

Isaac Sim, Isaac Lab, and MuJoCo: A Complete Guide to Robot Simulation and Reinforcement Learning

2026-10-19

Contents

1. Platform Roles and Research-Task Selection	2
2. Versions, Hardware, and Platform Orientation	3
3. Robot Assets: URDF, MJCF, and USD	4
4. Dynamics, Control, Contact, and Sensors	5
5. The Engineering Contract of an RL Environment	6
6. Parallel Learning Workflows in Isaac Lab	7
7. MuJoCo, MJX, and MJWarp Workflows	8
8. Training Throughput, Memory, and Debugging	8
9. Randomization Implementation, Evaluation, and Reproducibility	9
10. Deployment Interfaces, Calibration, and Sim-to-Sim	10
12. Troubleshooting and Readiness Checks	11
Official References and Further Reading	11

[Course home](#) · [简体中文](#) · [Motion-control RL](#)

Item	Details
Scope	Engineering tools and training infrastructure (How to Use); not a duplicate treatment of PPO, SAC, or control theory
Version scope	Isaac Lab 2.3.2 / Isaac Sim 5.1; Isaac Lab 3.0 Early Access / Isaac Sim 6.1; MuJoCo/MJX/MJWarp versions depend on workload
Documentation review	2026-10-19
License	CC BY-NC-SA 4.0 · Xiong Zeng and contributors

1. Platform Roles and Research-Task Selection

Boundary with the companion course: This guide asks how to build a reliable, efficient, reproducible simulation/training platform; the companion motion-control RL guide asks how to learn tracking, dynamic skills, and recovery.

Course knowledge map: Robot model → simulation scene → environment contract → parallel training → deployment validation.

1.1 Their roles in the robotics workflow

- **Isaac Sim** is NVIDIA’s robotics simulation platform for scene authoring, USD assets, PhysX simulation, RTX rendering, camera simulation, and ROS 2 integration. Its primary concern is *how a robot and its environment operate and are perceived in simulation*.
- **Isaac Lab** is a robot learning framework built on simulation backends. It provides task definitions, vectorized environments, observations, actions, rewards, resets, termination rules, domain randomization, and interfaces to RL libraries. Its primary concern is *how to train and evaluate a policy systematically*.
- **MuJoCo** is a dynamics and contact simulation engine useful for rigid-body mechanics, constraints, actuators, control, and model debugging. Traditional MuJoCo is primarily CPU-based; MJX and MuJoCo Warp (MJWarp) enable accelerated batched simulation.

Important recent development: Isaac Lab 3.0 Early Access introduces a multiple-physics-backend architecture, including Newton/MuJoCo-Warp-based execution paths. Accordingly, “Isaac Lab always means PhysX” and “MuJoCo always means CPU simulation” are no longer generally accurate. Backend feature parity and API compatibility must still be checked for each release. [1–4]

Dimension	Isaac Sim	Isaac Lab	MuJoCo and GPU ecosystem
Primary focus	Scenes, sensing, digital twins	Robot learning and batched tasks	Dynamics, contact, control, batching
Common assets	USD; URDF import	Backend-supported assets	MJCF; URDF import
Visualization	Strong RTX rendering	Depends on the selected renderer	Lightweight viewer; backend-dependent batched rendering
RL integration	Requires a learning/task layer	RSL-RL, RL-Games, skrl, etc.	MuJoCo Playground, Gymnasium, custom stacks
Best first use	Scene and sensor inspection	Vectorized reinforcement learning	Dynamics and control
Main pitfalls	GPU drivers, VRAM, RTX support	Version/API changes, rewards and resets	Actuation, contacts, CPU/GPU discrepancies

1.2 Which tool should you select?

Research goal	Suggested starting point	Why
Dynamics, PD/MPC, inverse dynamics	MuJoCo	Direct access to physical behavior and straightforward single-step debugging
Replay-buffer-based SAC	A compatible MuJoCo RL stack or supported Isaac Lab integration	Confirm that SAC and vectorized replay are actually implemented
Head and wrist RGB-D, lighting and occlusions	Isaac Sim / Isaac Lab	Integrated rendering and camera modeling

Research goal	Suggested starting point	Why
High-throughput state-based contact simulation	Suitable Isaac Lab backend or MJX/MJWarp	Depends on scene complexity, hardware, and data movement
Pre-deployment robustness checks	Both physics engines	Exposes dependence on one solver or actuator approximation

Selection rule: Decide on the physical task and observations before optimizing raw simulation FPS. Photorealism, contact fidelity, and real-world compatibility may be more consequential than peak steps per second.

