+1} = s_k + v_k^x \, \Delta t.$$

- **Action Holding (Decimation)**:

$$u_k = u_{md} \quad \forall k \in \{md, \ldots, (m+1)d - 1\}, \; m \in \mathbb{Z}_{\geq 0}.$$

- **Joint and Actuation Limits**:

$$u_i^{\min} \leq (u_k)_i \leq u_i^{\max}, \quad q_i^{\min} \leq (q_k)_i \leq q_i^{\max}, \quad \dot{q}_i^{\min} \leq (\dot{q}_k)_i \leq \dot{q}_i^{\max}.$$

- **Contact Constraints**:

$$c_{k,i} \in \{0, 1\}, \qquad c_{k,i} = 0 \Rightarrow f_{k,i} = 0.$$
$$c_{k,i} = 1 \Rightarrow \begin{cases} (f_{k,i})_z \geq 0, \\ \sqrt{(f_{k,i})_x^2 + (f_{k,i})_y^2} \leq \mu_{k,i} (f_{k,i})_z, \end{cases}$$

- **Initial Conditions**:

$$\text{BasePosition}_0 = \text{InitBasePos} + \Delta p, \qquad \|\Delta p\|_\infty \leq \bar{\Delta}_{\text{base}}.$$
$$q_0 = \bar{q} + \delta, \qquad \|\delta\|_\infty \leq \bar{\delta}.$$
$$\|\text{BaseLinearVelocity}_0\|_2 \leq \bar{v}_{\text{base}},$$
$$\|\text{BaseAngularVelocity}_0\|_2 \leq \bar{\omega}_{\text{base}},$$
$$\|\dot{q}_0\|_\infty \leq \bar{v}_q.$$

State

- **Description**: Proprioceptive and base state required to plan stable forward locomotion. The state additionally includes the previous action to preserve the Markov property when using action-smoothing penalties in the reward.
- **Variables**:
  - `JointPositions` [8]
  - `JointVelocities` [8]
  - `BaseOrientationQuat` (w, x, y, z)
  - `BaseLinearVelocity` (v_x, v_y, v_z)
  - `BaseAngularVelocity` $(\omega_x, \omega_y, \omega_z)$
  - `BaseHeight`
  - `ForwardVelocity`
  - `FootContactStates` [front_left, front_right, left_back, right_back]
  - `ElapsedTime`
  - `PrevJointEffortCommands` [8]
- **Shape**: [41]

Action

- **Description**: Continuous joint-level effort commands applied to the eight actuated joints. Actions are issued every `ControlInterval` and held constant between updates due to action decimation.
- **Variables**:

– `JointEffortCommands`:

$$\begin{bmatrix}\text{front_left_leg}, & \text{front_left_foot},\\ \text{front_right_leg}, & \text{front_right_foot},\\ \text{left_back_leg}, & \text{left_back_foot},\\ \text{right_back_leg}, & \text{right_back_foot}\end{bmatrix}$$

- **Type**: Continuous.
- **Shape**: $[8]$.
- **Range**:

$$u_i^{\min} \le \text{action}_i \le u_i^{\max},$$

typically implemented by scaling normalized policy outputs in $[-1, 1]$ to the actuator effort limits.

Reward

- **Description**: Per-step forward progress along $+X$ with penalties for tilt, angular motion, effort magnitude, effort changes (smoothness), and lateral drift. Reward is masked to zero after termination.
- **Reward (compact form)**:

$$r_t = \mathbb{I}\{h_t \ge h_{\min}\}\left(r_t^{\text{prog}} - \lambda_{\text{tilt}}\, r_t^{\text{tilt}} - \lambda_\omega\, r_t^{\omega} - \lambda_u\, r_t^{u} - \lambda_\Delta\, r_t^{\Delta} - \lambda_\perp\, r_t^{\perp}\right).$$

- **Reward Components**:
  - $r_t^{\text{prog}} = v_t^x \cdot \Delta t$ (forward progress).
  - $r_t^{\text{tilt}} = (\text{BaseOrientationQuat}_{t,x})^2 + (\text{BaseOrientationQuat}_{t,y})^2$ (body tilt penalty).
  - $r_t^{\omega} = \|\text{BaseAngularVelocity}_t\|_2^2$ (angular motion penalty).
  - $r_t^{u} = \|\text{JointEffortCommands}_t\|_2^2$ (control effort penalty).
  - $r_t^{\Delta} = \mathbb{I}\{t > 0\}\|\text{JointEffortCommands}_t - \text{JointEffortCommands}_{t-1}\|_2^2$ (action smoothness penalty).
  - $r_t^{\perp} = (\text{BaseLinearVelocity}_{t,y})^2$ (lateral drift penalty).
- **Symbol Definitions**:
  - $\Delta t = \text{PhysicsTimeStep}$.
  - $v_t^x = \text{ForwardVelocity}_t = (\text{BaseLinearVelocity}_t)_x$.
  - $h_t = \text{BaseHeight}_t, \quad h_{\min} = \text{TerminationMinBaseHeight}$.
  - $\lambda_{\text{tilt}}, \lambda_\omega, \lambda_u, \lambda_\Delta, \lambda_\perp > 0$ are scalar weights.
  - $\mathbb{I}\{\cdot\}$ is the indicator function (equals 1 if the condition holds, 0 otherwise).

B.3 Humanoid

Parameters

- **Simulation and Control Timing**:
  - **SimDt**: Physics integration time step (seconds, e.g., $1/120$ s) (float).
  - **SimFrequencyHz**: Physics simulation frequency (Hz) (int).
  - **ControlInterval**: Time between applied control commands due to action decimation (seconds) (float).
  - **ControlFrequencyHz**: Effective control command frequency (Hz) (int).
  - **ActionDecimation**: Number of physics steps per control action (int).
- **Episode Horizon and Termination**:
  - **EpisodeDurationSec**: Maximum episode duration (seconds) (float).
  - **EpisodePhysicsSteps**: Maximum number of physics steps per episode (int).
  - **EpisodeControlSteps**: Maximum number of control steps per episode (int).

  - **TerminationBaseHeightMin**: Minimum humanoid base height before termination (meters) (float).
- **Environment and Ground Properties**:
  - **ForwardAxisVector**: Global forward motion direction unit vector (shape [3], float).
  - **GroundStaticFriction**: Static friction coefficient of the ground (float).
  - **GroundDynamicFriction**: Dynamic friction coefficient of the ground (float).
  - **GroundRestitution**: Coefficient of restitution for the ground (float).
- **Initialization Parameters**:
  - **InitBasePosition**: Initial base position (x, y, z) in meters (shape [3], float).
  - **InitAllJointPosValue**: Initial joint position applied uniformly to all joints (radians) (float).
- **Physics Engine Settings**:
  - **MaxDepenetrationVelocity**: Maximum depenetration velocity for rigid bodies (float).
  - **GyroscopicForcesEnabled**: Whether gyroscopic forces are enabled (binary).
  - **SelfCollisionsEnabled**: Whether self-collisions are enabled for the articulation (binary).
  - **SolverPositionIterationCount**: Number of position solver iterations (int).
  - **SolverVelocityIterationCount**: Number of velocity solver iterations (int).
  - **SleepThreshold**: Linear/angular velocity threshold below which bodies may go to sleep (float).
  - **StabilizationThreshold**: Stabilization threshold for the articulation root (float).
  - **CopyFromSource**: Whether simulation settings are copied from the USD source (binary).
- **Humanoid Structure**:
  - **NumberOfLinks**: Total number of links in the humanoid (int).
- **Actuator Stiffness and Damping**:
  - **StiffnessWaist**, **DampingWaist**: Waist joint stiffness and damping (float).
  - **StiffnessPelvis**, **DampingPelvis**: Pelvis joint stiffness and damping (float).
  - **StiffnessUpperArm**, **DampingUpperArm**: Upper arm joint stiffness and damping (float).
  - **StiffnessLowerArm**, **DampingLowerArm**: Lower arm joint stiffness and damping (float).
  - **StiffnessThigh0**, **DampingThigh0**: Thigh joint axis 0 stiffness and damping (float).
  - **StiffnessThigh1**, **DampingThigh1**: Thigh joint axis 1 stiffness and damping (float).
  - **StiffnessThigh2**, **DampingThigh2**: Thigh joint axis 2 stiffness and damping (float).
  - **StiffnessShin**, **DampingShin**: Shin joint stiffness and damping (float).
  - **StiffnessFoot**, **DampingFoot**: Foot joint stiffness and damping (float).
- **Transmission Parameters**:
  - **TorsoPelvisTransmissionRange**: Effective transmission ratio range for torso and pelvis joints (shape [2], float).
  - **ThighHighTransmissionNominal**: Nominal transmission ratio for high-strength thigh components (float).
  - **ThighModerateTransmissionNominal**: Nominal transmission ratio for moderate thigh components (float).
  - **ShinTransmissionNominal**: Nominal transmission ratio for shin joints (float).
  - **FootTransmissionRange**: Transmission ratio range for foot joints (shape [2], float).

Variables

- **JointPositions**: Current joint angles/positions for all actuated degrees of freedom of the humanoid (shape $[n_{\text{dof}}]$).
- **JointVelocities**: Current joint angular velocities for all actuated degrees of freedom (shape $[n_{\text{dof}}]$).
- **ActuatorCommandTargets**: Agent-selected joint-level command targets (e.g., PD target positions or equivalent setpoints) issued at approximately 60 Hz and held constant over `ActionDecimation` physics steps (shape $[n_{\text{dof}}]$).
- **AppliedJointTorques**: Resulting joint torques applied by the spring–damper actuators based on deviations from the command targets (shape $[n_{\text{dof}}]$).
- **BasePositionWorld**: World-frame position of the torso (root) link; forward progress corresponds to the $+X$ component (shape $[3]$).
- **BaseOrientationQuatWorld**: World-frame orientation of the torso (root) represented as a unit quaternion (shape $[4]$).
- **BaseLinearVelocityWorld**: World-frame linear velocity of the torso (root); forward speed corresponds to the $+X$ component (shape $[3]$).
- **BaseAngularVelocityWorld**: World-frame angular velocity of the torso (root) (shape $[3]$).
- **BaseHeightZ**: Torso height above the ground, defined as the Z component of `BasePositionWorld`; falling occurs if this value drops below a predefined threshold (shape $[\,]$, scalar).
- **BaseForwardDirWorld**: Unit vector representing the torso's forward axis expressed in world coordinates, used to assess alignment with the global $+X$ direction (shape $[3]$).
- **HeadingErrorRad**: Signed yaw-angle error between the torso's projected forward direction and the global $+X$ axis; zero indicates perfect forward alignment (shape $[\,]$, scalar).
- **UprightCosine**: Cosine of the angle between the torso's up axis and the world $+Z$ axis; a value of 1 indicates a perfectly upright posture (shape $[\,]$, scalar).
- **ForwardVelocityX**: Scalar forward velocity along the global $+X$ axis, i.e., $(\texttt{BaseLinearVelocityWorld})_x$ (shape $[\,]$, scalar).
- **LateralPositionY**: Torso lateral displacement along the global Y axis, used to penalize lateral deviation (shape $[\,]$, scalar).
- **LateralVelocityY**: Torso lateral velocity along the global Y axis, used to discourage side-to-side motion (shape $[\,]$, scalar).
- **LeftFootInContact**: Binary indicator denoting whether the left foot is in contact with the ground at the current timestep (shape $[\,]$, scalar).
- **RightFootInContact**: Binary indicator denoting whether the right foot is in contact with the ground at the current timestep (shape $[\,]$, scalar).
- **LeftFootContactForce**: Resultant ground contact force acting on the left foot expressed in world coordinates (shape $[3]$).
- **RightFootContactForce**: Resultant ground contact force acting on the right foot expressed in world coordinates (shape $[3]$).
- **EpisodeTimeSec**: Elapsed simulation time since the beginning of the current episode, useful for time-based scheduling or curricula (shape $[\,]$, scalar).

