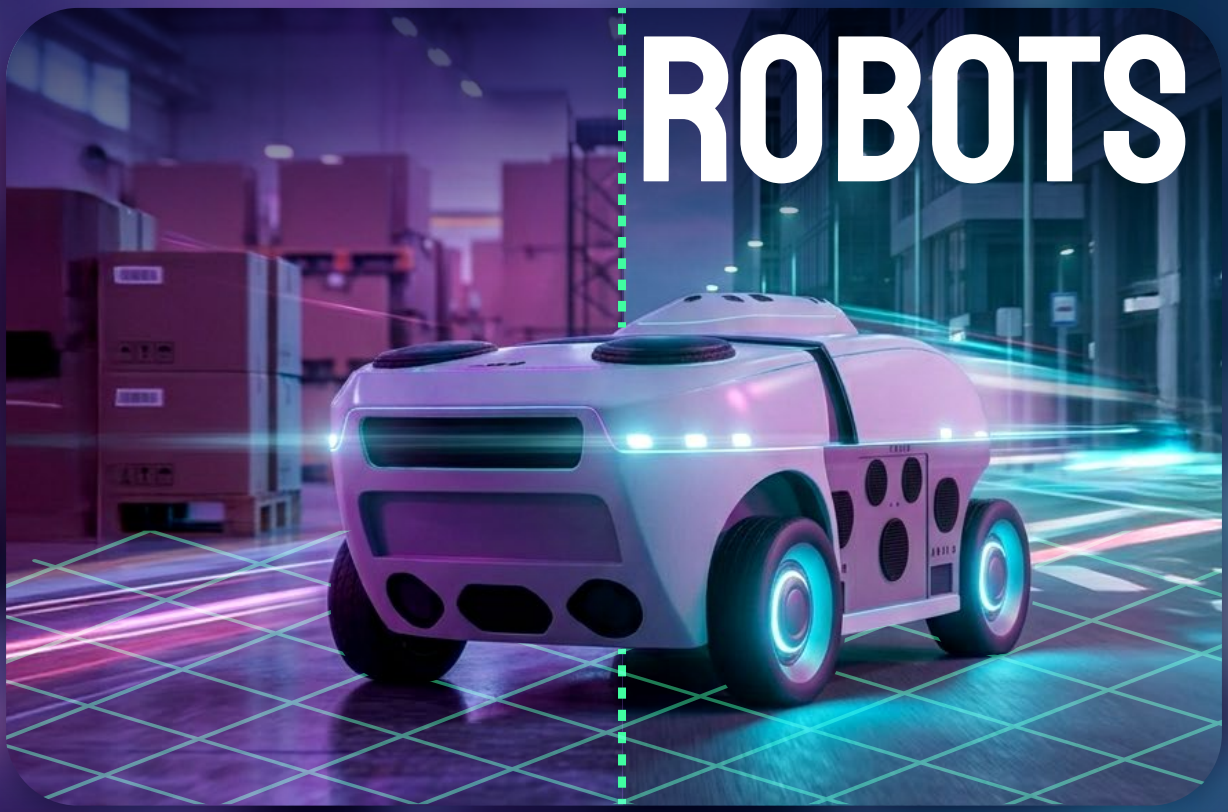
EKUMEN
A GRID DYNAMICS COMPANY

NVIDIA

SIMULATING → REAL-WORLD ROBOTS



with NVIDIA® Isaac Robotics Development Platform

EXPERT GUIDANCE FROM
INDOOR AND OUTDOOR
DEPLOYMENTS

This guide covers what we've learned deploying robots using NVIDIA's simulation and edge inference tools across real indoor and outdoor programs. It includes an honest assessment of each tool's role and decision criteria for choosing which parts of the NVIDIA stack to adopt, as well as practical warnings about the gaps between simulation and reality.

AUTHOR: Christian Barceló

Senior Software Developer at Ekumen, a Grid Dynamics Company.

EKUMEN
A GRID DYNAMICS COMPANY

NVIDIA

Table of contents

Executive Summary & Who Should Read This

NVIDIA Isaac is an open robotics development platform consisting of simulation and robot learning frameworks, CUDA-accelerated libraries, AI models, and reference workflows to create autonomous mobile robots (AMRs), robot arms and manipulators, and humanoids. It offers unprecedented capabilities for simulation, reinforcement learning, and synthetic data generation. However, without a strategic approach, teams quickly find themselves bottlenecked.