2. Versions, Hardware, and Platform Orientation

2.1 Two sensible version tracks

Stable reproduction track: Isaac Lab 2.3.2 with Isaac Sim 5.1. This is a practical option for replicating existing 2.x research code, RSL-RL pipelines, and established PhysX manipulation tasks. Pin the Git tag, Python, PyTorch, extensions, and GPU driver rather than installing moving targets.

New architecture track: Isaac Lab 3.0 Early Access with Isaac Sim 6.1 when full Isaac Sim integration is needed. Version 3.0 adds selected kit-less workflows and a multi-backend architecture, but includes breaking changes. As of this guide's date it is still Early Access; older tasks and extensions should not be assumed compatible. A particularly important migration detail is the change from the common `wxyz` quaternion convention in 2.x to `xyzw` in 3.0, alongside data interface changes. [1][2]

Recommended setup sequence: (1) Identify the paper or feature to reproduce; (2) consult the relevant release compatibility table; (3) keep Isaac and MuJoCo dependencies isolated; (4) validate an official minimal scene; (5) check simulation stepping, asset import, and GPU use; and only then (6) connect the RL trainer. Avoid installing unrelated machine-learning packages before the simulator baseline works.

2.2 Where should the compute run?

- **RTX 4090, 24 GB:** Well suited to GPU physics training and carefully budgeted vision-based experiments. High-resolution multi-camera tasks still need active memory management.
- **RTX 3090, 24 GB:** Useful for compatible experiments and MuJoCo GPU training; however, Isaac Sim 5.1 lists an RTX 4080 as its minimum GPU performance tier. Treat a 3090-based Isaac Sim setup as a configuration requiring its own verification.
- **A100/A800:** Excellent for many neural-network workloads and compute paths that do not need RTX rendering. **The A100 is not supported for the full RTX-dependent Isaac Sim 5.1 experience.** High tensor throughput does not imply RTX renderer compatibility.
- **CPU machines:** Often appropriate for classic MuJoCo model validation, MPC, single-robot control, and smaller rollouts. Move to MJX/MJWarp when a measured batch-throughput need justifies it.

Official Isaac Sim 5.1 requirements specify supported Ubuntu configurations, GPU classes, memory, storage, and driver expectations. Updating every dependency to “latest” can easily break an otherwise working simulator stack. [5]

2.3 Acceptance checks: launching is not enough

Check	Acceptance criterion
Basic scene	Ground, gravity, and robot load; no unintended penetration
Joint motion	Directions, limits, home positions, and velocities verified joint by joint
Robot state	Poses, velocities, and forces agree with the model definitions

Check	Acceptance criterion
Rendering and cameras	Correct orientation, depth interpretation, timestamps, and resolution
Physics stability	No persistent resting oscillation; plausible free fall
RL connectivity	Observation-action-step-reset-reward loop operates correctly
Reproducibility	Software versions, asset hashes, GPU, driver, and random seeds recorded

2.4 A practical MuJoCo learning sequence

1. **Understand the model:** A robot is built from bodies, joints, geoms, actuators, sensors, and constraints. Its rendered geometry and collision geometry can be distinct.
2. **Inspect a robot:** Check joint axes, home positions, collision shapes, inertia, and tool reference frames in the viewer.
3. **Test passive motion:** Observe a pendulum or arm moving under gravity; compare physical intuition with simulated behavior.
4. **Test actuation:** Examine position, velocity, and torque control separately. An actuator's command is not necessarily an applied joint torque.
5. **Introduce contacts:** Add a tabletop and a small object. Study free fall, sliding, grasping, and release.
6. **Prepare for RL:** Specify observations, actions, resets, and randomization before integrating a trainer.

