Our Eigen Philosophy

ℹ️ **TL;DR:**

Robotics today suffers from fragmentation — separate codebases for simulation, testing, and real robots slow progress. **Eigen** fixes this with a unified, Python-first framework that lets the same code run across simulators and hardware. It's modular, ML-native, and easy to configure, turning robotic development into a single, seamless workflow — the PyTorch moment for robotics.

Introduction

Over the past decade, **machine learning** has transformed entire fields — from biology and healthcare to computer vision and natural language processing. This progress was not only driven by advances in optimization, specialized hardware, and deep architectures, but also by the rise of **software frameworks** that made innovation repeatable. Libraries like **PyTorch** and **TensorFlow** standardized experimentation, abstracted away low-level engineering, and made scaling up research accessible to everyone. Around them, complementary ecosystems such as **OpenAI Gym**, **OpenCV**, and **Hugging Face Transformers** provided reusable building blocks that fueled rapid iteration and reproducibility.

In **robotics**, however, progress has been slower. Frameworks like **ROS** and **Drake** are immensely powerful but notoriously complex. They require steep learning curves, platform-specific builds, and significant systems expertise — barriers that push most teams to reinvent the wheel. In practice, every lab and startup ends up building its own patchwork of scripts and services: one codebase for simulation, another for testing in a different simulator, and yet another for the real robot. This "**three-codebases problem**" fragments development, slows iteration, and introduces failure points at every transition.

To sidestep this, many modern robots now ship with **Python APIs** — from Franka and Unitree to Kinova and Luxonis — letting researchers prototype directly in Python without touching ROS or C++. These APIs lowered the barrier dramatically, enabling control loops, visualization, and machine-learning integration in familiar environments like Jupyter and PyTorch.

Yet accessibility brought new fragmentation. Each API lives in isolation, unaware of simulators, clocks, or sensors. Switching hardware or adding a new device means rewriting large sections of code, and simulation-to-real transfer remains brittle. Python made robotics easier to start, but not easier to scale.

What robotics needs today is what machine learning had a decade ago: a **unified, lightweight, and research-friendly framework** — one that bridges experimentation and deployment, simulation and reality, without friction or duplication.

We introduce **Eigen**, a **Python-first robotics framework** built to unify robot learning and experimentation. Eigen provides a consistent interface across simulation and hardware, enabling teams to move from idea to deployment without rebuilding infrastructure or rewriting code.

The Landscape of Robotics Frameworks

Existing robotics frameworks each solve part of the problem — but none address the full workflow.

ROS remains the industry standard for middleware. It provides powerful tools, a vast library ecosystem, and an active community. But its **C++-centric design**, **build complexity**, and **dependency sprawl** make it ill-suited for rapid research iteration or ML-native workflows.

Simulators such as **Gazebo** and **Webots** have long supported robotic prototyping but are tightly coupled to ROS and often lack **flexible Python interfaces** or **modular design**. They are excellent testbeds but poor platforms for experimentation that spans learning, simulation, and deployment.

More recently, projects like **LeRobot** by Hugging Face have brought much-needed standardization in data representation and collection for robotics. Yet their scope remains narrow — focused on datasets, not full-stack integration across simulators, sensors, and hardware.

The result is an ecosystem that’s rich in tools but poor in cohesion. Robotics today lacks an end-to-end, **ML-native framework** that makes experimentation

fluid, composable, and accessible.

The Painful Robotics Journey

The journey from an initial idea to a deployed robot mirrors the early days of machine learning — full of potential, but fragmented and inefficient. Each phase demands its own tools, interfaces, and workarounds. Progress feels tangible, yet fragile.

The problem is not a lack of creativity or capability; it's a lack of cohesion. Every team faces the same roadblocks, repeats the same integrations, and rebuilds the same systems. What could be an elegant cycle of experimentation and iteration becomes a slow march through incompatible ecosystems.

1. Simulation and Prototyping

Most robotics projects begin in **simulation**. Tools like **PyBullet**, **MuJoCo**, and **Isaac Gym** provide virtual environments to prototype behaviors, collect data, and train policies. They are powerful, fast, and safe — ideal for early experimentation.

But each simulator speaks its own language. APIs differ, coordinate frames vary, and physics models rarely align. Researchers spend days — sometimes weeks — writing wrappers just to make basic control loops work. Debugging consumes as much time as discovery, and progress becomes tied to the quirks of a single simulator.

Instead of focusing on ideas, teams find themselves building infrastructure that already exists elsewhere — only slightly different.

2. Cross-Simulator Validation

Once a policy performs well in one environment, validation begins in another — a simulator with more accurate contact dynamics, better lighting, or more realistic sensors. This transition is never straightforward.

Every element of the experiment — assets, reward functions, observation spaces, and reset logic — must be reimplemented from scratch. Small discrepancies in timing or physics can cause large deviations in behavior. The same task, when ported across simulators, rarely feels identical.

What should be a test of generalisation becomes a test of patience. Code diverges, assumptions drift, and results lose comparability.