Many teams come to us facing the same expensive and frustrating hurdles: struggling with poor Real-Time Factor (RTF), getting slowed down by deprecated or poorly configured sensors, having a bad strategy to grow their simulation - trying to test the whole stack at once from the get-go instead of having an iterative approach mixing testing the stack and using ground truth data, while targeting full-featured simulation in the long run. Others get bogged down by brittle communication layers, or relying too heavily on OmniGraphs and unmanaged USDs rather than building robust, orchestratable Python standalone workflows. In the realm of perception, we frequently see teams either abandoning synthetic data altogether, generating overly simplistic scenarios that fail to capture real-world edge cases, or jumping straight into Cosmos before building a solid foundation with Omniverse Replicator.

This eBook is written for **Robotics Tech Leads**, **Perception Engineers**, and **CTOs** navigating this complex ecosystem.

Have you ever wondered:

- Will migrating to Isaac Sim actually deliver a high ROI for our pipeline?
- Does Isaac Sim fit our specific use case and hardware constraints?
- Is Isaac Lab the right framework for training our RL models?
- Can synthetic data reliably improve our real-world perception models?

The critical insight from our deployments: A simulation that hasn't been validated against real sensor data and real deployment conditions is a hypothesis, not evidence. At the same time, neglecting simulation altogether inevitably leads to higher costs and compounded problems down the road.

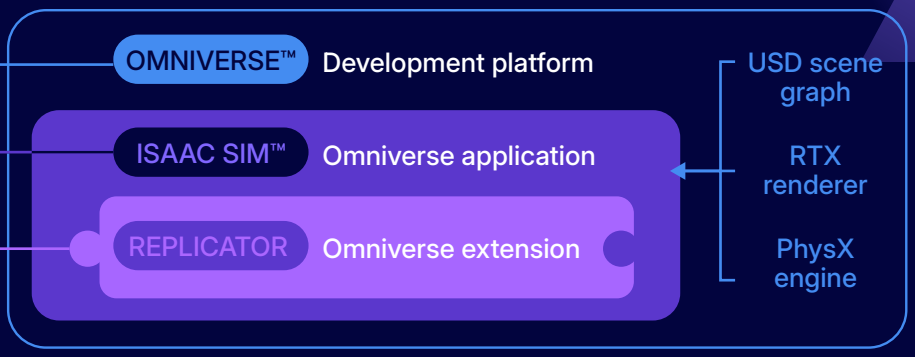
The NVIDIA Robotics Platform at a Glance

A common mistake is treating each NVIDIA robotics platform as a standalone tool. They're interconnected, and understanding the relationships between them is the first step toward using them well.

*not a separate tool
Omniverse is there from the moment you start working with Isaac Sim

*built on its Kit SDK

*typically used within Isaac Sim in robotics workflows



Next, we'll highlight some key components of the platform

ISAAC SIM™

The core simulation environment. Robot models (URDF, MJCF), sensor simulation (cameras, LiDAR, IMU, contact sensors), physics via PhysX (and more recently, Newton), and a ROS 2 bridge. This is where teams spend most of their time and where most integration decisions are made.

In practice, work in Isaac Sim follows two phases. The first is interactive: importing robot models, tuning physics and sensor parameters, and iterating on environment setup through the GUI with live feedback, changes take effect immediately without restarting the simulation, which contrasts with the edit-file, launch, inspect, close cycle that's still common in other simulators. The second phase is programmatic: building automated pipelines that open test scenarios, spawn configured robots and obstacles, set up environments, and run tasks repeatedly. This is where Isaac Sim becomes part of a Software-in-the-Loop (SITL) pipeline, with scene setup, test execution, and data collection all driven through code.

ISAAC LAB

An open-source, GPU-accelerated, agent-ready, simulation framework for robot learning designed to train robot policies at scale. Provides massively parallel environment instances, reward function definition, domain randomization, and policy evaluation. Designed for training learned controllers such as grasping and locomotion policies. Isaac Lab's parallel training infrastructure and flexible integration across physics engines, renderers, and learning algorithms, including Newton and Omniverse, accelerates vision and perception training for real-world robot applications. If your robot relies on classical algorithms, Isaac Sim alone covers your simulation needs without the added complexity.