Recommended first complete robot: Franka Panda. It is sufficiently complex for useful manipulation while remaining easy to inspect; start from a maintained model in MuJoCo Menagerie. [6]

2.5 An Isaac Sim / Isaac Lab learning sequence

1. **Scene authoring in Isaac Sim:** Learn the Stage, Prims, USD assets, viewport, timeline, physical materials, and rigid-body properties.
2. **Import a robot:** Inspect the articulation root, drives, masses, joint limits, and collision geometry after URDF import.
3. **Build a simple scene:** Ground, table, fixed-base arm, manipulated object, and lighting.
4. **Validate sensors:** Place head and wrist cameras; verify intrinsics, extrinsics, depth semantics, occlusions, and synchronization.
5. **Study the Isaac Lab task abstraction:** Scene, robot assets, observations, actions, rewards, terminations, and events/domain randomization.
6. **Try maintained benchmark tasks:** Cartpole, Franka Reach, Franka Lift.
7. **Add RL gradually:** Start with state-based policies and limited randomization before adding vision and difficult contacts.

Manager-based vs. Direct workflows: Manager-based tasks emphasize reuse of observation, reward, reset, and event components. Direct tasks offer more explicit control for unusual contact phases or research algorithms. Both support reinforcement learning; the difference is software organization rather than inherent learning power. [7]

2.6 A shared exercise: Panda, cube, and table

Do not begin by comparing two entirely different robots in different engines. Fix the robot, cube dimensions, table height, tool frame, initial pose distribution, and action definition. Progress through **single-joint response** → **end-effector tracking** → **cube grasping** → **perturbed regrasping**.

3. Robot Assets: URDF, MJCF, and USD

3.1 There is no universally lossless conversion

Modeling aspect	URDF	MJCF	USD / Isaac
Links, joints, inertia	Standard capabilities	Supported	Supported
Visual and collision meshes	Supported	Rich contact modeling	Strong scene and rendering support
Motors and actuation	Limited in core standard	Actuators, tendons, transmissions	Joint drives and physics properties
Advanced contact behavior	Usually engine-specific extensions	Detailed contact parameters	PhysX contacts and materials
Scene composition	Relatively basic	Physics-centric	Scalable scene composition

Maintain a **canonical physical parameter sheet** covering joint names, axis orientations, zero positions, limits, link masses, centers of mass, inertias, TCP, collision meshes, actuation modes, and sensor extrinsics. This provides better cross-engine consistency than an URDF file alone.

3.2 Ten checks after import

1. Base and world frames; 2. joint names and ordering; 3. positive rotation directions; 4. angle and length units; 5. tool-center-point definition; 6. mass and center of mass; 7. physically plausible inertias; 8. collision simplification; 9. appropriate self-collision filtering; and 10. actuation limits, stiffness, damping, and control period.

4. Dynamics, Control, Contact, and Sensors

4.1 A common mechanics perspective

For rigid-body robots, the essential balance is:

Inertial effects + Coriolis/centrifugal/gravity-related bias terms = generalized actuator forces + generalized contact/constraint forces.

MuJoCo and PhysX use different numerical implementations. Before attributing discrepancies to the engine, first make robot parameters, actuation, and contact scenarios as equivalent as possible.

4.2 Five common control interfaces

Control style	Policy produces	Main caveats
Joint position targets	Desired joint angles	Hidden PD gains, torque saturation, velocity limits
Joint position increments	Changes relative to a target or state	Accumulated drift, scaling, temporal smoothness
Joint velocity	Desired joint velocity	Stability near contact, integration error
Joint torque	Actuator effort	Gravity compensation, friction, saturation, hardware safety
Cartesian pose or impedance	TCP pose or impedance quantities	IK feasibility, singularities, frame conventions

Crucial distinction: A policy that *commands* a joint position does not instantaneously *set* the simulated joint to that position. The position target must go through a comparable controller and actuator. Directly teleporting joints can bypass physical constraints and create a major sim-to-real mismatch.