Objective

- **Objective**: Learn a locomotion policy π that maximizes forward progress while maintaining heading alignment, upright posture, lateral stability, and avoiding falls over a

finite control horizon:

$$\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T_c-1}\left(r_t^{\text{fwd}} - r_t^{\text{head}} - r_t^{\text{upr}} - r_t^{\text{lat-pos}} - r_t^{\text{lat-vel}} - r_t^{\text{fall}}\right)\right],$$

where $T_c = \text{EpisodeControlSteps}$.

- **Reward Components**:
  - $r_t^{\text{fwd}} = \text{ForwardVelocityX}_t$ (forward progress reward).
  - $r_t^{\text{head}} = |\text{HeadingErrorRad}_t|$ (heading misalignment penalty).
  - $r_t^{\text{upr}} = 1 - \text{UprightCosine}_t$ (uprightness penalty).
  - $r_t^{\text{lat-pos}} = |\text{LateralPositionY}_t|$ (lateral position deviation penalty).
  - $r_t^{\text{lat-vel}} = |\text{LateralVelocityY}_t|$ (lateral velocity penalty).
  - $r_t^{\text{fall}} = \mathbb{I}\{\text{BaseHeightZ}_t < \text{TerminationBaseHeightMin}\}$ (fall penalty).
- **Symbol Definitions**:
  - $\text{ForwardVelocityX}_t$: Forward velocity along the global $+X$ axis.
  - HeadingErrorRad_t: Signed yaw-angle error relative to the global forward direction.
  - UprightCosine_t: Cosine of the angle between the torso up axis and the world $+Z$ axis.
  - $\text{LateralPositionY}_t$: Lateral displacement along the global Y axis.
  - $\text{LateralVelocityY}_t$: Lateral velocity along the global Y axis.
  - BaseHeightZ_t: Torso height above the ground.
  - $\text{TerminationBaseHeightMin}$: Minimum base height threshold triggering termination.
  - $\mathbb{I}\{\cdot\}$: Indicator function equal to 1 if the condition holds, and 0 otherwise.

Constraints

- **Timing and Horizon**:

$$\Delta t = \text{SimDt}, \quad f_{\text{sim}} = \text{SimFrequencyHz}, \quad d = \text{ActionDecimation}.$$
$$\Delta t = \tfrac{1}{120}, \quad f_{\text{sim}} = \tfrac{1}{\Delta t}, \quad d = 2.$$
$$T_p = \text{EpisodePhysicsSteps}, \quad T_c = \text{EpisodeControlSteps}.$$
$$T_p = \tfrac{\text{EpisodeDurationSec}}{\Delta t}, \quad T_c = \tfrac{T_p}{d}.$$

- **Episode Time and Termination**:

$$0 \le t \le T_p \Delta t,$$
$$t \ge \text{EpisodeDurationSec} \ \vee \ h_t < h_{\min} \ \Rightarrow \ \text{episode terminates}.$$

- **Base State and Orientation**:

$$h_t = (\text{BasePositionWorld}_t)_z \ge 0, \qquad \|\mathbf{q}_t\|_2 = 1.$$
$$\mathbf{e}_x = [1,0,0], \qquad \|\mathbf{e}_x\|_2 = 1.$$
$$\mathbf{f}_t = R(\mathbf{q}_t)\,\mathbf{e}_x, \qquad \|\mathbf{f}_t\|_2 = 1.$$

- **Heading and Uprightness**:

$$\theta_t = \text{atan2}(f_{t,y}, f_{t,x}), \qquad -\pi < \theta_t \le \pi.$$
$$u_t = \big(R(\mathbf{q}_t)[0,0,1]\big) \cdot [0,0,1], \qquad -1 \le u_t \le 1.$$

- **Derived Velocities**:

$$v_t^x = (\text{BaseLinearVelocityWorld}_t)_x,$$
$$y_t = (\text{BasePositionWorld}_t)_y,$$
$$v_t^y = (\text{BaseLinearVelocityWorld}_t)_y.$$

- **Action Decimation**:

$$\mathbf{u}_k = \mathbf{u}_{\lfloor k/d \rfloor d}, \qquad \forall k \in \{0, \ldots, T_p - 1\}.$$

- **Joint Actuation (PD Law)**:

$$\boldsymbol{\tau}_i(k) = K_i\big(u_i(k) - q_i(k)\big) - D_i \dot{q}_i(k).$$

- **Actuator Parameter Assignment**:

$$(K_i, D_i) \in \begin{cases} (\text{StiffnessWaist}, \text{DampingWaist}) & i \in \text{waist}, \\ (\text{StiffnessPelvis}, \text{DampingPelvis}) & i \in \text{pelvis}, \\ (\text{StiffnessUpperArm}, \text{DampingUpperArm}) & i \in \text{upper arm}, \\ (\text{StiffnessLowerArm}, \text{DampingLowerArm}) & i \in \text{lower arm}, \\ (\text{StiffnessThigh0}, \text{DampingThigh0}) & i \in \text{thigh axis 0}, \\ (\text{StiffnessThigh1}, \text{DampingThigh1}) & i \in \text{thigh axis 1}, \\ (\text{StiffnessThigh2}, \text{DampingThigh2}) & i \in \text{thigh axis 2}, \\ (\text{StiffnessShin}, \text{DampingShin}) & i \in \text{shin}, \\ (\text{StiffnessFoot}, \text{DampingFoot}) & i \in \text{foot}. \end{cases}$$

- **Contact and Friction**:

$$c_t^L, c_t^R \in \{0, 1\}.$$

$$c_t^L = 0 \Rightarrow \mathbf{f}_t^L = \mathbf{0}, \qquad c_t^R = 0 \Rightarrow \mathbf{f}_t^R = \mathbf{0}.$$

$$c_t = 1 \Rightarrow \begin{cases} (f_t)_z \geq 0, \\ \sqrt{(f_t)_x^2 + (f_t)_y^2} \leq \mu\,(f_t)_z. \end{cases}$$

State

- **Description**: Proprioceptive and base-centric kinematic state required to generate and stabilize forward, upright locomotion at each 60 Hz control step.
- **State Vector**:

$$s_t = \big(q_t,\ \dot{q}_t,\ v_t,\ \omega_t,\ \theta_t,\ u_t,\ v_t^x,\ y_t,\ v_t^y,\ h_t,\ c_t^L,\ c_t^R\big).$$

- **Shape**: $[2n_{\text{dof}} + 14]$.
- **Symbol Definitions**:
  - $q_t \in \mathbb{R}^{n_{\text{dof}}}$: Joint positions.
  - $\dot{q}_t \in \mathbb{R}^{n_{\text{dof}}}$: Joint velocities.
  - $v_t \in \mathbb{R}^3$: Base linear velocity (world frame).
  - $\omega_t \in \mathbb{R}^3$: Base angular velocity (world frame).
  - $\theta_t \in \mathbb{R}$: Heading error relative to the global $+X$ axis.
  - $u_t \in [-1, 1]$: Upright cosine (alignment with the world $+Z$ axis).
  - $v_t^x \in \mathbb{R}$: Forward velocity along global $+X$.
  - $y_t \in \mathbb{R}$: Lateral position along global Y.
  - $v_t^y \in \mathbb{R}$: Lateral velocity along global Y.
  - $h_t \in \mathbb{R}$: Base height above the ground.
  - $c_t^L, c_t^R \in \{0, 1\}$: Left and right foot contact indicators.

Action

- **Description**: Continuous joint-level command targets applied at 60 Hz via action decimation. Actions are held constant over two physics steps and mapped to torques through an implicit spring–damper actuator model.

- **Action Vector**:
$$a_t \in [-1, 1]^{n_{\text{dof}}}.$$
- **Shape**: $[n_{\text{dof}}]$.
- **Bounds**: Each dimension is clipped to $[-1, 1]$ and scaled to joint-specific target ranges prior to application.
- **Symbol Definitions**:
  - a_t: Normalized actuator command targets.
  - n_{dof}: Number of actuated degrees of freedom.
  - Action decimation: Each a_t is held constant for two physics steps.

Reward

- **Description**: At each control step, the reward encourages forward velocity while penalizing heading misalignment, non-upright posture, lateral deviation, and falling.
- **Reward (compact form)**:
$$r_t = r_t^{\text{fwd}} - r_t^{\text{head}} - r_t^{\text{upr}} - r_t^{\text{lat-pos}} - r_t^{\text{lat-vel}} - r_t^{\text{fall}}.$$
- **Reward Components**:
  - $r_t^{\text{fwd}} = \text{ForwardVelocityX}_t$.
  - $r_t^{\text{head}} = |\text{HeadingErrorRad}_t|$.
  - $r_t^{\text{upr}} = 1 - \text{UprightCosine}_t$.
  - $r_t^{\text{lat-pos}} = |\text{LateralPositionY}_t|$.
  - $r_t^{\text{lat-vel}} = |\text{LateralVelocityY}_t|$.
  - $r_t^{\text{fall}} = \mathbb{I}\{\text{BaseHeightZ}_t < \text{TerminationBaseHeightMin}\}$.
- **Symbol Definitions**:
  - $\text{ForwardVelocityX}_t$: Forward velocity along global $+X$.
  - HeadingErrorRad_t: Yaw-angle error relative to global forward direction.
  - UprightCosine_t: Cosine of angle between torso up axis and world $+Z$.
  - $\text{LateralPositionY}_t$: Lateral displacement along global Y.
  - $\text{LateralVelocityY}_t$: Lateral velocity along global Y.
  - BaseHeightZ_t: Torso height above the ground.
  - $\text{TerminationBaseHeightMin}$: Minimum base height threshold (e.g., 0.6 m).
  - $\mathbb{I}\{\cdot\}$: Indicator function (1 if condition holds, 0 otherwise).

B.4 Reach

Parameters

- **Simulation Timing**
  - **PhysicsDtSeconds**: Fixed physics integration timestep (seconds).
  - **StepsPerSecond**: Number of physics steps per second ($= 1/\text{PhysicsDtSeconds}$).
  - **ControlActionDecimation**: Number of physics steps between successive control actions.
  - **ControlIntervalSeconds**: Time interval between control updates ($=$ ControlActionDecimation $\cdot$ PhysicsDtSeconds).
  - **ControlFrequencyHz**: Control update frequency ($= 1/\text{ControlIntervalSeconds}$).
- **Episode Horizon**
  - **EpisodeHorizonSeconds**: Maximum episode duration before time-limit termination.
  - **EpisodePhysicsSteps**: Total number of physics steps per episode (EpisodeHorizonSeconds / PhysicsDtSeconds).

- **ControlStepsPerEpisode**: Total number of control steps per episode (EpisodeHorizonSeconds / ControlIntervalSeconds).