OMNIVERSE REPLICATOR

An NVIDIA Omniverse™ extension for generating labeled synthetic datasets. Randomizes visual properties (lighting, textures, object poses, camera angles, animation states and actor poses) and outputs a wide range of ground truth data including RGB images, depth, segmentation masks, bounding boxes, normals, and camera and object poses. It can be used both for capturing static randomized snapshots and for generating sequences of data driven by physics interactions. It is focused on perception model training.

COSMOS™

NVIDIA's family of world foundation models for physical AI. The family includes several models with different roles:

Cosmos Predict generates plausible future video frames from past observations.

Cosmos Transfer transforms real or synthetic data by modifying visual properties based on a text prompt. It works best when the transformation doesn't involve significant changes to object shapes or positions; for example: changing weather conditions in a park, adjusting warehouse lighting, or making low-fidelity actor models look like real people. In these cases, the original ground truth annotations remain usable. More drastic transformations (turning cubes into people, reshaping objects) risk label drift or visual artifacts depending on how the model weights input data against the prompt, and may require re-segmentation with other tools.

Cosmos Reasoning is a vision-language-action model for physical reasoning.

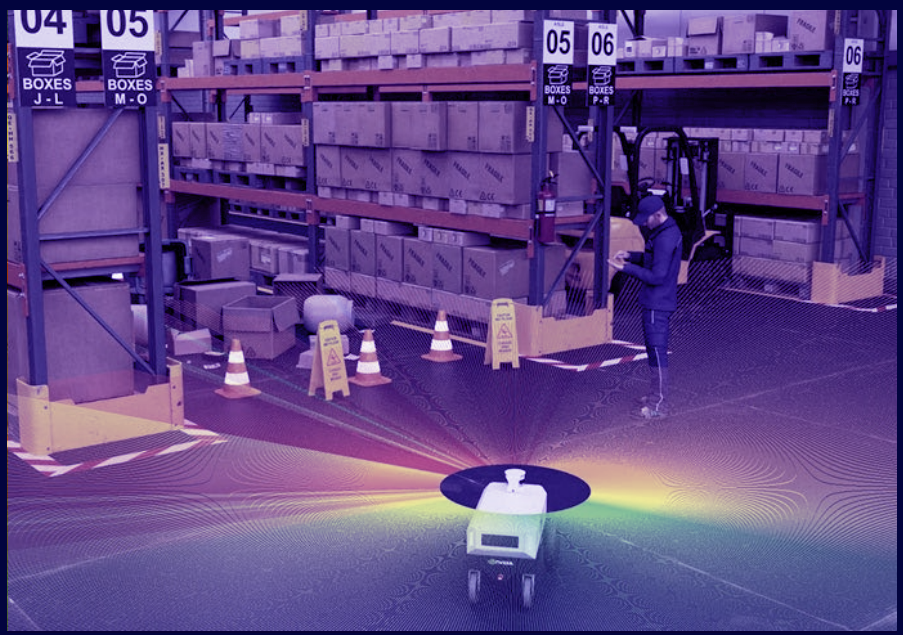
Cosmos Curator is a service that can segment videos of various lengths into semantically consistent clips, generate embedding data, and create text prompts for each video.

COSMOS™

All Cosmos models carry significant compute costs. In our experience, for full synthetically generated data, their adoption makes the most sense once traditional simulation or Replicator pipelines are already established and producing results. Cosmos Transfer may add the most value as an enhancement step on top of existing synthetic data, not as a replacement for it. While this doesn't fully apply to expanding existing real data, the amount of data you can generate using either this method or Replicator must be considered.

NVIDIA ISAAC™ ROS

A set of GPU-accelerated ROS 2 packages: visual SLAM, nvblox for 3D reconstruction, DNN inference, freespace segmentation. These are standard ROS 2 nodes that work with any ROS 2 data source, real robots, Isaac Sim via the Omniverse ROS 2 Bridge extension, or other ROS 2-compatible simulators. Runs on both workstations (for development) and Jetson (for deployment). In practice, using these packages against Isaac Sim during development and then against real hardware during deployment keeps the software stack consistent across both stages.



Choosing What to Use

The criteria to select which parts of the stack to adopt depend entirely on the scope of your project. Not every program needs every component. Below are the key questions we use to guide that decision.

Project scope

Is your robot using learned control policies (RL)?

→ **If yes**, Isaac Lab may be the right tool for parallel policy training.

→ **If your robot uses classical planners** (Nav2, MoveIt), you likely don't need Isaac Lab. Isaac Sim alone covers simulation of the robot and environment.

Do you need synthetic data for perception model training?