---

3. Real-World Deployment

Then comes the hardest step: **deployment on the physical robot**. Everything changes again.

Simulated clocks give way to real hardware timing. Sensor data now includes noise, latency, and calibration offsets. Safety layers must be added, drivers recompiled, and control interfaces rewritten to match manufacturer SDKs. The elegant simulation pipeline becomes a patchwork of scripts and monitors.

What once worked flawlessly in a virtual scene now fails in subtle, unpredictable ways. Bugs are harder to reproduce. Experiments become slower, riskier, and more expensive. Each robot feels like a unique, isolated system — familiar in concept but alien in practice.

---

4. The Fragmentation Gap

By this stage, most teams maintain **three separate codebases**: one for simulation, one for validation, and one for real-world deployment. Each evolves independently, accumulating small inconsistencies that compound over time.

This "**three-codebases problem**" drains productivity and obscures insight. Teams spend more time maintaining infrastructure than advancing research. Collaboration becomes harder, and reproducibility nearly impossible.

Machine learning once faced a similar challenge. Before frameworks like **PyTorch** and **TensorFlow**, every lab managed its own training scripts, data loaders, and evaluation loops. Progress was rapid but siloed. The unification of that ecosystem didn't just accelerate research — it transformed it.

Robotics stands on the edge of the same transformation.

It just needs its framework moment.

Rethinking Robotics Software

Eigen exists to close that gap — to make robotics as iterative, modular, and expressive as modern ML research. Its goal is simple: **standardize experimentation, unify environments, and remove the friction between simulation and reality.**

Design Principles

Eigen is guided by five principles that together define its philosophy.

1. Effortless Installation

Complex build systems and dependency hell should not stand between a researcher and their robot. Eigen installs with a single command:

```
pip install eigen-robotics
```

No CMake. No containers. No platform lock-in. It integrates naturally into existing Python environments on macOS or Linux.

2. Simplicity First

Robotics should feel as approachable as NumPy. Eigen exposes a **clean, Pythonic API** that minimizes boilerplate. Control, logging, teleoperation, and visualisation take only a few lines of code. The focus is on readability and experimentation — not middleware engineering.

3. Modularity and Composability

Every component in Eigen — robot, sensor, simulator — is a self-contained module defined via YAML configuration. These modules communicate through lightweight channels, allowing users to mix and match components effortlessly. Want to swap a camera, replace a simulator, or add a neural policy? Just update the config.

4. Unified Across Simulation and Reality

A single experiment should run everywhere. Eigen ensures that the same Python code operates in PyBullet, MuJoCo, Isaac Gym, or on a real robot. The **sim–real gap** is handled at the framework level, not by the user. Switching environments becomes a configuration change, not a code rewrite.

5. ML-Native by Design

Modern robotics is inseparable from learning. Eigen's types and APIs integrate directly with PyTorch and JAX, making data collection, training, and inference part of one continuous flow. Reinforcement learning, imitation learning, or diffusion control — all plug in seamlessly.

These principles echo the ethos that made PyTorch transformative: **low barrier, high ceiling** — accessible to beginners, powerful for experts.

The Eigen Workflow

Today, robotics workflows are dominated by setup friction. Each new simulator, robot, or sensor means new integration work, new dependencies, new bugs. Eigen replaces that grind with a workflow that feels intuitive and fluid.

1. From Setup to Experiment — in Minutes

A researcher starts not by compiling code, but by writing a **single YAML file** that describes the system — the robot, simulator, and sensors. One command later:

```
eigen launch config.yaml
```

The system comes alive. The robot (real or simulated) is online, data is streaming, and control loops are ready to run. What once required days of integration now happens in minutes.

2. Designing, Not Debugging

In traditional robotics, changing one component often breaks everything else. Eigen's modular design removes this fragility. Components are interchangeable; the framework adapts automatically. Users focus on designing experiments — testing algorithms, tuning behavior, creating ideas — not patching infrastructure.

3. Seeing the Whole System

Complex robot systems are hard to reason about because their data flows are invisible. Eigen brings transparency. It provides live visualisation tools — a graph of active nodes, real-time plots of sensor data, and instant camera feeds — turning the system into something you can *see*. Understanding replaces guesswork, and debugging becomes discovery.

4. Seamless Transition from Simulation to Reality

The step that once required rebuilding everything — moving from sim to real — is now trivial. Eigen keeps observation and action schemas identical across domains. Toggling a single flag switches from simulation to hardware:

```
sim: false
```

Same code, same experiment — just now on the real robot. This continuity collapses the “three-codebases problem” into one unified flow.

Architecture Overview

Under the hood, Eigen is designed for **modularity, distribution, and simulator-agnostic operation**. Its architecture unifies simulation, control, and learning within one Python-first ecosystem.

The Eigen Network

At its core lies the **Eigen Network** — a distributed, node-based system where each process (robot driver, sensor, policy, visualization) runs independently but communicates through a shared messaging layer. This promotes **fault isolation, reusability, and composability**: complex systems are assembled from small, testable building blocks.