- **Physics and Solver Settings**
  - **GravityEnabled**: Whether gravity is enabled in rigid body dynamics.
  - **SelfCollisionsEnabled**: Whether self-collisions are enabled for the articulation.
  - **ContactSensorsActivated**: Whether contact sensors are enabled on the robot.
  - **MaxDepenetrationVelocity**: Maximum depenetration velocity used by the physics solver.
  - **SolverPositionIterationCount**: Number of solver position iterations.
  - **SolverVelocityIterationCount**: Number of solver velocity iterations.
  - **SoftJointPosLimitFactor**: Scaling factor applied to soft joint position limits.
- **Robot Structure**
  - **NumLinks**: Number of links in the robot kinematic chain.
  - **NumArmJoints**: Number of revolute joints in the 7-DoF arm.
  - **NumFingerJoints**: Number of prismatic joints in the gripper.
  - **NumTotalJoints**: Total number of actuated joints (arm + gripper).
  - **ActionDimension**: Dimension of the continuous joint-level action vector.
- **Initial Joint Configuration**
  - **JointInitPosArm**: Initial joint positions for the 7-DoF arm (`panda_joint1`–`panda_joint7`).
  - **JointInitPosFingers**: Initial joint positions for the gripper fingers (`panda_leftfinger`, `panda_rightfinger`).
- **Actuator Groups and Limits**

  Shoulder actuator group (`panda_joint1`–`panda_joint4`):
  - **ShoulderJointCount**
  - **ShoulderEffortLimit**
  - **ShoulderVelocityLimit**
  - **ShoulderStiffness**
  - **ShoulderDamping**

  Forearm actuator group (`panda_joint5`–`panda_joint7`):
  - **ForearmJointCount**
  - **ForearmEffortLimit**
  - **ForearmVelocityLimit**
  - **ForearmStiffness**
  - **ForearmDamping**

  Hand actuator group (gripper fingers):
  - **HandJointCount**
  - **HandEffortLimit**
  - **HandVelocityLimit**
  - **HandStiffness**
  - **HandDamping**

Variables

- **Action Variables**
  - **ActionJointTargets** ($\in \mathbb{R}^{\text{ActionDimension}}$): Continuous joint-level command targets applied at each control step, including 7 revolute arm joints and 2 prismatic finger joints.
- **Proprioceptive State**

  - **JointPositions** ($\in \mathbb{R}^{\text{NumTotalJoints}}$): Current joint positions (radians for arm joints, meters for finger joints).
  - **JointVelocities** ($\in \mathbb{R}^{\text{NumTotalJoints}}$): Current joint angular or linear velocities for all actuated joints.
- **End-Effector State**
  - **ActualEndEffectorPosition** ($\in \mathbb{R}^3$): Cartesian position of the end effector (`panda_hand`) expressed in the robot base frame.
  - **ActualEndEffectorOrientationQuat** ($\in \mathbb{R}^4$): Unit quaternion representing the current end-effector orientation in the robot base frame.
- **Commanded Targets**
  - **CommandedEndEffectorPosition** ($\in \mathbb{R}^3$): Target Cartesian position for the end effector in the robot base frame, resampled within workspace bounds above the table.
  - **CommandedEndEffectorOrientationQuat** ($\in \mathbb{R}^4$): Target end-effector orientation represented as a unit quaternion in the robot base frame.
- **Tracking Errors**
  - **EndEffectorPositionError** ($\in \mathbb{R}^3$): Position tracking error defined as $\text{CommandedEndEffectorPosition} - \text{ActualEndEffectorPosition}$.
  - **EndEffectorOrientationErrorAxisAngle** ($\in \mathbb{R}^3$): Orientation tracking error between commanded and actual end-effector orientations, expressed in axis–angle form (rotation axis scaled by rotation angle).
- **Temporal and Smoothness Signals**
  - **ActionChange** ($\in \mathbb{R}^{\text{ActionDimension}}$): Difference between the current and previous action vectors, used to penalize rapid changes in control inputs and encourage smooth control.
  - **EpisodeTimeSeconds** ($\in \mathbb{R}$): Elapsed simulation time since episode start, capped at the episode horizon (e.g., 12 s).
  - **ControlStepIndex** ($\in \mathbb{N}$): Index of the current control step within the episode, $[0, \ldots, \text{ControlStepsPerEpisode} - 1]$.

Objective(JSON)

- **Objective**: Learn a reaching policy π that minimizes the time-averaged tracking error and control-induced motion over a finite horizon:

$$\max_{\pi} \mathbb{E}_{\pi}\left[-\frac{1}{T} \sum_{t=0}^{N_{\text{ctrl}}-1} \Delta t_c \left(r_t^{\text{pos}} + r_t^{\text{ori}} + r_t^{\text{vel}} + r_t^{\Delta u} \right) \right],$$

  where $T = \text{EpisodeHorizonSeconds}$, $N_{\text{ctrl}} = \text{ControlStepsPerEpisode}$, and $\Delta t_c = \text{ControlIntervalSeconds}$.
- **Reward Components**:
  - **End-effector position error**:

$$r_t^{\text{pos}} = \|\text{EndEffectorPositionError}_t\|_2^2 .$$

  - **End-effector orientation error**:

$$r_t^{\text{ori}} = \|\text{EndEffectorOrientationErrorAxisAngle}_t\|_2^2 .$$

– **Joint velocity regularization**:

$$r_t^{\text{vel}} = \sum_{j \in \mathcal{S}} \frac{\text{JointVelocities}_{t,j}^2}{\text{ShoulderVelocityLimit}^2} + \sum_{j \in \mathcal{F}} \frac{\text{JointVelocities}_{t,j}^2}{\text{ForearmVelocityLimit}^2} + \sum_{j \in \mathcal{H}} \frac{\text{JointVelocities}_{t,j}^2}{\text{HandVelocityLimit}^2},$$

where $\mathcal{S} = \{1, \ldots, 4\}$, $\mathcal{F} = \{5, \ldots, 7\}$, and $\mathcal{H} = \{8, 9\}$ denote shoulder, forearm, and hand joints, respectively.

– **Action smoothness penalty**:

$$r_t^{\Delta u} = \|\text{ActionChange}_t\|_2^2 .$$

- **Normalization**: The factor $\Delta t_c / T$ ensures that the objective approximates a time-normalized integral cost and is invariant to the control frequency.

Constraints(JSON)

- **Time and Indexing**

$$\Delta t = \text{PhysicsDtSeconds} > 0, \qquad \Delta t_c = \text{ControlIntervalSeconds} > 0$$

$$\Delta t_c = \text{ControlActionDecimation} \cdot \Delta t,$$
$$f_c = \text{ControlFrequencyHz} = \frac{1}{\Delta t_c},$$
$$f_{\text{phys}} = \text{StepsPerSecond} = \frac{1}{\Delta t}$$
$$T = \text{EpisodeHorizonSeconds} > 0$$

$$N_{\text{phys}} = \frac{T}{\Delta t} \in \mathbb{N}, \qquad N_{\text{ctrl}} = \frac{T}{\Delta t_c} \in \mathbb{N}$$

$$\forall t \in \{0, \ldots, N_{\text{ctrl}} - 1\}: \quad \begin{aligned} &\text{EpisodeTimeSeconds}(t) = t\,\Delta t_c, \\ &\text{ControlStepIndex}(t) = t \end{aligned}$$

- **Forward Kinematics and Tracking Errors**

$$\text{ActualEndEffectorPosition}(t) = f_{\text{FK,pos}}(\text{JointPositions}(t))$$

$$\begin{matrix}\text{ActualEndEffector} \\ \text{OrientationQuat}(t)\end{matrix} = f_{\text{FK,quat}}(\text{JointPositions}(t))$$

$$\begin{matrix}\text{EndEffector} \\ \text{PositionError}(t)\end{matrix} = \begin{matrix}\text{CommandedEndEffector} \\ \text{Position}(t)\end{matrix} - \begin{matrix}\text{ActualEndEffector} \\ \text{Position}(t)\end{matrix}$$

$$\left\| \begin{matrix}\text{ActualEndEffector} \\ \text{OrientationQuat}(t)\end{matrix} \right\|_2 = \left\| \begin{matrix}\text{CommandedEndEffector} \\ \text{OrientationQuat}(t)\end{matrix} \right\|_2 = 1$$

Let

$$q_{\text{err}}(t) = \begin{matrix}\text{CommandedEndEffector}\\ \text{OrientationQuat}(t)\end{matrix} \otimes \text{inverse}\left(\begin{matrix}\text{ActualEndEffector}\\ \text{OrientationQuat}(t)\end{matrix}\right),$$

then

$$\begin{matrix}\text{EndEffectorOrientation}\\ \text{ErrorAxisAngle}(t)\end{matrix} = \log\big(q_{\text{err}}(t)\big),$$
$$0 \le \|\cdot\|_2 \le \pi.$$

- **Joint Limits and Velocity Constraints**

$$\text{SoftJointPosLimitFactor} = 1.0$$

$$\forall t,\ \forall j: \quad \text{JointPosLowerUSD}[j] \le \text{JointPositions}_j(t) \le \text{JointPosUpperUSD}[j]$$

$$\forall t,\ j \in \{1,\dots,4\}: \quad |\text{JointVelocities}_j(t)| \le \text{ShoulderVelocityLimit}$$

$$\forall t,\ j \in \{5,\dots,7\}: \quad |\text{JointVelocities}_j(t)| \le \text{ForearmVelocityLimit}$$

$$\forall t,\ j \in \{8,9\}: \quad |\text{JointVelocities}_j(t)| \le \text{HandVelocityLimit}$$

- **Action Constraints and Zero-Order Hold**

$$\forall t,\ \forall j: \quad \text{JointPosLowerUSD}[j] \le \text{ActionJointTargets}_j(t) \le \text{JointPosUpperUSD}[j]$$

$$\text{ActionChange}(t) = \text{ActionJointTargets}(t) - \text{ActionJointTargets}(t-1),$$
$$\text{ActionChange}(0) = \mathbf{0}$$

$$\forall t,\ \forall \tau \in [t\Delta t_c, (t+1)\Delta t_c): \quad \text{AppliedAction}(\tau) = \text{ActionJointTargets}(t)$$

- **Workspace and Collision Constraints**

$$\forall t: \quad \text{CommandedEndEffectorPosition}(t) \in \mathcal{W}_{\text{workspace}},$$
$$\|\text{CommandedEndEffectorOrientationQuat}(t)\|_2 = 1$$

$$\forall t,\ \forall 1 \le a < b \le \text{NumLinks}: \quad \text{signed_distance}\big(\text{link}_a, \text{link}_b; \text{JointPositions}(t)\big) \ge 0$$

- **Initial Conditions**

$$\forall j: \quad \text{JointInitLower}[j] \le \text{JointPositions}_j(0) \le \text{JointInitUpper}[j]$$

State

- **Description**: Proprioceptive and task-related observation at each control step, combining robot joint states, commanded end-effector pose, pose-tracking errors, the previously applied action for smoothness reasoning, and elapsed episode time. All errors are computed in the robot base frame.
- **Variables**:
  - **JointPositions** ($\in \mathbb{R}^9$): Joint positions of the robot (`panda_joint1..7`, `panda_leftfinger`, `panda_rightfinger`).
  - **JointVelocities** ($\in \mathbb{R}^9$): Joint velocities corresponding to all actuated joints.

- **CommandedEndEffectorPosition** ($\in \mathbb{R}^3$): Target end-effector position in the robot base frame.
- **CommandedEndEffectorOrientationQuat** ($\in \mathbb{R}^4$): Target end-effector orientation as a unit quaternion.
- **EndEffectorPositionError** ($\in \mathbb{R}^3$): $e_t^{\text{pos}} = \text{CommandedEndEffectorPosition}_t - \text{ActualEndEffectorPosition}_t$.
- **EndEffectorOrientationErrorAxisAngle** ($\in \mathbb{R}^3$): Orientation tracking error expressed in axis–angle form.
- **PrevActionJointTargets** ($\in \mathbb{R}^9$): ActionJointTargets applied at time $t-1$ ($\mathbf{0}$ at $t=0$).
- **EpisodeTimeSeconds** ($\in \mathbb{R}$): Elapsed time since episode start.

- **Shape**: $[41]$.

Action

- **Description**: Continuous joint target setpoints applied to the robot's implicit PD actuators at each control step. Targets are clamped to the USD joint position limits.
- **Variables**:

$$\text{ActionJointTargets} = \begin{bmatrix} \texttt{panda_joint1_target}, \texttt{panda_joint2_target}, \\ \texttt{panda_joint3_target}, \texttt{panda_joint4_target}, \\ \texttt{panda_joint5_target}, \texttt{panda_joint6_target}, \\ \texttt{panda_joint7_target}, \texttt{panda_leftfinger_} \\ \texttt{target}, \texttt{panda_rightfinger_} \\ \texttt{target} \end{bmatrix}$$

- **Type**: Continuous.
- **Shape**: $[9]$.