→ **If yes**, Omniverse Replicator generates labeled datasets (bounding boxes, segmentation, depth) with visual randomization. This is its core strength.

- A well-balanced mix of synthetic and real data (for example, 70/30) can outperform models trained on real data alone, primarily because of edge case coverage that's expensive to stage in real environments and the ability to generate highly variable data at scale.

→ **Otherwise**, you may still use some of the Omniverse Replicator API (since the extension covers a lot more than simply synthetic data generation) but you won't require an Omniverse Replicator dedicated pipeline.

Are you considering cosmos transfer in your data pipeline?

→ Cosmos Transfer can work with both synthetic data (from Omniverse Replicator) and real-world data (curated through Cosmos Curator). In both cases, it modifies visual properties, weather, lighting, actor appearance, based on a text prompt. Annotations are preserved when transformations don't significantly alter object shapes or positions.

Choosing What to Use

- **On top of an Omniverse Replicator pipeline:** If your synthetic data pipeline is already producing good results and you need more visual diversity without re-running simulation, Cosmos Transfer is worth evaluating. It's also a good ally for those pipelines that may require randomization of factors that are hard to simulate (such as weather, people behavior, etc). If you don't have a stable synthetic data pipeline yet, start with Replicator. Transfer is an enhancement layer, and its compute costs are hard to justify without a baseline to improve on.
- **On top of real-world data:** If you've collected field data and need to expand its diversity (different seasons, weather, lighting) without additional collection campaigns, Cosmos Transfer with Curator provides that. This is especially valuable for outdoor deployments where conditions vary widely and revisiting the field is expensive.

In both cases, compute costs are significant. Evaluate whether the diversity improvement justifies the infrastructure investment compared to collecting more real data or expanding Replicator randomization.

Choosing What to Use

Environment

Is your deployment indoor or outdoor?

Indoor environments are where Isaac Sim performs best. Physics interactions and sensor simulation work reliably for structured spaces like warehouses, hospitals, and factories.

Outdoor environments, although plausible, expose real limitations. PhysX does not model deformable terrain (mud, sand, soft soil). Large-scale outdoor scenes push GPU memory hard, and while techniques like level-of-detail or de-spawning non-visible areas can help, Isaac Sim does not provide out-of-the-box solutions for these and they need to be implemented manually. Plan for a larger sim-to-real gap outdoors.

→ One common misconception is: "We cannot simulate every single challenge the real robot will face (for example, deformable terrain), so we cannot use simulation"; and that's far from reality. There is a sim-to-real gap that will always exist, but simulation should still be the first layer of testing to avoid the costs (time and money) of back and forth testing in real life. A mantra to always consider: "If your simulated robot cannot solve its task in perfect conditions (simulation), the real robot will struggle even more."

Choosing What to Use

Hardware

Does your scenario involve multiple robots?

Isaac Sim supports multi-robot simulations, there's no hard limit on how many articulations can exist in a scene. The bottleneck is RTX sensors. If your robots require multiple high-resolution cameras, or 3D RTX LiDARs (which outperform PhysX LiDARs in both fidelity and performance, at the cost of being harder to configure), each additional robot brings a significant reduction in real-time factor.

Whether this is acceptable depends on your clock strategy. If your stack runs on a simulated clock and slower-than-real-time simulation is tolerable, multi-robot scenes are viable. If your stack depends on system clock throughout, meaning the simulation must run close to real time, then multiple sensor-heavy robots will quickly become impractical.

For fleet-level testing, Isaac Sim is generally not the right tool unless you're testing fleet management logic in isolation. If you mock the per-robot stack, run the simulation as a kinematic rendering tool with collision overlap detection and statistics collection, it can work. If you need the real per-robot stack running in the loop, lighter-weight simulators are a better fit for fleet-scale coordination testing.

Does your scenario require dynamic actors (pedestrians, vehicles)?

Isaac Sim supports NavMesh-based actor navigation out of the box, including trajectory queries between points and best-effort collision avoidance between agents. For most robotics validation scenarios, having actors move through the environment near the robot to test avoidance and reaction behavior is sufficient.

Choosing What to Use

Hardware constraints

What GPU resources are available for simulation?

→ Isaac Sim requires an NVIDIA RTX GPU. A minimum of RTX 3080 is workable, though an RTX 4070 or higher is recommended for comfortable iteration even with single-robot, mid-size scenarios, especially if near-real-time performance is needed and multiple RTX sensors are in use.

