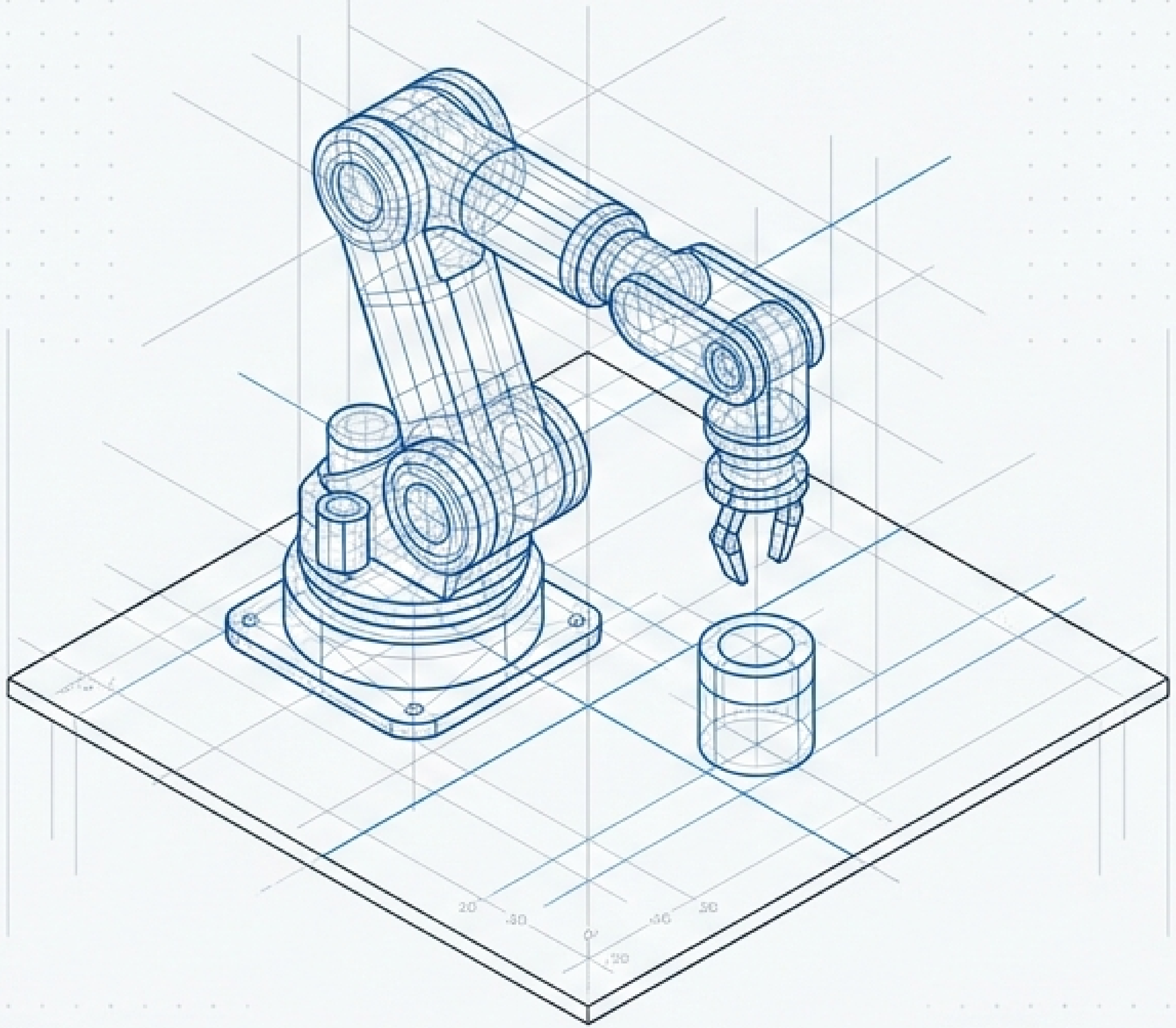
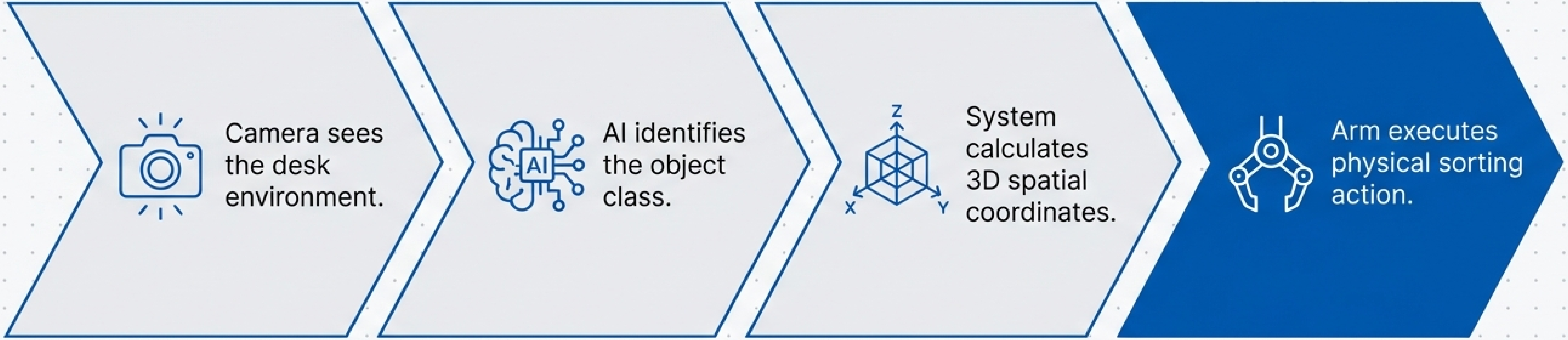