Reward

- **Description**: Per-step negative quadratic tracking cost with velocity and action-smoothness regularization. The reward is scaled such that summing over the episode corresponds to the stated time-normalized objective.
- **Reward Formula**:

$$R_t = -\frac{\Delta t}{T}\left(\|e_t^{\text{pos}}\|_2^2 + \|e_t^{\text{ori}}\|_2^2 + r_t^{\text{vel}} + \|\Delta a_t\|_2^2\right),$$

  where $\Delta t = \text{ControlIntervalSeconds}$ and $T = \text{EpisodeHorizonSeconds}$.
- **Velocity Regularization**:

$$r_t^{\text{vel}} = \sum_{j=1}^{4}\left(\frac{\dot{q}_{j,t}}{v_{\text{shoulder}}^{\max}}\right)^2 + \sum_{j=5}^{7}\left(\frac{\dot{q}_{j,t}}{v_{\text{forearm}}^{\max}}\right)^2 + \sum_{j=8}^{9}\left(\frac{\dot{q}_{j,t}}{v_{\text{hand}}^{\max}}\right)^2.$$

- **Symbol Definitions**:
  - R_t: reward at control step t.
  - e_t^{pos}: end-effector position error.
  - e_t^{ori}: end-effector orientation error (axis–angle).
  - $\dot{q}_{j,t}$: joint velocity of joint j at time t.
  - $v_{\text{shoulder}}^{\max}$: shoulder velocity limit ($j = 1..4$).
  - $v_{\text{forearm}}^{\max}$: forearm velocity limit ($j = 5..7$).
  - $v_{\text{hand}}^{\max}$: hand velocity limit ($j = 8, 9$).
  - $\Delta a_t = \text{ActionJointTargets}(t) - \text{ActionJointTargets}(t-1)$, with $\Delta a_0 = \mathbf{0}$.

B.5 Lift

Parameters

- **Simulation and Control Timing**:
  - **PhysicsHz**: Physics integration frequency (Hz) (float).
  - **PhysicsDtSec**: Physics integration timestep (seconds) (float).
  - **ControlHz**: Control command application frequency (Hz) (float).
  - **ControlIntervalSec**: Time between control steps (seconds) (float).
  - **ControlEveryNSteps**: Number of physics steps between control updates (int).
- **Episode Horizon and Thresholds**:
  - **EpisodeTimeLimitSec**: Episode time limit (seconds) (float).
  - **PhysicsStepsPerEpisode**: Physics steps per episode (int).
  - **ControlStepsPerEpisode**: Control steps per episode (int).
  - **ObjectDropHeightThresholdM**: Termination threshold on object height (meters) (float).
  - **LiftHeightThresholdM**: Approx. success threshold for lift height above table (meters) (float).
- **Commanded Target Pose Range (relative to robot base)**:
  - **TargetXMinM**, **TargetXMaxM**: Forward target range (meters) (float).
  - **TargetYMinM**, **TargetYMaxM**: Lateral target range (meters) (float).
  - **TargetZMinM**, **TargetZMaxM**: Vertical target range (meters) (float).
  - **ObjectInitXApproxM**: Approx. initial forward object distance (meters) (float).
- **Robot Kinematic Sizes**:
  - **NumArmJoints**: Number of arm DOF (int).
  - **NumFingerJoints**: Number of gripper finger joints (int).
  - **NumJointsTotal**: Total controlled joints (arm + gripper) (int).
  - **NumLinks**: Total number of links (int).
  - **ShoulderJointsCount**: Shoulder group joint count (panda_joint1–4) (int).
  - **ForearmJointsCount**: Forearm group joint count (panda_joint5–7) (int).
  - **FingerJointsCount**: Gripper finger joint count (int).
- **Initialization**:
  - **ArmInitPosRad**: Initial arm joint positions (radians) (shape $[\mathrm{NumArmJoints}]$, float).
  - **FingerInitPosM**: Initial finger joint positions (meters) (shape $[\mathrm{NumFingerJoints}]$, float).
- **Actuation (PD) and Limits**:
  - **ShoulderStiffness**, **ShoulderDamping**: PD gains for shoulder joints (float).
  - **ShoulderEffortLimitNm**, **ShoulderVelocityLimitRadPerSec**: Effort/velocity limits for shoulder joints (float).
  - **ForearmStiffness**, **ForearmDamping**: PD gains for forearm joints (float).
  - **ForearmEffortLimitNm**, **ForearmVelocityLimitRadPerSec**: Effort/velocity limits for forearm joints (float).
  - **GripperStiffness**, **GripperDamping**: PD gains for gripper finger joints (float).
  - **GripperEffortLimit**, **GripperVelocityLimit**: Effort/velocity limits for gripper fingers (float).
- **Simulation Options**:
  - **ActivateContactSensors**: Whether contact sensors are activated (binary).
  - **DisableGravity**: Whether gravity is disabled for the robot (binary).
  - **SelfCollisionsEnabled**: Whether self-collisions are enabled (binary).
- **Physics Solver and Limits**:

  - **MaxDepenetrationVelocity**: Maximum depenetration velocity (float).
  - **SolverPositionIterationCount**: Number of solver position iterations (int).
  - **SolverVelocityIterationCount**: Number of solver velocity iterations (int).
  - **SoftJointPositionLimitFactor**: Scaling factor for soft joint position limits (float).
- **Observation/Command Toggles**:
  - **ObservesJointPositions**: Observation includes joint positions (binary).
  - **ObservesJointVelocities**: Observation includes joint velocities (binary).
  - **ObservesObjectPoseRelToBase**: Observation includes object pose relative to base (binary).
  - **ObservesTargetPose**: Observation includes commanded target pose (binary).
  - **ObservesPreviousAction**: Observation includes previous action (binary).
  - **CommandPoseConstantPerEpisode**: Commanded target pose constant within episode (binary).
- **Assigned Values (`_values`)**:
  - **Timing**:
    $$\begin{aligned}&\text{PhysicsHz} = 100.0,\\ &\text{PhysicsDtSec} = 0.01,\\ &\text{ControlHz} = 50.0,\\ &\text{ControlIntervalSec} = 0.02,\\ &\text{ControlEveryNSteps} = 2.\end{aligned}$$
  - **Episode**:
    $$\begin{aligned}&\text{EpisodeTimeLimitSec} = 5.0,\\ &\text{PhysicsStepsPerEpisode} = 500,\\ &\text{ControlStepsPerEpisode} = 250.\end{aligned}$$
  - **Thresholds**:
    $$\begin{aligned}&\text{ObjectDropHeightThresholdM} = -0.05,\\ &\text{LiftHeightThresholdM} = 0.04.\end{aligned}$$
  - **Target Ranges**:
    $$\begin{aligned}&\text{TargetXMinM} = 0.4, \quad \text{TargetXMaxM} = 0.6,\\ &\text{TargetYMinM} = -0.25, \quad \text{TargetYMaxM} = 0.25,\\ &\text{TargetZMinM} = 0.25, \quad \text{TargetZMaxM} = 0.5,\\ &\text{ObjectInitXApproxM} = 0.5.\end{aligned}$$
  - **Sizes**:
    $$\begin{aligned}&\text{NumArmJoints} = 7, \quad \text{NumFingerJoints} = 2,\\ &\text{NumJointsTotal} = 9, \quad \text{NumLinks} = 11,\\ &\text{ShoulderJointsCount} = 4, \quad \text{ForearmJointsCount} = 3,\\ &\text{FingerJointsCount} = 2.\end{aligned}$$
  - **Initialization**:
    $$\begin{aligned}&\text{ArmInitPosRad} = [0.0, -0.569, 0.0, -2.81, 0.0, 3.037, 0.741],\\ &\text{FingerInitPosM} = [0.04, 0.04].\end{aligned}$$

- **PD Gains and Limits**:

$$\begin{aligned}&\text{ShoulderStiffness} = 80.0,\\ &\text{ShoulderDamping} = 4.0,\\ &\text{ShoulderEffortLimitNm} = 87.0,\\ &\text{ShoulderVelocityLimitRadPerSec} = 2.175,\\ &\text{ForearmStiffness} = 80.0,\\ &\text{ForearmDamping} = 4.0,\\ &\text{ForearmEffortLimitNm} = 12.0,\\ &\text{ForearmVelocityLimitRadPerSec} = 2.61,\\ &\text{GripperStiffness} = 2000.0,\\ &\text{GripperDamping} = 100.0,\\ &\text{GripperEffortLimit} = 200.0,\\ &\text{GripperVelocityLimit} = 0.2.\end{aligned}$$

- **Options**:

$$\begin{aligned}&\text{ActivateContactSensors} = 0,\\ &\text{DisableGravity} = 0,\\ &\text{SelfCollisionsEnabled} = 1.\end{aligned}$$

- **Solver**:

$$\begin{aligned}&\text{MaxDepenetrationVelocity} = 5.0,\\ &\text{SolverPositionIterationCount} = 8,\\ &\text{SolverVelocityIterationCount} = 0,\\ &\text{SoftJointPositionLimitFactor} = 1.0.\end{aligned}$$

- **Command**:

$$\text{CommandPoseConstantPerEpisode} = 1.$$

Variables

- **JointPositions**: Current positions of all controllable robot joints, consisting of 7 revolute arm joints and 2 prismatic gripper finger joints (shape $[\text{NumJointsTotal}]$).
- **JointVelocities**: Current velocities of all controllable robot joints, including the 7 arm revolute joints and 2 prismatic gripper finger joints (shape $[\text{NumJointsTotal}]$).
- **ObjectPositionRelToBase**: Three-dimensional position of the object's reference frame expressed relative to the robot base frame (x, y, z) (shape $[3]$).
- **TargetPosition**: Commanded end-effector target position expressed relative to the robot base frame (x, y, z). The target remains constant within an episode but is observable by the agent (shape $[3]$).
- **Action**: Control signal selected by the agent at the current control step for all controllable joints (arm + gripper). Depending on the controller, this corresponds to joint-space position targets or target deltas (shape $[\text{NumJointsTotal}]$).
- **PreviousAction**: Action applied at the previous control step for all controllable joints (arm + gripper), included to preserve the Markov property when action-smoothing or implicit actuator dynamics are used (shape $[\text{NumJointsTotal}]$).

Objective

- **Objective**: Learn a lifting policy π that brings the object close to the commanded target, lifts it above a threshold height, maintains smooth control, respects joint velocity limits,

and avoids dropping the object:

$$\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T_c-1}\left(r_t^{\text{reach}} + r_t^{\text{lift}} - r_t^{\Delta a} - r_t^{\dot{q}} - r_t^{\text{drop}} - r_t^{\text{grip}}\right)\right],$$

where $T_c = \text{ControlStepsPerEpisode}$.

- **Reward Components**:
  - **Reaching term**:
$$r_t^{\text{reach}} = -\left\|\mathbf{p}_t - \mathbf{p}^*\right\|_2^2.$$
  - **Lifting term**:
$$r_t^{\text{lift}} = \max\left(0,\ z_t - z_{\text{lift}}\right).$$
  - **Action smoothness penalty**:
$$r_t^{\Delta a} = \left\|\mathbf{a}_t - \mathbf{a}_{t-1}\right\|_2^2.$$
  - **Joint velocity penalty**:
$$r_t^{\dot{q}} = \sum_{j\in\mathcal{S}}\left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{sh}}^{\max}}\right)^2 + \sum_{j\in\mathcal{F}}\left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{fa}}^{\max}}\right)^2 + \sum_{j\in\mathcal{G}}\left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{gr}}^{\max}}\right)^2.$$
  - **Drop penalty**:
$$r_t^{\text{drop}} = \mathbb{I}\{z_t < z_{\text{drop}}\}.$$
  - **Post-lift gripper closure penalty**:
$$r_t^{\text{grip}} = \mathbb{I}\{z_t \geq z_{\text{lift}}\}\sum_{j\in\mathcal{G}} q_{t,j}.$$

- **Symbol Definitions**:
  - $\mathbf{p}_t = \text{ObjectPositionRelToBase}_t \in \mathbb{R}^3$: Object position relative to the robot base.
  - $\mathbf{p}^* = \text{TargetPosition}$: Commanded target position.
  - $z_t = (\mathbf{p}_t)_z$: Object height above the base frame.
  - $z_{\text{lift}} = \text{LiftHeightThresholdM}$: Height threshold indicating successful lift.
  - $z_{\text{drop}} = \text{ObjectDropHeightThresholdM}$: Height threshold triggering drop penalty.
  - $\mathbf{a}_t = \text{Action}_t$, $\mathbf{a}_{t-1} = \text{PreviousAction}_t$: Current and previous control actions.
  - $\dot{q}_{t,j} = \text{JointVelocities}_{t,j}$: Joint velocity of joint j at time t.
  - $\mathcal{S} = \{1,\ldots,4\}$: Shoulder joint indices.
  - $\mathcal{F} = \{5,\ldots,7\}$: Forearm joint indices.
  - $\mathcal{G} = \{\text{NumArmJoints}+1,\ldots,\text{NumJointsTotal}\}$: Gripper joint indices.
  - $\dot{q}_{\text{sh}}^{\max} = \text{ShoulderVelocityLimitRadPerSec}$, $\dot{q}_{\text{fa}}^{\max} = \text{ForearmVelocityLimitRadPerSec}$, $\dot{q}_{\text{gr}}^{\max} = \text{GripperVelocityLimit}$: Joint velocity limits for shoulder, forearm, and gripper joints.
  - $\mathbb{I}\{\cdot\}$: Indicator function equal to 1 if the condition holds, and 0 otherwise.