→ A common workflow is to use moderately powerful workstations (in the RTX 3080 to 4070 range) for interactive development and scene iteration, then run automated SITL pipelines in the cloud for heavy testing. This way only the simulation developers need high-end local hardware, while the bulk of compute for the rest of the engineering team happens on cloud infrastructure.

Integration

How is your stack expected to connect to the simulation?

ROS 2: Isaac Sim provides the Omniverse ROS 2 Bridge extension for native topic, service, and action communication. This is the most common integration path for robotics teams already using ROS 2. The bridge allows quick setup of ROS 2 communications using Omnigraphs, and if finer control is needed, `rclpy` can be used directly from the standalone workflow.

- A note to take into account: the Python version used in Isaac Sim 5.1 is 3.11, which is not out of the box compatible with any ROS 2 distro, meaning that, if custom messages or services are needed, they should be built separately in an environment with Python 3.11 and ROS 2 built from source. This does not happen with Isaac Sim 6.0 (in developer preview as of May 2026) that supports Python 3.12.

Choosing What to Use

PYTHON (NON-ROS): If your stack uses Python libraries without ROS 2, the standard Python standalone workflow is the easiest option and gives the best results. Isaac Sim exposes its full API through Python, and standalone scripts can control simulation, read sensors, and send commands directly.

- As noted in the previous item, Isaac Sim 5.1 is only compatible with Python 3.11, while Isaac Sim 6.0 supports Python 3.12.

C++ OR OTHER COMPILED LANGUAGES: There is no C++ equivalent to the Python standalone workflow. If your stack is C++ only, Isaac Sim's extension system is the way to integrate it. Extensions can expose C++ bindings to Python or provide direct configurable functionality for stack communication. That said, maintaining extensions across Isaac Sim versions can be more involved and harder to debug than pure Python workflows.

C++ OR OTHER LANGUAGES BINDABLE TO PYTHON: Another option is building a Python wheel from your existing codebase and installing it in Isaac Sim's bundled Python interpreter. This is more expensive to develop upfront, but once done it enables straightforward integration from Python without maintaining a full extension. The only requirement is that the wheel must be compatible with the Python version used by Isaac Sim, which has historically changed with every major release.

Indoor Deployments: What Worked and What Didn't

Indoor simulation feels approachable because the environment is structured. The complexity hides in the details.

What worked well:

- **Importing models from other formats:** Importing facility models, placing assets, and setting up a robot's simulation environment is straightforward no matter the format. For robots defined in URDF or MJCF, the importers work out of the box in most cases, the one caveat being that link and mesh names must be valid USD Prim path values, since violations tend to fail silently and can catch teams off guard. For CAD-originated models, the amount of adjustment needed depends on the source: exports from tools like SolidWorks or OnShape each have their own quirks and may require additional cleanup before the model is simulation-ready.
- **ROS 2 integration for an already-modeled robot:** Configuring a robot to communicate with its stack over ROS 2 is a simple task once the simulation model is in place.
- **Synthetic data for perception:** Replicator-generated datasets covering edge cases (damaged labels, occluded pallets, unusual stacking) that are impractical to stage in live warehouses. Basic pipelines can be spun up very quickly and are easy to iterate on.
- **Locomotion policy training:** Isaac Lab with parallel environments. Locomotion, randomization, variadic training difficulty. Transferred to hardware with minimal fine-tuning.

What didn't work well:

- **Omnigraph-first ROS 2 integration at scale:** Using Omnigraphs for ROS 2 is fast for early bring-up, but it limits fine-grained control over exactly what gets published and when. As scenarios grew, we needed detailed publish control, explicit timing logic, or more complex callbacks/message generation, which pushed us toward standalone Python nodes and custom integration code.
- **CAD-origin robot imports as a "one-click" assumption:** Robot models coming from different CAD sources (for example, SolidWorks vs. OnShape) did not require the same cleanup work. Joint setup, mesh organization, and naming conventions needed source-specific adjustments before behavior matched expectations in simulation. Even though it is a little contradictory with what is mentioned in the What worked well section, it makes the difference between the ease of "spinning a simulation with a brand new model" and "having a performant/scalable model".
- **Sensor-heavy single-robot scenes:** A robot can be simulated with multiple cameras and 3D LiDARs, but achieving realtime or good performance, although possible, requires a lot of fine tuning over the existing documentation. Long testing sessions may be required to get good results.
- **Sensor-heavy indoor multi-robot scenes:** Performance dropped quickly when multiple robots each used several RTX sensors, reducing real-time factor enough to make closed-loop timing validation impractical in larger scenarios.