Messaging and Communication

Communication follows a **publisher–subscriber model**. Each node publishes structured data — joint states, transforms, images — to named channels. For tasks needing acknowledgments or commands, Eigen also supports **request–response services** using the same schema.

A **central registry** maintains awareness of all active nodes, enabling live introspection, debugging, and visualization — a transparent view of the system at runtime.

Unified Configuration and Launch

Eigen’s YAML-based configuration defines the entire system topology — what runs, where it runs, and how components connect. A single launcher spins everything up from this configuration, no manual process management required.

Observation and Action Abstraction

Borrowing from **OpenAI Gym**, Eigen formalizes **observation** and **action spaces** that align with ML expectations. Observations include sensor data (images, joints, forces); actions correspond to control signals (torques, poses, velocities). This shared interface ensures that policies trained in simulation can run unmodified on hardware.

Sim–Real Consistency

A **transparent sim–real switch** guarantees consistent behavior between domains. Simulation backends like PyBullet and MuJoCo can be swapped without touching control logic, maintaining timing and message schemas across both virtual and physical systems.

Extensible Drivers and Backends

Eigen standardizes device integration through a **ComponentDriver** interface. Whether implemented in Python for rapid prototyping or C++ via `pybind11` for low-level access, every driver adheres to the same minimal API. This means **any robot or sensor** — industrial arms, mobile bases, depth cameras, LiDARs, tactile arrays — can be integrated without altering the core framework.

Through this abstraction, Eigen decouples communication, control, and data flow, letting new hardware plug in as easily as a library import.

Developer Tooling

Eigen comes with built-in developer tools:

- **Eigen Graph** — live visualization of nodes and channels.
- **Eigen Viewer** — real-time camera and depth stream inspection.
- **Eigen Plot** — live plotting of numeric data streams.

Together they form the robotics equivalent of **TensorBoard** — giving visibility, context, and confidence as systems scale in complexity.

The Road to General Robotics

Robotics is moving toward a new era — one defined not by isolated machines, but by **general systems** that can learn, adapt, and act across domains. As simulators grow more realistic, as world models mature, and as learning-based

policies become more capable, the distinction between experimentation and deployment begins to fade. The boundaries that once divided simulation from reality, or perception from control, are dissolving.

Eigen was built for this transition. Its architecture is not tied to any single simulator, robot, or control paradigm. Instead, it provides a common language — a shared infrastructure through which ideas can evolve, regardless of what technologies emerge next. When a new simulator appears, it can be integrated into Eigen through the same driver and messaging interfaces. When world models reach new levels of fidelity, they slot into the same communication layer. As reinforcement learning, imitation learning, and diffusion-based control advance, the policies can be trained and deployed without changing the underlying system.

This **future-proof design** ensures that progress in robotics never invalidates prior work. Code written in Eigen today will continue to run as simulators and environments improve tomorrow. What changes is not the workflow, but the fidelity of the world it operates in. The same configuration that once controlled a robot in PyBullet can, years later, be powered by a differentiable physics model or a learned world simulator — no rewrites, no migration effort.

As the **simulation–reality gap** narrows, Eigen's one-codebase workflow becomes even more powerful. When the physics of simulation begin to reflect the nuances of the real world — friction, latency, visual noise — the boundary between virtual and physical experiments becomes seamless. Training, validation, and deployment collapse into a single unified loop. Researchers can prototype in simulation, test in high-fidelity digital twins, and deploy to hardware with confidence that the same logic will behave consistently. This is not just a productivity gain; it's an architectural shift that enables robotics to scale as quickly as machine learning once did.

At the same time, advances in **foundation models for control** will make robotic intelligence increasingly data-driven. As these policies grow more general, the bottleneck shifts from model design to **integration and evaluation** — connecting models to real systems, testing them safely, and iterating at scale. Eigen directly addresses this challenge. By offering a consistent interface across both simulators and hardware, it turns the deployment of learned policies into a repeatable process. Every improvement in AI translates immediately into an improvement in robotics.

Finally, Eigen's **Python-first philosophy** ensures that this future is accessible. The language of machine learning, data science, and modern research is now

also the language of robotics. This opens the field to a new generation of builders — students, researchers, and startups — who can develop full robotic systems using the same workflows they already know. Small, lean teams gain the leverage to do what previously required specialized infrastructure and large engineering groups.

As the robotics ecosystem matures — with richer world models, higher-fidelity simulators, and increasingly capable AI — **Eigen will evolve with it**. Each step forward in one domain automatically strengthens the others. What began as a framework for unifying simulation and hardware becomes the foundation for a broader transformation — a path toward **general robotics**, where experimentation, learning, and deployment are part of one continuous system.

In this future, progress no longer fragments the field. It compounds.

And Eigen ensures that every breakthrough in simulation, learning, or control flows directly into practice — keeping robotics coherent, connected, and always moving forward.