Constraints

- **Target Pose Constraints**:
$$\mathbf{p}^* = (x^*, y^*, z^*),$$
$$x_{\min} \leq x^* \leq x_{\max},$$
$$y_{\min} \leq y^* \leq y_{\max},$$
$$z_{\min} \leq z^* \leq z_{\max}.$$
$$\mathbf{p}_t^* = \mathbf{p}_0^*, \qquad \forall t \in \{0,\ldots,T_c-1\}.$$

- **Initial Configuration**:

$$q_{0,i} = q_i^{\text{arm}}, \qquad i \in \{1, \dots, n_{\text{arm}}\}.$$
$$q_{0,n_{\text{arm}}+j} = q_j^{\text{grip}}, \qquad j \in \{1, 2\}.$$

- **Gripper Symmetry**:

$$q_{t,n_{\text{arm}}+1} = q_{t,n_{\text{arm}}+2},$$
$$a_{t,n_{\text{arm}}+1} = a_{t,n_{\text{arm}}+2}, \qquad \forall t.$$

- **Joint Position Limits**:

$$q_t \in \mathcal{Q}_{\text{USD}}(\alpha), \qquad \alpha = \text{SoftJointPositionLimitFactor}.$$
$$q_{t,n_{\text{arm}}+j} \geq 0, \qquad j \in \{1, 2\}.$$

- **Joint Velocity Limits**:

$$|\dot{q}_{t,i}| \leq \dot{q}_{\text{sh}}^{\max}, \qquad i \in \{1, \dots, 4\}.$$
$$|\dot{q}_{t,i}| \leq \dot{q}_{\text{fa}}^{\max}, \qquad i \in \{5, 6, 7\}.$$
$$|\dot{q}_{t,n_{\text{arm}}+j}| \leq \dot{q}_{\text{gr}}^{\max}, \qquad j \in \{1, 2\}.$$

- **Action Decimation and Timing**:

$$a_{t \cdot d+m} = a_t, \qquad m \in \{0, \dots, d-1\}.$$
$$\Delta t_c = d\, \Delta t_p, \qquad f_c = \tfrac{1}{\Delta t_c}.$$
$$a_{t-1} = \text{PreviousAction}_t, \qquad t \geq 1.$$

- **Termination Conditions**:

$$t\, \Delta t_c \geq T_{\max} \;\; \vee \;\; z_t < z_{\text{drop}} \;\Rightarrow\; \text{episode terminates.}$$

- **Index Sets**:

$$k \in \{0, \dots, T_p - 1\}, \qquad t \in \{0, \dots, T_c - 1\}.$$

- **Self-Collision Constraint**:

$$\text{SelfCollisionsEnabled} = 1 \;\Rightarrow\; q_t \text{ is free of self-penetration}, \qquad \forall t.$$

- **Object Initialization**:

$$z_0 = z_{\text{table}}, \qquad x_0 \approx x_{\text{init}}.$$

- **Symbol Definitions**:
  - $\mathbf{p}^*$: Commanded target position.
  - $(x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}, z_{\max})$: Target workspace bounds.
  - $q_t, \dot{q}_t$: Joint positions and velocities.
  - a_t: Action at control step t.
  - $n_{\text{arm}} = \text{NumArmJoints}$, $n_{\text{tot}} = \text{NumJointsTotal}$.
  - $\dot{q}_{\text{sh}}^{\max}$, $\dot{q}_{\text{fa}}^{\max}$, $\dot{q}_{\text{gr}}^{\max}$: Velocity limits for shoulder, forearm, and gripper joints.
  - $d = \text{ControlEveryNSteps}$: Action decimation factor.
  - $\Delta t_p = \text{PhysicsDtSec}$, $\Delta t_c = \text{ControlIntervalSec}$.
  - $T_{\max} = \text{EpisodeTimeLimitSec}$.
  - $z_{\text{drop}} = \text{ObjectDropHeightThresholdM}$.
  - z_{table}: Table height.
  - $x_{\text{init}} = \text{ObjectInitXApproxM}$.

State

- **Description**: Concatenated proprioceptive and task-relevant exteroceptive observations at each control step. The state includes all joint positions and velocities (arm and gripper), the object position relative to the robot base, the per-episode constant target position, and the most recent action. This representation provides sufficient information for smooth closed-loop manipulation and grasp stabilization.
- **Variables**:
$$s_t = \left[q_t,\ \dot{q}_t,\ \mathbf{o}_t,\ \mathbf{p}^*,\ a_{t-1}\right]$$
- **Shape**: $[33]$.
- **Symbol Definitions**:
  - $q_t \in \mathbb{R}^9$: Joint positions (7 arm joints + 2 gripper fingers).
  - $\dot{q}_t \in \mathbb{R}^9$: Joint velocities.
  - $\mathbf{o}_t \in \mathbb{R}^3$: Object position relative to the robot base.
  - $\mathbf{p}^* \in \mathbb{R}^3$: Per-episode constant target position.
  - $a_{t-1} \in \mathbb{R}^9$: Action applied at the previous control step.

Action

- **Description**: Continuous joint-space target deltas applied at 50 Hz via PD-controlled joints. Each action specifies a per-step change in joint targets (radians for arm joints and meters for the prismatic gripper fingers). The two gripper finger commands are constrained to be identical, enforcing symmetric gripper motion.
- **Variables**:
$$a_t = \left[\Delta q^{\text{arm}}_{t,1:7},\ \Delta q^{\text{grip}}_t,\ \Delta q^{\text{grip}}_t\right]$$
- **Shape**: $[9]$.
- **Type**: Continuous.
- **Symbol Definitions**:
  - $\Delta q^{\text{arm}}_{t,i}$: Target increment for arm joint i.
  - Δq^{grip}_t: Target increment for each gripper finger.
  - Action magnitudes are implicitly bounded by joint velocity limits and the control interval.

Reward

- **Description**: A dense per-step reward that encourages moving the object toward the commanded target, lifting it above a height threshold, and executing smooth, low-velocity joint motions. Object drops are penalized, and once lifting is achieved, positive gripper openings are discouraged to promote stable grasping.
- **Objective**:
$$\max_{\pi}\ \mathbb{E}_{\pi}\left[\sum_{t=0}^{T_c-1}\left(r^{\text{pos}}_t + r^{\text{lift}}_t - r^{\Delta}_t - r^{\text{vel}}_t - r^{\text{drop}}_t - r^{\text{grip}}_t\right)\right].$$
- **Reward Components**:
  - Position tracking:
$$r^{\text{pos}}_t = -\|\mathbf{o}_t - \mathbf{p}^*\|_2^2.$$
  - Lift bonus:
$$r^{\text{lift}}_t = \max\left(0,\ o_{t,z} - h_{\text{lift}}\right).$$
  - Action smoothness:
$$r^{\Delta}_t = \|a_t - a_{t-1}\|_2^2.$$

- Joint velocity penalty:

$$r_t^{\text{vel}} = \sum_{j \in \mathcal{J}_{\text{sh}}} \left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{sh}}^{\max}} \right)^2 + \sum_{j \in \mathcal{J}_{\text{fa}}} \left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{fa}}^{\max}} \right)^2 + \sum_{j \in \mathcal{J}_{\text{gr}}} \left(\frac{\dot{q}_{t,j}}{\dot{q}_{\text{gr}}^{\max}} \right)^2.$$

- Drop penalty:

$$r_t^{\text{drop}} = \mathbb{I}\{o_{t,z} < h_{\text{drop}}\}.$$

- Gripper closure encouragement:

$$r_t^{\text{grip}} = \mathbb{I}\{o_{t,z} \geq h_{\text{lift}}\} \sum_{j \in \mathcal{J}_{\text{gr}}} q_{t,j}.$$

- **Symbol Definitions**:
  - T_c: Number of control steps per episode.
  - $o_{t,z}$: Object height at time t.
  - h_{lift}: Lift success height threshold.
  - h_{drop}: Drop (failure) height threshold.
  - $\mathcal{J}_{\text{sh}} = \{1, \ldots, 4\}$ (shoulder), $\mathcal{J}_{\text{fa}} = \{5, 6, 7\}$ (forearm), $\mathcal{J}_{\text{gr}} = \{8, 9\}$ (gripper).
  - $\mathbb{I}\{\cdot\}$: Indicator function.

C Agent Prompt Templates

Below are partial examples of the prompts used for each agent.

C.1 Modeling and Implementation Agent Prompts

Parameter Agent Prompt

Here is the natural language description of an optimization problem:

—
{description}
—

Your task is to identify and extract parameters from the description.
The parameters are values that are already known.
Please generate the output in the following format:

⋮

Where SYMBOL is a string representing the parameter (use CamelCase),
SHAPE is the shape of the parameter (e.g. "[]" for scalar, or "[N, M]" for a matrix of size $N \times M$
where N and M are scalar parameters),
DEFINITION is a string describing the parameter, and TYPE is one of "int", "float", or "binary".

⋮

- Put all the parameters in a single json object.
- Do not generate anything after and before the json object.
Take a deep breath and think step by step.

Robotics Agent Prompt

Here is the natural language description of task what we will do:
——
{description}
——

And is the documentation for the code that configures the robot in the simulator.
——
{robot document}
——

This is the code for the robot that will be used in this task.
——
{robot code}
——

⋮

Your task is here: Based on the provided document and code, understand how this robot operates.

The following is the output format:
——
- the description of the robot 1.
- the description of the robot 2.
- the description of the robot 3.
- the description of the robot...
——
-It must include the information about the given parameters.
-It must include an explanation of how the robot's state changes in response to joint movements.
-It must describe the specific structure, including the position and configuration of the joints.
-Please provide only an explanation of how the robot works.
-Do not include explanation about tasks or code specifics. Only the description of the robot.

Take a deep breath and think step by step.

Objective Agent Prompt

Here is the natural language description of an optimization problem:

——
{description}
——

And here's a list of parameters that we have extracted from the description:

——
{params}
——
And here is the natural language description of the robot.

——
{robot description}
——

Your task is to identify and extract the optimization objective from the description.
The objective is the goal that the optimization model is trying to achieve (e.g. maximize profit,

minimize cost).
The objective will be used in MDP.
Please generate the output in the following format:

=====
OBJECTIVE: objective description
=====

⋮

- Do not generate anything after and before the objective.
Take a deep breath and think step by step.

Variable Agent Prompt

Here is the natural language description of an optimization problem:

—
{description}
—

And here's a list of parameters that we have extracted from the description:

—
{params}
—

Your task is to identify and extract variables from the description.
The variables are values that are not known and need to be determined by the optimization model.
Please generate the output in the following format:

⋮

Where SYMBOL is a string representing the variable (use CamelCase),
SHAPE is the shape of the variable (e.g. "[]" for scalar, or "[N, M]" for a matrix of size $N \times M$),
and DEFINITION is a string describing the variable.

⋮

- Put all the parameters in a single json object.
- Do not generate anything after and before the json object.
Take a deep breath and think step by step.

Objective Modeling Agent Prompt

Here is the natural language description of an optimization problem:

—
{description}
—

Parameters: {params}
Variables: {vars}
Constraints: {constraints}

Your task is to model the following objective mathematically in LaTeX for the MDP formulation:

{objective}

MDP objective formula will be Expectation of action that is chosen at each time step.
Please generate the output in the following format:

=====
objective formulation in LaTeX, between $...$,
=====

⋮

- You can only use existing parameters and variables in the formulation.
- But you can change the shape of variable and parameters.
- Do not generate anything after and before the objective.
Take a deep breath and think step by step.