4.3 Keep three clocks separate

- **Physics step:** Integrates motion, collision, and contact. Smaller steps often improve stability at higher cost.
- **Low-level control frequency:** Updates servo, impedance, or IK targets.
- **Policy and sensor frequencies:** RL decisions and camera acquisitions; these rates may be asynchronous.

Example: 500 Hz physics and 50 Hz policy inference imply **10 physics steps per policy action**. A 30 Hz camera requires explicit frame holding, timestamps, and delay handling. The deployed system additionally includes communication, estimation, filtering, and actuator response latency.

4.4 Fair control comparisons

Hold four things constant: **action meaning in joint or task space, action scaling, low-level controller semantics, and effective sampling period**. Without those controls, measured differences cannot be cleanly attributed to the physics engine.

5. The Engineering Contract of an RL Environment

This section specifies an **engineering contract for simulation interfaces** rather than repeating reward-design theory. For mathematical control objectives, see the companion motion-control RL tutorial.

Module	Isaac Lab interface	MuJoCo interface	Validation contract
Scene and batch	Scene / Articulation / Manager-based or Direct workflows	MjModel / MjData, vectorized model/data	No state leakage across instances; versioned assets
Observation	Observation Manager or custom buffers	Sensors, state extraction and wrappers	Shapes, units, frames, timestamps, deployability
Action	Action Manager or custom control path	ctrl and actuator interface	Limits, semantics, update rate
Reward/termination	Reward/Termination Manager or Direct callback	Task callbacks	Explicit success tests, separate time limits, logged components
Reset and randomization	Event/Reset Manager or Direct callbacks	Vectorized reset/parameter updates	Seed control, no stale hidden state
Learning adapter	Supported RSL-RL, RL-Games, skrl integrations	Playground/Brax/other adapters	Rollout, device, checkpoint and normalizer consistency

5.1 Minimal interface acceptance tests

- **Reset consistency:** identical assets, initial conditions and seeds should permit meaningful reruns; document non-determinism in CUDA and asynchronous pipelines.
- **Clock integrity:** record physics steps, actuator updates, policy actions, and sensor timestamps separately. Physics FPS is not policy experience throughput.
- **Numeric contract:** verify action scaling, joint limits, reward bounds, NaN/Inf handling, terminal signals and time limits.
- **Privileged-state isolation:** information available only in simulation may be used by a critic or teacher, never accidentally leaked to the deployed actor.
- **Traceable dependencies:** persist model hashes, backend choices, sensor settings, trainer configuration, normalization statistics and policy checkpoints.

5.2 PPO/SAC: selection at the tooling layer only

PPO commonly uses synchronized, high-throughput rollouts. SAC requires replay buffers, off-policy updates, and reliable handling of parallel sampling. The **engineering** question here is whether the selected trainer supports the algorithm and the environment correctly; the algorithmic derivation belongs in the dedicated PPO/SAC tutorials.

6. Parallel Learning Workflows in Isaac Lab

6.1 From a benchmark to a custom manipulation task

Step 1 — Pick a starting task. Cartpole demonstrates the end-to-end RL loop; Franka Reach checks controls and observations; Lift adds contacts.

Step 2 — Fix the backend and release. Established 2.x PhysX tasks and new 3.0 multi-backend workflows are not automatically interchangeable. Reproduce one supported baseline first.

Step 3 — Specify the scene. Record the robot, table, object, drive settings, physics material, initial configurations, and camera geometry.

Step 4 — Define five environment interfaces. Observations, actions, rewards, termination, and reset; make sure state and cached sensor history are synchronized after every reset.

Step 5 — Overfit a single easy configuration. Use fixed objects and goals, with randomization disabled. If the policy cannot solve that configuration, inspect the environment before scaling the neural network.