Outdoor Deployments: Where the Stack Gets Stressed

Moving outside exposes fundamental limitations in the stack. Teams working on agricultural, inspection, or off-road robots should plan for a significantly larger sim-to-real gap.

What worked well:

- **Perception pre-training with Replicator:** Specifically, training a model that combined fine-grained instance segmentation with 6D pose estimation. Omniverse Replicator generated labeled datasets across growth stages, object placements, and lighting conditions (dawn to dusk), which reduced the amount of real field data needed to reach baseline accuracy. Time-of-day lighting and hazardous scenarios that would be prohibited in real-world data collection campaigns were the highest-value features.

What didn't work well:

- **Terrain physics.** PhysX models rigid body contact. Real farmland involves soft soil, mud, variable traction, and rut formation, none of which the simulator reproduces. Required real-world fine-tuning on actual terrain.
- **Suspension on 4-wheel platforms over non-flat ground.** PhysX does not handle the 4-contact-with-suspension case correctly unless the terrain is completely flat. Even mild unevenness produces significant jitter in the wheel contacts, which propagates into chassis pose and sensor data and makes the simulation unusable for validating anything that depends on stable odometry or suspension behavior.
- **Cosmos as a starting point for synthetic data generation:** Cosmos Transfer needs an input with reliable ground truth, either real-world data curated through Cosmos Curator, or synthetic data generated with Omniverse Replicator. For those cases where real data is not available, trying to use Cosmos to bootstrap a perception pipeline without a stable Omniverse Replicator baseline ended up with high costs and little retributions. The only path that worked for us was investing first in a solid Omniverse Replicator randomization pipeline and only then layering Cosmos Transfer on top to extend visual diversity.

The Sim-to-Real Gap

The gap between simulation and reality is not a single problem. It manifests differently depending on the deployment environment and the type of robot behavior being validated.

GAP TYPE	WHAT IT LOOKS LIKE	WHERE IT'S WORST	WHAT HELPS
 Behavioral	Actuator delays, fleet timing, contact forces that sim approximates but doesn't match	Indoor fleet coordination, outdoor terrain interaction	From an RL standpoint, physics domain randomization in Isaac Lab (mass, friction, damping, actuator delay)
 Perceptual	Lighting, material appearance, sensor noise differences between sim and real	Both, but outdoor is harder (weather, seasonal change)	Visual domain randomization in Omniverse Replicator, calibrated against real sensor statistics. Data augmentation through noise and Cosmos.
 Operational	Sensor contamination, mechanical wear, network congestion, software memory leaks	Outdoor field deployments, long-duration indoor runs	Not addressable through simulation. Requires staged real-world validation.

Hidden Costs to Watch

Sensor model calibration

Isaac Sim's sensor models (LiDAR, cameras) don't match real sensor behavior out of the box. Calibrating noise profiles, multipath artifacts, and lens distortion against real captures is manual work that may take weeks, not hours. An uncalibrated sensor model produces sim results that don't transfer, regardless of how many randomized scenarios you run.

GPU infrastructure

Isaac Sim requires RTX GPUs. Cloud costs add up quickly for teams without on-premise GPU clusters.

Asset creation

High-fidelity environments require high-quality 3D models. Sourcing or creating these assets takes time and potentially specialized artists, especially for custom facilities that aren't covered by NVIDIA asset libraries.

USD learning curve

If your team is coming from Gazebo, MuJoCo or Unity workflows, the transition to USD-based scene description has a real ramp-up cost. Budget time for it rather than assuming immediate productivity.

Practical Recommendations

Start with perfect sensor data, then layer in degradation

Build and integrate the simulation against ideal sensor output first, that's enough to validate algorithms, integration, and nominal behavior. Once the stack runs end to end, add noise, dropouts, reduced range, and contamination patterns on top of the simulated sensor output or generated dataset, and use those as stress tests. Don't try to model dust, mud, or wear as a physical process inside the simulator; the simulator has no basis to reproduce it accurately. If needed, try to model the effect of those to validate if the robot can compensate for them.