Constraints Modeling Agent Prompt

You are an expert in optimization modeling.

Here is the natural language description of an optimization problem:

—
{description}
—

Parameters: {params}
Variables: {vars}

Your task is to model the following constraint mathematically in LaTeX for the MDP formulation:

{constraints}

The constraints are the conditions that must be satisfied by the variables.
Please generate the output in the following format:

=====
constraint formulation in LaTeX, between $...$,
=====

⋮

- You can only use existing parameters and variables in the formulation.
- Do not generate anything after and before the constraint.
Take a deep breath and think step by step.

SAR Agent Prompt

You are an expert in reinforcement learning and scheduling optimization.
Your task is to extract key components for formulating a Markov decision process (MDP)

Here is the natural language description of the scheduling problem:
—
{description}
—

Parameters: {params}
Variables: {vars}
Constraints: {constraints}
Objective: {objective}

Your task is to identify and define the following components for reinforcement learning:

1. State Space
The state space represents the environment's status at a single time step.

⋮

2. Action Space
The action space is defined as the set of all possible actions that the agent can take.

⋮

3. Reward Function
The reward function quantifies the quality of the agent's decision.

⋮

Take a deep breath and think step by step.

Scene Cfg. Agent

Here is the MDP of task what we will do:
—
{SAR}
—

⋮

Your task is to generate a scene configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references."

Belows are document of scene configuration class:

{document scene}

Output only the complete working Python class code as a complete python object in this format :
@configclass
class SceneCfg(InteractiveSceneCfg):

[CRITICAL RULES — SceneCfg]
- SceneCfg(InteractiveSceneCfg) MUST define ONLY spawnable scene assets.
- Allowed cfg classes inside SceneCfg:
• AssetBaseCfg
• ArticulationCfg
• RigidObjectCfg
• Sensor cfgs (if any)

- DO NOT define or instantiate SceneEntityCfg inside SceneCfg. SceneEntityCfg is NOT a scene asset and MUST NOT appear in SceneCfg.

- DO NOT create alias entities such as "robot entity", "time", "goal", or similar. The ONLY valid identifiers for later reference are the variable names used for spawnable assets (e.g., "robot", "object").

- prim path is ONLY allowed in:
• AssetBaseCfg
• ArticulationCfg
• RigidObjectCfg
It is STRICTLY FORBIDDEN in any other cfg.

If any SceneEntityCfg appears in SceneCfg output, the result is INVALID.

- Do not include any commented-out example blocks or explanatory sections (such as state space, action space, reward, or termination) in the output.
- Only output executable and necessary configuration code for ManagerBasedRLEnv in Isaac Lab.
- Exclude any Markdown, comments, docstrings, or reference annotations. The code must be runnable directly in Isaac Lab without further cleanup.

Action Cfg. Agent Prompt

⋮

Your task is to generate a Action configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references."

Below is code of Scenecfg. By referring to this, you can identify the components of the environment and match the variables accordingly.

{scene code}

Belows are document of Action configuration class:

{document action}

Output only the complete working Python class code as a complete python object in this format :

@configclass
class ActionsCfg:

- Include only actual ActionTermCfg items.
- Do not include dictionaries or metadata fields such as components, state space, action space, or constraints.
- Take a deep breath and think step by step. You will be awarded a million dollars if you get this

right.

Observation Cfg. Agent Prompt

⋮

Your task is to generate a observation configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references.

Below is code of SceneCfg. By referring to this, you can identify the components of the environment and match the variables accordingly.
{scene code}

Belows are document of observation configuration class:
{document obs}

Output only the complete working Python class code as a complete python object in this format :
{
@configclass
class ObservationsCfg:
class PolicyCfg(ObsGroup):

⋮

def post_init(self) − > None:

⋮

}

[CRITICAL RULES — ObservationsCfg]

- SceneEntityCfg is used ONLY as a selector for existing scene assets.
- SceneEntityCfg MUST reference an existing asset by name ONLY.

Correct usage:
SceneEntityCfg("robot")
SceneEntityCfg("robot", joint names=[...])
SceneEntityCfg("robot", body names=[...])

STRICTLY FORBIDDEN in SceneEntityCfg:
- prim path
- articulations
- bodies
- dof names
- enable * flags

⋮

Event Cfg. Agent Prompt

⋮

Your task is to generate a Event configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and

reward function, using the document and examples below as references.

Below is code of SceneCfg. By referring to this, you can identify the components of the environment and match the variables accordingly.
{scene code}

Belows are document of Event configuration class:
{event document}

Output only the complete working Python class code as a complete python object in this format :
{
@configclass
class EventCfg:

⋮

}

[CRITICAL RULES — Event / Termination]
- All asset references MUST use SceneEntityCfg with a valid asset name.
- SceneEntityCfg MUST NOT contain prim path or any spawn-related fields.
- Only SceneCfg-defined asset names are allowed.

Example (valid):
`params={{"asset_cfg": SceneEntityCfg("robot")}}`
Example (invalid):
`SceneEntityCfg(prim_path="/World/robot")` `SceneEntityCfg("robot_entity")`

Custom Reward Func Agent Prompt

⋮

==========
YOUR TASK (STRICT): OUTPUT A SINGLE PYTHON FUNCTION ONLY
==========

You must output ONLY the following Python function code (no extra text):

def custom reward(env: ManagerBasedRLEnv, *args, **kwargs) − > torch.Tensor:
...

[CRITICAL RULES — Reward Custom]

- SceneEntityCfg is allowed ONLY inside term parameters.
- SceneEntityCfg MUST NOT include prim path.
- SceneEntityCfg MUST reference ONLY SceneCfg-defined asset names.

If an entity does not exist in SceneCfg, DO NOT infer or create it.

==========
STRICT OUTPUT CONSTRAINTS
==========

A) Output must be ONLY the function code.
- No explanations, no markdown, no extra text.

B) Do NOT add any imports.
- Assume torch and ManagerBasedRLEnv are already available.

C) Do NOT define any new helper functions or classes.
- You MAY store per-env buffers using function attributes only:
custom reward. prev pot, custom reward. stage, etc.

D) The returned tensor MUST have shape [N] where N = env.num envs.

E) torch.nan to num must use scalar constants:
torch.nan to num(x, nan=0.0, posinf=0.0, neginf=0.0)

⋮

==========
EXECUTION ORDER
==========

Step 1) Identify available signals (NO guessing)
- Only use entities/sensors/signals explicitly present in {scene code} and {observation code}.
- Only use access paths and patterns described in {document reward custom}.

Step 2) Choose the ONE target distance for potential
You MUST define a single distance to target (scalar per env) using exactly ONE of:

⋮

(Option S: State stabilization distance)
If neither L nor B is feasible, define distance to target as L2 norm to a desired state
explicitly provided in {vars} or implied by {objective model} (e.g., zero velocities, zero angles).

IMPORTANT:
- You must justify the choice internally by checking which signals exist.
- You must NOT invent a goal position or target angle if it is not present.

Step 3) Define ForwardPotential
- ForwardPotential MUST be negative distance to the chosen target:
ForwardPotential = - distance to target / max(env.step dt, eps)

Step 4) Potential-difference progress
- r progress = ForwardPotential(t+1) - ForwardPotential(t)
- Store and update previous potential internally using function attributes.
- On first call or when an episode just terminated/time out: set prev potential := potential so progress ≈ 0.

⋮

- If no additional shaping terms are available, keep a single-stage structure.

Step 6) Regularization penalties (MANDATORY when signals exist)
- These penalties must be NON-ZERO by default, normalized, and clamped.

Add ONLY if signals exist:
(a) Action smoothness (if prev action exists):
pen action = mean(abs(action - prev action))

(b) Speed penalty (MANDATORY if any velocity signal exists):

- If base/root linear velocity exists: L2 / v ref
- Else: joint velocity penalty / qd ref

(c) Angular speed penalty (if base/root angular vel exists):
$penang = sum((rootangvel/wref)^2)$

(d) Effort/energy penalty (if torque/effort exists):
$peneffort = mean((jointeffort/taueff)^2)ORmean(abs(tau * qd))normalized$

(e) Optional safety terms (only if signals exist and relevant):
- projected gravity / uprightness
- joint limit proximity

Normalization and caps:
v ref = 1.0, w ref = 3.0, qd ref = 5.0, tau ref = 100.0
TERM CAP = 10.0
Each penalty term must be clamped to TERM CAP.

Step 7) Survival / feasibility reward (OPTIONAL, SAFE)
⋮

Step 8) Terminal gating (MANDATORY)
⋮

Step 9) Final numeric safety (MANDATORY)
- reward = torch.nan to num(reward, nan=0.0, posinf=0.0, neginf=0.0)
- return reward with shape [N]

==========
CORE DESIGN REQUIREMENTS
==========

⋮

==========
FINAL REMINDER
==========

Output ONLY the Python function code:

def custom reward(env: ManagerBasedRLEnv, *args, **kwargs) $->$ torch.Tensor:

⋮

No other text.

Reward Cfg. Agent Prompt

⋮

Your task is to generate a Reward configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references.

Below is code of SceneCfg. By referring to this, you can identify the components of the environment and match the variables accordingly.
Referring to the SceneCfg, write the float-type values inside params as constants
{scene code}

Belows are document of Reward configuration class:
{document reward}

Output only the complete working Python class code as a complete python object in this format :
{
@configclass
class RewardsCfg:
}

⋮

- Do not generate anything after and before the code!
- Use mdp.custom_reward functions defined above.
- Match the parameter format with the custom reward functions.
- Therefore, always include both `"args": ...` and `"kwargs": ...` in the params field, even if they are empty.
- Take a deep breath and think step by step. You will be awarded a million dollars if you get this right.

Termination Cfg. Agent Prompt

⋮

Your task is to generate a Termination configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references.

Below is code of SceneCfg. By referring to this, you can identify the components of the environment and match the variables accordingly.
Referring to the SceneCfg, write the float-type values inside params as constants
{scene code}

Belows are document of Termination configuration class:
{document termination}

Output only the complete working Python class code as a complete python object in this format :
{
@configclass
class TerminationsCfg:
}

⋮

[CRITICAL RULES — Event / Termination]
- All asset references MUST use SceneEntityCfg with a valid asset name.
- SceneEntityCfg MUST NOT contain prim path or any spawn-related fields.
- Only SceneCfg-defined asset names are allowed.

⋮

- Do not generate anything after and before the code!
- Take a deep breath and think step by step. You will be awarded a million dollars if you get this right.

Policy Cfg. Agent Prompt

⋮

Your task is to generate a policy configuration class by extracting the paths of each component defined in the description, along with the MDP-defined objective, state space, action space, and reward function, using the document and examples below as references.

Belows are document of policy configuration class:
{document policy}

Output only the complete working Python class code as a complete python object in this format :
from isaaclab.utils import configclass
from isaaclab_rl.rsl_rl import RslRlOnPolicyRunnerCfg, RslRlPpoActorCriticCfg, RslRlPpoAlgorithmCfg

@configclass
class CustomPPORunnerCfg(RslRlOnPolicyRunnerCfg):

⋮

- Do not generate anything after and before the code!
- Take a deep breath and think step by step. You will be awarded a million dollars if you get this right.

Coding Agent Prompt

Your task is to take multiple code fragments generated by other LLM agents and assemble them into the final code.
First, import the necessary libraries, then write the scene configuration, observation configuration, action configuration, event configuration, termination configuration, and reward configuration as they are, and finally add the EnvCfg to complete the Python code.