Step 6 — Grow vectorization carefully. Begin with a manageable batch, then measure environment steps per second, GPU utilization, VRAM, and contact stability. More environments do not guarantee faster end-to-end learning.

Step 7 — Establish a PPO baseline. Freeze seeds, rollout horizon, optimizer settings, architecture, reward scales, and control period. Change one factor family at a time.

Step 8 — Evaluate independently. Use a fixed evaluation protocol, archive trajectories and failure videos, and measure success and safety rather than training return alone.

Step 9 — Introduce realistic variation. Add dynamics uncertainty, actuator delay, observation noise, visual occlusions, and disturbances one at a time.

Step 10 — Archive a reproducibility bundle. Preserve model assets, full task configuration, reward weights, normalization statistics, trainer version, checkpoints, evaluation implementation, and logs.

6.2 Selecting the Isaac Lab RL library

Training library	Good starting use	Caveats
RSL-RL	GPU-parallel PPO, state control, distillation	Verify actor/critic observation groups and architecture
RL-Games	Existing RL-Games projects or its supported algorithms	Capabilities and task schemas vary across releases
skrl	Wider algorithm range or PyTorch/JAX options	Needs careful integration and reproducibility checks
Stable-Baselines3	Smaller baselines and familiar analysis ecosystem	Not necessarily optimal for large GPU-resident batches

6.3 Common Isaac Lab RL pitfalls

VRAM: Memory can grow rapidly with camera count $\times$ resolution $\times$ environment count; benchmark again after adding vision. **Headless mode:** No interactive viewport does not imply that all offscreen rendering is disabled, or that camera-based RL works without configuring camera outputs. **Reset correctness:** Individual environments terminate asynchronously; reset indices and observation history must be updated correctly. **Reward timing:** Changing physics rate or policy decimation can unintentionally change accumulated reward magnitudes if time weighting is inconsistent.

7. MuJoCo, MJX, and MJWarp Workflows

7.1 Classic MuJoCo: correctness first

Classic MuJoCo is well suited to isolated control tests, dynamics validation, small Gymnasium tasks, and CPU rollouts. It offers direct debugging and low single-step latency. In very large GPU-training workflows, repeated CPU-to-GPU movement of observations may become a bottleneck. [3]

Use classic MuJoCo to validate the robot model, reward, reset, contact behavior, and low-level control before deciding whether the environment needs a GPU-batched implementation.

7.2 MJX and MJWarp: throughput-oriented training

MJX-JAX uses JAX/XLA for batched computation. **MJWarp** targets NVIDIA GPU parallel simulation, especially many independent robot environments. Higher *aggregate throughput* does not imply lower *single-step latency*: MJWarp focuses on throughput, while classic MuJoCo is often preferable for low-latency single-system control. The current MJX-Warp route does not target automatic differentiation. [3][4]

MuJoCo Playground provides locomotion, some vision-enabled tasks, and RL baselines built around MJX/MJWarp. It is a useful counterpart to Isaac Lab for evaluating state-control tasks and cross-engine robustness. [10]

7.3 Ten steps for a MuJoCo RL project

1. Choose a maintained Menagerie or validated MJCF robot model;
2. align the joints and actuators with hardware;
3. build a minimal scene;
4. validate free dynamics and contact;
5. define observations, actions, and normalization;
6. design rewards, resets, and terminal rules;
7. run a small deterministic debug experiment;
8. choose CPU, MJX-JAX, or MJWarp;
9. connect PPO or another supported learner; and
10. evaluate held-out initial states, physical parameters, and other simulation engines.

Backend feature check: GPU implementations need not have full one-to-one support for every solver feature. Revalidate tendons, unusual constraints, contact capacities, sensors, and batched resets. A CPU MuJoCo model should not automatically be assumed to produce identical results after migration to GPU simulation.

8. Training Throughput, Memory, and Debugging

8.1 Measure at least five quantities

Track **environment steps per second**, **aggregate physics steps per second**, **optimizer time**, **peak VRAM**, and **wall-clock time to a fixed success threshold**. For vision-based RL, also track rendering time, encoder inference, and inter-device transfers.