Move from GUI and OmniGraphs to event-driven standalone scripts early

Too many teams stay in the Isaac Sim visual interface or rely on OmniGraphs for integration for too long. Transition to Python standalone scripts as soon as the robot model and environment are stable. Crucially, standalone scripts allow you to hook into specific simulation events (like rendering or physics ticks) to explicitly manage execution timing, which is vital since physics and rendering pipelines operate at completely decoupled frequencies.

Dynamically manage your RTX sensors to save RTF

Running multiple cameras and 3D LiDARs continuously can quickly generate undesired bottlenecks in your simulation. If a specific test scenario doesn't require a sensor's data, do not just "disable" its "publisher", the simulator will still burn resources computing that data in the background. Instead, focus on entirely disabling the sensor or avoiding the creation of the underlying render products and annotators so the simulator stops allocating resources to them completely.

Practical Recommendations

Isolate your ROS 2 custom message builds (Isaac Sim 5.1)

If your stack uses ROS 2 and you are running Isaac Sim 5.1, remember that it bundles Python 3.11, which isn't compatible out of the box with standard ROS 2 distros. Build a dedicated, isolated ROS 2 workspace specifically compiled against Python 3.11 to build your custom messages and services. Experience says a custom message will be needed earlier rather than later, and being prepared in advance will save colleagues from trying uncomfortable workarounds when the time comes.

Structure USDs to be version-control friendly

Treating massive binary USD files like standard code in Git hides unintentional modifications. For robot models: place all visual and collider meshes into binary `.usd` files, and reference them from a plain-text `.usda` file where all joints, sensors, and physics configurations are defined. This makes your robot's kinematic and dynamic properties `git-diff`-able, so changes can be caught easily in pull requests.

For environments: define all props as binary `.usd` files (ensuring they meet Isaac Sim's requirements to be instanceable, to gain on performance), and either store the assembled scenes as `.usda` files or dynamically build them at runtime by spawning USD props based on a separate human-readable configuration file.

Use Fabric for high-frequency runtime data access in Isaac Sim

While the USD API is excellent for scene authoring and static setup, reading or modifying kinematic or non-physics objects and attributes frame-by-frame through the USD layer is computationally expensive. If your Isaac Sim pipeline relies heavily on runtime manipulations outside of the core physics engine, switch to the Fabric API. Fabric is Omniverse's scene data library designed for performance optimization. Because migrating an existing USD-based codebase to Fabric is non-trivial, architect this into your code as early as possible.

Practical Recommendations

Tune the standalone `.kit` file to reclaim performance

When transitioning to the Python standalone workflow, Isaac Sim automatically loads a dedicated, somewhat trimmed-down `.kit` configuration file rather than the full UI-heavy version. However, this default standalone configuration still loads many extensions that are unnecessary for a purely customized simulation loop. Create a custom `.kit` file to explicitly disable these unused extensions. Furthermore, this configuration file is the ideal place to set global physics and rendering configurations, tune viewport update rates, and adjust synchronization parameters. Proper tuning at the `.kit` file level can provide a drastic performance boost.

Keep Isaac ROS nodes connected via NITROS

Isaac ROS achieves high performance by keeping data (like high-resolution images) on the GPU using NITROS (NVIDIA Isaac Transport for ROS). A common mistake is inserting a standard, CPU-based ROS 2 node (like a custom Python image filter) between two Isaac ROS nodes. This forces the data off the GPU to the CPU, and then back to the GPU, completely destroying your latency and CPU budget. Architect your ROS 2 graphs so that hardware-accelerated nodes are sequentially linked.

Enforce heavy Domain Randomization in Isaac Lab

If you are using Isaac Lab to train reinforcement learning policies (like locomotion or grasping), your policy will rapidly overfit to the exact physics of the simulator. To make the policy transferrable to real hardware, you must heavily utilize Isaac Lab's domain randomization APIs. Randomize robot link masses, friction coefficients, actuator stiffness, and control delays during training. A policy that survives heavy randomization is much more likely to survive the real world.

EKUMEN
A GRID DYNAMICS COMPANY

NVIDIA

Making the Right Choice

The NVIDIA robotics stack is capable and, when used well, accelerates the path from concept to deployed robot. But not every project needs every tool, and the stack rewards teams that understand how the pieces connect rather than teams that adopt everything at once.

If you need expert support from a team that works with NVIDIA simulation tools and real robot deployments every day, reach out to us:

contact@ekumenlabs.com

EKUMEN

A GRID DYNAMICS COMPANY

contact@ekumenlabs.com

ekumenlabs.com