Below is the list of libraries that need to be imported in the code
—
from __future__ import annotations
import math
import isaaclab.sim as sim_utils
from isaaclab.actuators import ImplicitActuatorCfg
from isaaclab.assets import ArticulationCfg
from isaaclab.assets import ArticulationCfg, AssetBaseCfg, RigidObjectCfg
from isaaclab.utils.assets import ISAACLAB_NUCLEUS_DIR
from isaaclab.envs import ManagerBasedRLEnvCfg
from isaaclab.managers import EventTermCfg as EventTerm
from isaaclab.managers import ObservationGroupCfg as ObsGroup
from isaaclab.managers import ObservationTermCfg as ObsTerm
from isaaclab.managers import RewardTermCfg as RewTerm
from isaaclab.managers import SceneEntityCfg
from isaaclab.managers import TerminationTermCfg as DoneTerm
from isaaclab.scene import InteractiveSceneCfg
from isaaclab.utils import configclass

from isaaclab.utils.math import wrap_to_pi
import isaaclab.envs.mdp as mdp
import torch
—

Below are the six components that make up the code.
—
1. scene configuration.
{scene code}
2. observation configuration.
{obs code}
3. action configuration.
{action code}
4. event configuration.
{event code}
5. termination configuration.
{termination code}
6. reward configuration.
{reward code}
—

Paste the above code first, and then add the environment configuration.
If a variable is used without being defined, it should be properly defined first

The following is the format about CustomEnvCfg:

@configclass
class CustomEnvCfg(ManagerBasedRLEnvCfg):

- Do not generate anything after and before the code!
- Take a deep breath and think step by step. You will be awarded a million dollars if you get this right.

C.2 Error Correction Agent Prompt and Checklists

Error Correction Agent Prompt

You are the ARPG Modeling Error Correction Agent. Diagnose only; do not write a revised artifact.
Task description:
—
{task_description}
—
{common_inputs_section}
Current stage: {stage}
Relevant upstream artifacts:
—
{upstream_context}
—
Stage checklist:
—
{checklist}
—
Artifact to inspect:
—

{artifact}
—
Identify only concrete violations supported by the inputs. Keep every evidence quote at most 20 words.
Return exactly one valid JSON object and nothing else:
{
"violations": [
{
"item": "violated checklist requirement",
"evidence": "short quote or precise location",
"reason": "why this is a violation",
"fix": "localized instruction for the original stage agent"
}
],
"summary_fix_plan": "short stage-local revision plan"
}

Stage-specific checklists. The following seven checklists specify the criteria and fail triggers for each modeling stage. Each invocation receives only the checklist associated with its current modeling stage.

Parameters Agent Checklist

Checklist
- Are all task-critical constants explicitly defined with concrete numeric values?
- Are all task-applicable environment bounds and limits clearly specified?
- Are all values free from vague expressions such as "approximately", an unqualified "default", or "reasonable"? An explicitly configured default with a concrete value is not vague.
- Are units and scales consistent across parameters?
- Are there no internal contradictions among parameters?

Fail triggers
- Presence of placeholders such as TBD, TODO, or undefined values.
- Missing parameters required to determine task success or termination.

Objective Agent Checklist

Checklist
- Is the task objective clearly stated in a single, unambiguous sentence?
- Is the success condition explicitly described?
- If multiple objectives exist, are their priorities or relative importance indicated?
- Is the objective compatible with known feasibility constraints?

Fail triggers
- Objectives that are abstract, qualitative, or not directly measurable.
- Missing or unclear success conditions that prevent reward or termination design.

Variables Agent Checklist

Checklist
- Are all state variables explicitly defined?
- Do state variables include all information required to evaluate the objective?
- Do state variables include all signals needed to enforce constraints?
- Are state variables observable or computable within the simulator?
- Is the state representation minimal and free of redundant variables?
- Are all action variables explicitly defined?
- Do action variables correspond only to controllable inputs of the robot?
- Are action dimensions, ranges, or scales clearly specified?
- Are actions consistent with the robot's actuation interface?

- Does the action space avoid unnecessary or redundant controls?

Fail triggers
- Missing state variables required to compute reward or success conditions.
- Inclusion of unobservable or undefined state signals.
- Inclusion of non-controllable quantities as actions.
- Mismatch between action definitions and the robot's control API.

Constraints Agent Checklist

Checklist
- Are all task-applicable safety, workspace, and physical feasibility constraints explicitly stated?
- Do constraints reference only previously defined state, action, or parameter variables?
- Are constraint conditions stated clearly without vague language?
- Do constraints avoid directly contradicting the task objective?
- Are constraints reasonable and not overly restrictive?

Fail triggers
- Use of undefined variables in constraints.
- Constraints that trivially prevent task completion.

Objective Modeling Agent Checklist

Checklist
- Is the objective formalized as a scalar reward or return to be optimized?
- Are all reward terms computable from defined state variables and parameters?
- Are all weights, thresholds, and coefficients explicitly specified?
- Does the reward meaningfully depend on the agent's actions?
- If the reward is sparse, is some form of shaping or intermediate guidance provided?

Fail triggers
- Presence of undefined symbols or variables in the reward formulation.
- Degenerate rewards that are constant or independent of the agent's behavior.

Constraint Modeling Agent Checklist

Checklist
- Are constraints expressed as explicit bounds or inequalities?
- Do all constraint expressions rely only on previously defined variables and parameters?
- Are constraint ranges numerically reasonable and physically plausible?
- Is it clear whether constraints are enforced as hard constraints or penalty terms?
- Do constraints preserve a non-empty feasible action space?

Fail triggers
- Contradictory or impossible constraint formulations.
- Constraints that eliminate all feasible actions.

SAR Agent Checklist

Checklist
- Are State, Action, and Reward definitions consistently integrated into a complete MDP?
- Is the reward definition consistent with the previously defined objective model?
- Are modeled constraints properly reflected in the MDP formulation?
- Are termination and success conditions explicitly specified?
- Are all previously defined parameters, variables, and constraints accounted for, with no orphan or unused components?

Fail triggers
- Missing termination or success conditions.
- Inconsistencies between SAR definitions and earlier modeling stages.

D Simulator Configuration Documents

Below we provide partial examples of the documents used by agents associated with each simulator configurator in Isaac Sim.

Scene Document

Scene config is created by inheriting from the InteractiveSceneCfg class.
This base class provides the foundational structure for defining the simulation scene.
In this config, the AssetBaseCfg, RigidObjectCfg, and ArticulationCfg classes are typically used.

AssetBaseCfg is used to create basic elements of the environment, such as the ground and lighting.
It can be used as follows:

```
object_name = AssetBaseCfg(
    prim_path=...,
    spawn=...,
    init_state=...
)
```

- `prim_path` (str): Unique path in the USD stage (e.g., `/World/light`).
- `spawn`: Configuration object defining what to spawn (e.g., `GroundPlaneCfg`, `DomeLightCfg`).
- `init_state` (`AssetBaseCfg.InitialStateCfg`): Initial position and orientation.

The following configurations can be used in the `spawn` parameter:

- **GroundPlaneCfg**: Creates a flat ground plane.
  Typically uses `prim_path=/World/ground`.
  Parameter: `size=(1000.0, 1000.0)`
- **DomeLightCfg**: Adds ambient dome lighting.
  Typically uses `prim_path=/World/light`.
  Parameters: `color`, `intensity`
- **DistantLightCfg**: Directional sunlight-like source.
  Typically uses `prim_path=/World/sun_light`.
  Parameters: `color`, `intensity`

RigidObjectCfg is used to create objects that physically interact with the robot, such as liftable or pushable objects.

```
object = RigidObjectCfg(
    prim_path=...,
    spawn=...,
    init_state=...
)
# The variable name must be 'object'
```

- `prim_path`: Typically `{ENV_REGEX_NS}/object`
- `spawn`: Shape configuration (e.g., `CuboidCfg`, `SphereCfg`)
- `init_state`: Initial position and orientation

Supported shape configurations include:

- **CylinderCfg**: `radius`, `height`
- **CuboidCfg**: `size=(x, y, z)`
- **SphereCfg**: `radius`
- **ConeCfg**: `radius`, `height`

All rigid objects must explicitly define:

- `rigid_props=sim_utils.RigidBodyPropertiesCfg()`
- `collision_props=sim_utils.CollisionPropertiesCfg()`

Optional physical properties include:

- `mass_props=sim_utils.MassPropertiesCfg(mass=...)`

- `visual_material=sim_utils.PreviewSurfaceCfg(...)`

ArticulationCfg is used to create the robot.

```
robot = ArticulationCfg(
    prim_path=...,
    spawn=...,
    init_state=...,
    actuators=...
)
```

⋮

Reward Documents

Purpose
This document provides a minimal, pattern-based reference for authoring reward functions for robotic manipulation tasks using `ManagerBasedRLEnv`.
The document intentionally avoids task names or robot identities. Reward applicability is inferred solely from:
- the presence of end-effector commands,
- object-centric observations,
- termination and success conditions.

No task-specific reward recipes are encoded.

0. Core Contract
All reward functions MUST:
- Take `env: ManagerBasedRLEnv` as input
- Return a `torch.Tensor` of shape $[N]$
- Use only the access paths and helpers listed below
- Avoid manual frame, quaternion, or contact reasoning

All tensors are already on `env.device`.

1. Minimal Access Paths
(1) Robot and object handles

$$\texttt{robot} = \texttt{env.scene["robot"]}, \qquad \texttt{object} = \texttt{env.scene["object"]}$$

(2) Joint-level signals

$$\texttt{robot.data.joint_pos} \in \mathbb{R}^{N\times J}, \qquad \texttt{robot.data.joint_vel} \in \mathbb{R}^{N\times J}$$

(3) Action signals

$$\texttt{env.action_manager.action} \in \mathbb{R}^{N\times A}$$
$$\texttt{env.action_manager.prev_action} \in \mathbb{R}^{N\times A}$$

(4) Command signals

$$\texttt{env.command_manager.get_command(command_name)}$$

2. End-Effector & Object Helpers
The following helpers encapsulate frame handling and SHOULD be used instead of manual geometry.

$$\begin{aligned}
\texttt{position_command_error} &\rightarrow \text{L2 distance} \in [N] \\
\texttt{position_command_error_tanh} &\rightarrow \text{smooth reward} \in [0,1] \\
\texttt{orientation_command_error} &\rightarrow \text{orientation error (rad)} \\
\texttt{object_ee_distance} &\rightarrow \text{EE–object distance} \\
\texttt{object_is_lifted} &\rightarrow \{0,1\} \\
\texttt{object_goal_distance} &\rightarrow \text{goal distance (conditional)}
\end{aligned}$$

⋮

4. Reward Composition Rule
Rewards are combined additively:

$$r = + w_{\text{track}} \cdot r_{\text{tracking}} + w_{\text{sparse}} \cdot r_{\text{success}} - w_{\text{act}} \cdot r_{\text{action}} - w_{\text{smooth}} \cdot r_{\text{joint}}$$

5. Non-Leakage Statement
- No reward pattern is tied to a specific task or robot
- Task identities are never used as selection signals
- Applicability is inferred only from MDP structure

⋮

7. Safety Checklist
- Reward output shape is $[N]$
- No manual frame transforms
- No raw quaternion arithmetic
- No object-specific constants without normalization
- Sparse rewards are complemented with shaping
- Zero-action baseline does not yield high reward
- Large single-step motion is penalized

If the implementation matches these patterns, it is considered valid.

ActionsCfg Document

In `ActionsCfg`, the robot's actions are defined. The class used varies depending on how the robot operates.
It can be used as follows:

$$\texttt{action_} = \texttt{ActionClass}(\ldots)$$

In the above explanation, various ActionClass can be used depending on the robot's mode of operation. The following is a list of available classes:

- `mdp.JointActionCfg`: Defines the basic configuration for mapping input actions to joint commands by applying scaling and offset. If this class is used, the required parameters are as follows:
  - `asset_name`: The name of the robot that will operate. In `SceneCfg`, it is the variable name for the robot, and it is typically `"robot"`.
  - `joint_names`: The names (or patterns) of the joints where the action will be applied. The joint names are defined in the robot section of `SceneCfg`. Within `SceneCfg`, the joint names can be determined by checking the `joint_names_expr` field in the actuators section of `ArticulationCfg`. Among these joints, you must determine which ones the command should apply to, depending on the task.
  - `scale`: A scaling factor multiplied with the input action. If multiple joints are grouped under a single name/pattern, you can use `scale` to fine-tune the command applied to each individual joint.