FPS/SPS reported by different projects may refer to one world's physics frames, aggregate vectorized steps, or policy steps. Always report environment count, physics/control frequencies, GPU model, and camera configuration alongside throughput.

8.2 More environments are not always better

Too few worlds underutilize the GPU; too many exhaust VRAM or stress contact solving, rendering, transfers, and optimization. Increase batch size incrementally and select the operating point using measured **success improvement per hour**, not an arbitrary goal of thousands of environments.

8.3 Two routes to visual RL

- **End-to-end visual RL:** Train directly from images. Conceptually simple but more expensive and sensitive to occlusion and appearance mismatch.
- **Privileged-state teacher to visual student:** Learn competent control before introducing perception; separates “the task is mechanically hard” from “the image is ambiguous.”

When teacher and student update actions at different rates, distillation must align timestamps and action meanings, not merely array dimensions.

8.4 Debugging priorities before optimizer tuning

When learning fails, check data contracts first: observation shape and normalization, action scale and clipping, reward/termination logs, batch definitions after changing parallelism, and the physical meaning of one episode step. PPO KL/clip fraction/value loss or SAC Q-values/temperature/replay age are **diagnostic signals**, not a reason to turn this tools course into an algorithm derivation.

For image-based RL, report image sizes, camera count, rendering throughput, encoder latency, peak VRAM and queuing delays from rendering to action execution. A genuine speedup reduces wall-clock time without sacrificing performance under an identical evaluation protocol.

9. Randomization Implementation, Evaluation, and Reproducibility

Category	Example randomized factors	Intended generalization
Geometry	Initial object pose, table height, small size errors	Workspace and layout variation
Dynamics	Mass, inertia, friction, joint damping	Modeling uncertainty
Actuation	Delay, deadband, stiffness, saturation, filtering	Real servo behavior
Observation	Encoder noise, pose bias, dropped frames	Sensing and synchronization robustness
Vision	Lighting, material, background, occlusion, camera extrinsics	Visual domain shift
Disturbances	Pushes, slip, drops, moving targets	Recovery and corrective behavior

Recommended sequence: First solve the task without randomization; then expand initial states; then dynamics and actuator uncertainty; followed by state-estimation errors, visual randomization, and finally externally induced drops/recovery. Set ranges using measured robot uncertainty or physically defensible identification intervals, not arbitrarily extreme values.

Keep training and evaluation distributions distinct. Use held-out combinations of poses, friction, mass, lighting, and calibration offsets to distinguish true robustness from interpolation within the training set.

9.1 Core metrics

Metric	What it measures
Success rate	Fraction of episodes meeting objective completion criteria
Time-to-success	Steps or elapsed real time until task completion
Drop/slip rate	Frequency of losing or slipping the object
Recovery success	Success after a recoverable perturbation
Pose/tracking error	Object or end-effector trajectory accuracy
Unsafe-event rate	Collisions, excessive loads, workspace violations
Environment steps / wall-clock	Sample efficiency and total training efficiency
Real-time latency	End-to-end perception, inference, and control delay

Use multiple random seeds, report central estimates and uncertainty, and avoid cherry-picking one unusually strong checkpoint. For algorithm comparisons, align assets, observations, action interfaces, training budgets, and evaluation distributions.

9.2 Avoid unsupported conclusions

- **Cross-engine comparison is not exact mechanics equivalence.** Report friction and contact-model differences.
- **Equal success rate does not imply equal robustness.** Break down failures by grasp, reorientation, release, and recovery phase.
- **Training frames are not automatically interaction samples.** Distinguish physics steps, policy steps, and vectorized totals.
- **Checkpoint weights are not the entire policy.** Retain normalization statistics, action scaling, and software dependencies.

10. Deployment Interfaces, Calibration, and Sim-to-Sim

10.1 Six calibration layers

1. **Geometry:** Joint zero positions, TCP, robot base relative to the table.
2. **Dynamics:** End-effector payload, center of mass, inertia, friction, and gravity compensation.
3. **Actuation:** Command mode, position/velocity/torque limits, and response delay.
4. **Vision:** Head/wrist camera intrinsics and extrinsics, image timestamps, depth units.
5. **Software integration:** ROS 2 or vendor SDK timing, buffering, packet loss, and error recovery.
6. **Safety:** Emergency stops, workspace restrictions, collision thresholds, current limits, and dropped-object handling.

10.2 A deployment validation ladder

Unloaded joint motion → contact-free tracking → slow approach → low-force grasp → object lift → object rotation → small supervised disturbance → controlled drop-and-regrasp. Apply limits, protection, and human supervision before advancing to each stage.

If hardware results lag behind simulation: First inspect frames and synchronization, then action meaning and servo period, then contacts and mechanical parameters. Only afterward should you consider a larger neural network or more complex RL hyperparameter search.

12. Troubleshooting and Readiness Checks

Symptom	Inspect first	Avoid the reflex to...
Robot jitters after reset	Joint zeros, stiffness, collisions, physics step	Raise gains immediately
Cube penetrates surfaces	Collision meshes, contacts, step size	Only increase friction
PPO reward rises but success stays low	Reward terms, success metric, rollout video	Just train longer
SAC critic diverges	Replay distribution, critic, target update, reward scales	Only enlarge the network
GPU remains underutilized	CPU/GPU transfers, rendering, batch size, sync points	Maximize environment count blindly
Visual training is extremely slow	Camera count, resolution, renderer, encoder	Start with full-resolution cameras
Hardware grasp is too weak	Contact geometry, materials, servo behavior, model error	Raise simulated friction only
Policy cannot recover after a drop	Termination logic, RSI, visual coverage, objective	Increase success bonuses alone
Arm tracking error is high	IK, TCP, delays, scaling, velocity limits	Retune PPO first
Behavior changes after Isaac update	Pinned dependencies, quaternion order, sensor/backend changes	Trust old checkpoints without revalidation

Global diagnostic order: Model integrity → frames and units → actuator control → physics/contact → observations and delay → reward/reset design → optimization algorithm.

Model and physics: Verified joint order, axes, limits, masses, inertias, TCP, collisions, actuators, physics step, and friction.

RL environment: Explicit success criterion, no observation leakage, bounded actions, interpretable reward terms, correct timeout semantics, and reset coverage near key events.

Training: Fixed-start baseline solved; seeds, trainer, task, and model releases recorded; interaction count, throughput, wall time, and peak VRAM reported.

Evaluation: Multiple seeds, held-out initial states, OOD disturbances, cross-engine tests, failure categories, and drop recovery documented.

Deployment: Frames and cameras calibrated; rates and delays matched; joint/force/velocity limits, supervision, and emergency stops prepared.

One-sentence takeaway: MuJoCo clarifies the mechanics and actuation; Isaac Sim provides rich scenes and realistic sensing; Isaac Lab or MuJoCo Playground structures RL training; cross-engine, out-of-distribution, and real-hardware evaluation determine whether the policy is genuinely robust.

Official References and Further Reading

1. [Isaac Lab Releases and compatibility](#)
2. [Isaac Lab 3.0 EA documentation](#)
3. [MuJoCo Warp documentation](#)
4. [MuJoCo XLA](#)
5. [Isaac Sim 5.1 requirements](#)
6. [MuJoCo Menagerie](#)
7. [Isaac Lab 2.3.2 Direct RL Environment](#)
8. [Isaac Lab sensors](#)
9. [Isaac Lab RL integrations](#)

10. MuJoCo Playground
11. MuJoCo official documentation
12. Isaac Sim official documentation
13. MuJoCo MJX implementations and differentiability
14. Isaac Lab 3.0 Early Access release notes

Companion guide: Motion tracking, Residual RL, Teacher–Student methods and recovery are covered in *Reinforcement Learning for Motion Control in Embodied AI*.