- `mdp.JointPositionActionCfg`: Applies the processed action as a joint position command; a default joint position can be used as an offset. If this class is used, the required parameters are as follows:
  - `asset_name`: The name of the robot that will operate. In `SceneCfg`, it is the variable name for the robot, and it is typically `"robot"`.
  - `joint_names`: The names (or patterns) of the joints where the action will be applied. The joint names are defined in the robot section of `SceneCfg`. Within `SceneCfg`, the joint names can be determined by checking the `joint_names_expr` field in the actuators section

of `ArticulationCfg`. Among these joints, you must determine which ones the command should apply to, depending on the task.
  - `scale`: A scaling factor multiplied with the input action. If multiple joints are grouped under a single name/pattern, you can use `scale` to fine-tune the command applied to each individual joint.

- `mdp.JointVelocityActionCfg`: Applies the processed action as a joint velocity command; an initial velocity can be used as an offset. If this class is used, the required parameters are as follows:
  - `asset_name`: The name of the robot that will operate. In `SceneCfg`, it is the variable name for the robot, and it is typically `"robot"`.
  - `joint_names`: The names (or patterns) of the joints where the action will be applied. The joint names are defined in the robot section of `SceneCfg`. Within `SceneCfg`, the joint names can be determined by checking the `joint_names_expr` field in the actuators section of `ArticulationCfg`. Among these joints, you must determine which ones the command should apply to, depending on the task.
  - `scale`: A scaling factor multiplied with the input action. If multiple joints are grouped under a single name/pattern, you can use `scale` to fine-tune the command applied to each individual joint.

- `mdp.JointEffortActionCfg`: Applies the processed action as a force/torque command to the joints. If this class is used, the required parameters are as follows:
  - `asset_name`: The name of the robot that will operate. In `SceneCfg`, it is the variable name for the robot, and it is typically `"robot"`.
  - `joint_names`: The names (or patterns) of the joints where the action will be applied. The joint names are defined in the robot section of `SceneCfg`. Within `SceneCfg`, the joint names can be determined by checking the `joint_names_expr` field in the actuators section of `ArticulationCfg`. Among these joints, you must determine which ones the command should apply to, depending on the task.
  - `scale`: A scaling factor multiplied with the input action. If multiple joints are grouped under a single name/pattern, you can use `scale` to fine-tune the command applied to each individual joint.

- `mdp.JointPositionToLimitsActionCfg`: Rescales the input action to match the joint's minimum–maximum range and applies it as a position command. If this class is used, the required parameters are as follows:
  - `asset_cfg`: A `SceneEntityCfg` specifying the target robot and joints (recommended usage pattern).

$$\texttt{asset_cfg} = \texttt{SceneEntityCfg}(\texttt{"robot"},\ \texttt{joint_names} = [\ldots])$$

  - `joint_names`: The names of the joints where the action will be applied. The joint names are defined in the robot section of `SceneCfg`. Within `SceneCfg`, the joint names can be determined by checking the `joint_names_expr` field in the actuators section of `ArticulationCfg`. Among these joints, you must determine which ones the command should apply to, depending on the task.
  - `scale`: A scaling factor multiplied with the input action. If multiple joints are grouped under a single name/pattern, you can use `scale` to fine-tune the command applied to each individual joint.

Additional rule for specifying `scale`. When specifying the `scale` argument, follow the task-specific rules below.

Observation Cfg Document

In ObservationsCfg, the observations that the agent can perceive are defined. Observations are used as the state in an MDP.
Typically, a PolicyCfg (ObsGroup) class is specified within ObservationsCfg, and the observations are defined inside it. This class must also have the `@configclass` tag above it.
In addition, the line

```
policy: PolicyCfg = PolicyCfg()
```

must be placed at the very end of the observation configuration class.
It can be used as follows:

```
obs_name = ObsTerm(func=..., params=..., scale=...)
```

- func: The function used to collect observation data. - params: A dictionary providing additional arguments requiredduring the function call. - scale: (float) The scale to apply to the observation after clipping.

The following observation functions can be used in the `func` parameter.
These functions belong to the category of base-related observations.
- `mdp.base_pos_z`: Returns the root height (z-axis position) of the asset.
- `mdp.base_lin_vel`: Returns the root's linear velocity in the asset's local frame.
- `mdp.base_ang_vel`: Returns the root's angular velocity in the asset's local frame.
- `mdp.projected_gravity`: Returns the gravity direction projected into the asset's root frame.

These functions belong to the category of world-frame kinematics.
- `mdp.root_pos_w`: Returns the asset's root position in the simulation world frame.
- `mdp.root_quat_w`: Returns the asset's root orientation as a quaternion in the simulation world frame.
- `mdp.root_lin_vel_w`: Returns the root's linear velocity in the world frame.
- `mdp.root_ang_vel_w`: Returns the root's angular velocity in the world frame.

These functions belong to the category of body & wrench.
- `mdp.body_incoming_wrench`: Returns external forces/torques (wrenches) applied to each body.
- `mdp.body_pose_w`: Returns the positions and orientations of specified bodies.
- `mdp.body_projected_gravity_b`: Returns the gravity direction in the body frame.

These functions belong to the category of joint-related observations.
- `mdp.joint_pos`: Returns the positions of specified joints.
- `mdp.joint_pos_rel`: Returns joint positions relative to default positions.
- `mdp.joint_pos_limit_normalized`: Returns joint positions normalized by their limit ranges.
- `mdp.joint_vel`: Returns joint velocities.
- `mdp.joint_vel_rel`: Returns joint velocities relative to default velocities.
- `mdp.joint_effort`: Returns the torque or force applied to the joints.

The following keys can be used in the params dictionary.
- `"asset_cfg"`: SceneEntityCfg, which specifies the target robot or object. The `SceneEntityCfg` class can be defined with:
  - name: Name of the robot or object defined in SceneCfg.
  - joint_names: Names of the joints (only valid if name="robot").
  - body_names: Names of the bodies required by the observation term.

⋮

Event Cfg Document

A configuration class that defines events as event terms, which are executed at specific points in the environment (e.g., at the start of an episode or during a reset). Each event specifies a function

to be executed upon reset or when a certain condition is met.

It can be used as follows:

`event_name = EventTerm(func=..., mode=..., params=...)`

- **func**: determines which event will be executed.
- **mode**: determines when the specified event will be triggered (typically `"reset"`).
- **params**: a dictionary providing additional arguments required during the function call (depends on which function is used in `func`).

The following reset functions can be used in the `func` parameter, and the dictionary should be provided in `params` accordingly.

- `mdp.reset_root_state_uniform`: Resets the root pose by adding an offset from the default root pose defined in `SceneCfg` (or the asset's default state), where the offset for each axis is sampled uniformly from the specified range.

  If this function is used in `func`, the required keys and values in `params` are:
  - `"pose_range"`: `dict[str, tuple[float, float]]`. The keys are `x`, `y`, `z`, `roll`, `pitch`, `yaw` (only necessary ones are specified). Each value is a tuple describing the range.
  - `"velocity_range"`: `dict[str, tuple[float, float]]`. Same key convention as `"pose_range"`.
  - `"asset_cfg"`: `SceneEntityCfg`. `SceneEntityCfg` can be defined with:
    - `name`: the name of robot or object defined at the last part of `prim_path` in `SceneCfg`.
    - `joint_names`: joint names (use only when `name == "robot"`).
    - `body_names`: body names required by the term.

- `mdp.reset_joints_by_offset`: Sets the initial joint positions and velocities by adding a specified offset.
  If this function is used in `func`, the required keys and values in `params` are:
  - `"position_range"`: `tuple[float, float]` (range for position initialization).
  - `"velocity_range"`: `tuple[float, float]` (range for velocity initialization).
  - `"asset_cfg"`: `SceneEntityCfg` (same convention as above).

- `mdp.reset_scene_to_default`: Resets the scene to the default state specified in the scene configuration. This function does not need `params`.

⋮

Termination Cfg Document

This is a configuration object used to define episode termination conditions.
Each condition is defined using `DoneTerm`, and it ends the episode when a specific situation occurs in the environment (e.g., the robot falls over or a goal is achieved).

It can be used as follows:

$$\texttt{termination_name} = \texttt{DoneTerm}(\text{func} = \cdot,\ \text{params} = \cdot)$$

- **func**: The function used as the termination condition.
- **params**: A dictionary providing additional arguments required during the function call. This depends on which function is used in `func`.

The following termination functions can be used in the `func` parameter, together with the corresponding dictionary entries required in `params`.

- `mdp.time_out`:
  Ends the episode if the maximum allowed time (in steps) is exceeded.
  If `func = mdp.time_out`, you must set:

$$\texttt{time_out} = \texttt{True}$$

- `mdp.root_height_below_minimum`:
  Ends the episode if the root (base) height falls below a defined minimum.
  If this function is used, the required key in `params` is:

$$\texttt{"minimum_height"} \in \mathbb{R}$$

$\vdots$

Policy Cfg Document

This section explains the components used to define the policy under which the agent operates and learns. In Isaac Lab, policies are typically defined by inheriting from `RslRlOnPolicyRunnerCfg`.

Runner configuration.
The following elements are defined directly within the runner configuration class:

- `num_steps_per_env`: Number of simulation steps collected per environment.
- `max_iterations`: Total number of training iterations.
- `save_interval`: Interval at which policy checkpoints are saved.
- `experiment_name`: Name of the experiment, used for logging and result storage.
- `empirical_normalization`: Whether to normalize observations using empirical mean and standard deviation.

Policy network definition.
The neural network architecture is defined using `policy = RslRlPpoActorCriticCfg()`, with the following parameters:

- `init_noise_std`: Initial standard deviation of the action noise in the actor.
- `actor_hidden_dims`: Hidden layer dimensions of the actor (policy) network.
- `critic_hidden_dims`: Hidden layer dimensions of the critic (value) network.
- `activation`: Nonlinear activation function used in the networks.

PPO algorithm configuration.
Training-specific parameters for PPO are defined using `algorithm = RslRlPpoAlgorithmCfg()`, with the following options:

- `value_loss_coef`: Weight applied to the value function loss.
- `use_clipped_value_loss`: Whether to apply clipping to the value loss.
- `clip_param`: Clipping range for the policy ratio.
- `entropy_coef`: Coefficient for policy entropy to encourage exploration.
- `num_learning_epochs`: Number of optimization epochs per rollout.
- `num_mini_batches`: Number of mini-batches per rollout.
- `learning_rate`: Learning rate of the optimizer.
- `schedule`: Learning rate scheduling strategy.
- `gamma`: Discount factor for future rewards.
- `lam`: Lambda parameter for Generalized Advantage Estimation (GAE).
- `desired_kl`: Target KL divergence for adaptive learning rate control.
- `max_grad_norm`: Maximum gradient norm for gradient clipping.

By defining the training parameters in this structured manner, they can be systematically modified to improve training stability and performance.

Naming requirement.
The policy runner configuration class must be named `CustomPPORunnerCfg`.

$\vdots$

Environment Document

The environment is defined by integrating all six configurations described earlier. This final configuration class represents the complete environment used for training. It loads all required sub-configurations and sets the initial environment and simulation parameters.

Initial environment and simulation settings are handled using the `__post_init__()` method. Within this method, parameters such as the physics simulation time step, rendering interval, and episode time limit are configured.

Naming requirement.
The environment configuration class must be named `CustomEnvCfg`.

Example configuration.

```
@configclass
class CustomEnvCfg(ManagerBasedRLEnvCfg):
    """Configuration for the MuJoCo-style Ant walking environment."""

    # Scene settings
    scene: MySceneCfg = MySceneCfg(
        num_envs=4096,
        env_spacing=5.0
    )

    # Basic settings
    observations: ObservationsCfg = ObservationsCfg()
    actions: ActionsCfg = ActionsCfg()

    # MDP settings
    rewards: RewardsCfg = RewardsCfg()
    terminations: TerminationsCfg = TerminationsCfg()
    events: EventCfg = EventCfg()

    def __post_init__(self):
        """Post initialization."""
        # general settings
        self.decimation = 2
        self.episode_length_s = 16.0

        # simulation settings
        self.sim.dt = 1 / 120.0
        self.sim.render_interval = self.decimation
        self.sim.physx.bounce_threshold_velocity = 0.2

        # default friction material
        self.sim.physics_material.static_friction = 1.0
        self.sim.physics_material.dynamic_friction = 1.0
        self.sim.physics_material.restitution = 0.0
```

⋮