Sort_Trash: Vision to Action

A Progressive Guide to Building an AI-Powered Sorting Robot

The Core Objective

A simplified garbage classification robot.



Key Takeaway: This project bridges the gap between digital AI perception and physical robotic interaction, moving objects to targeted drop zones based purely on visual recognition.

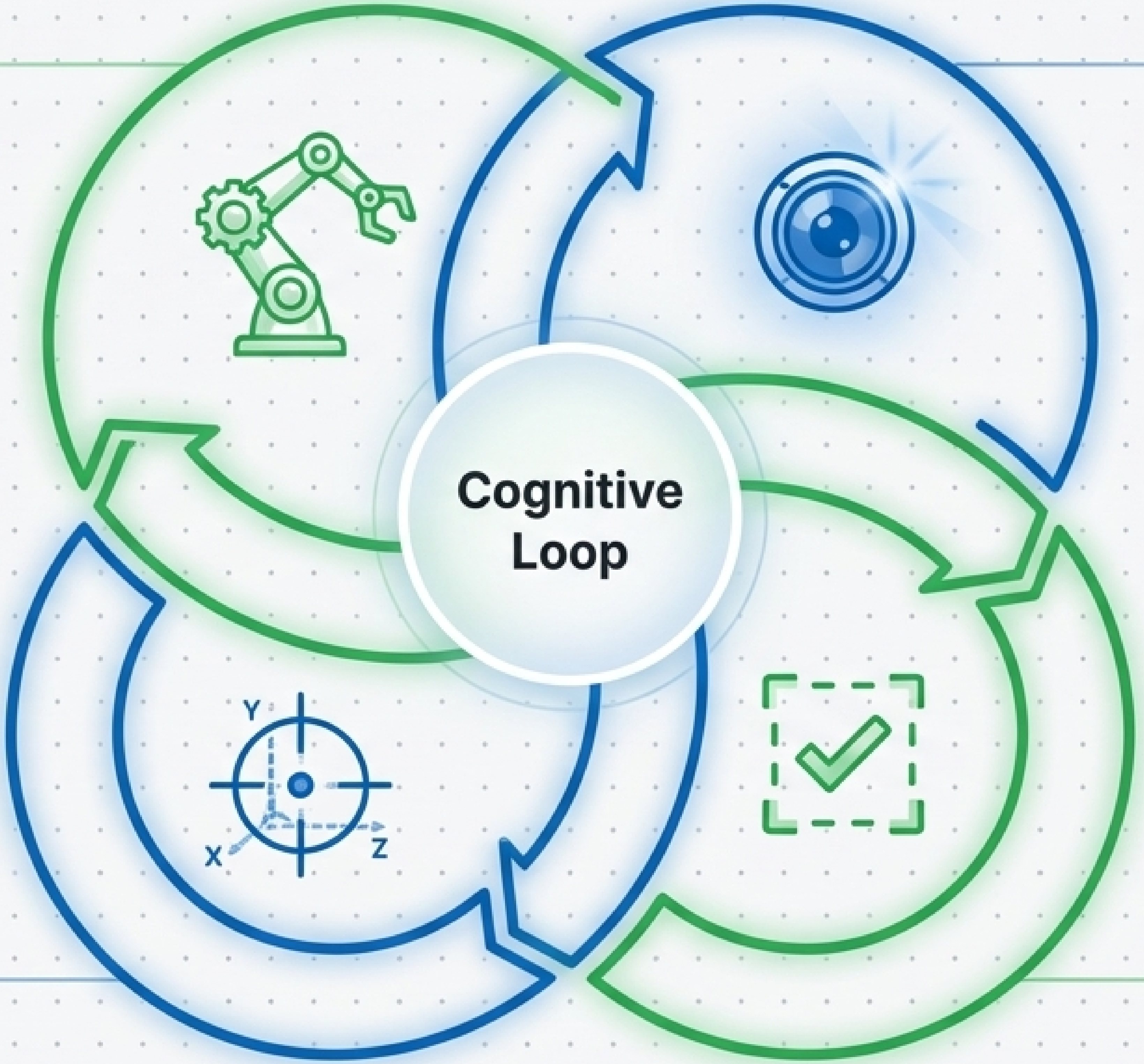
The Information Chain

Act

Execute precise mechanical movement.

See

Capture the physical environment.



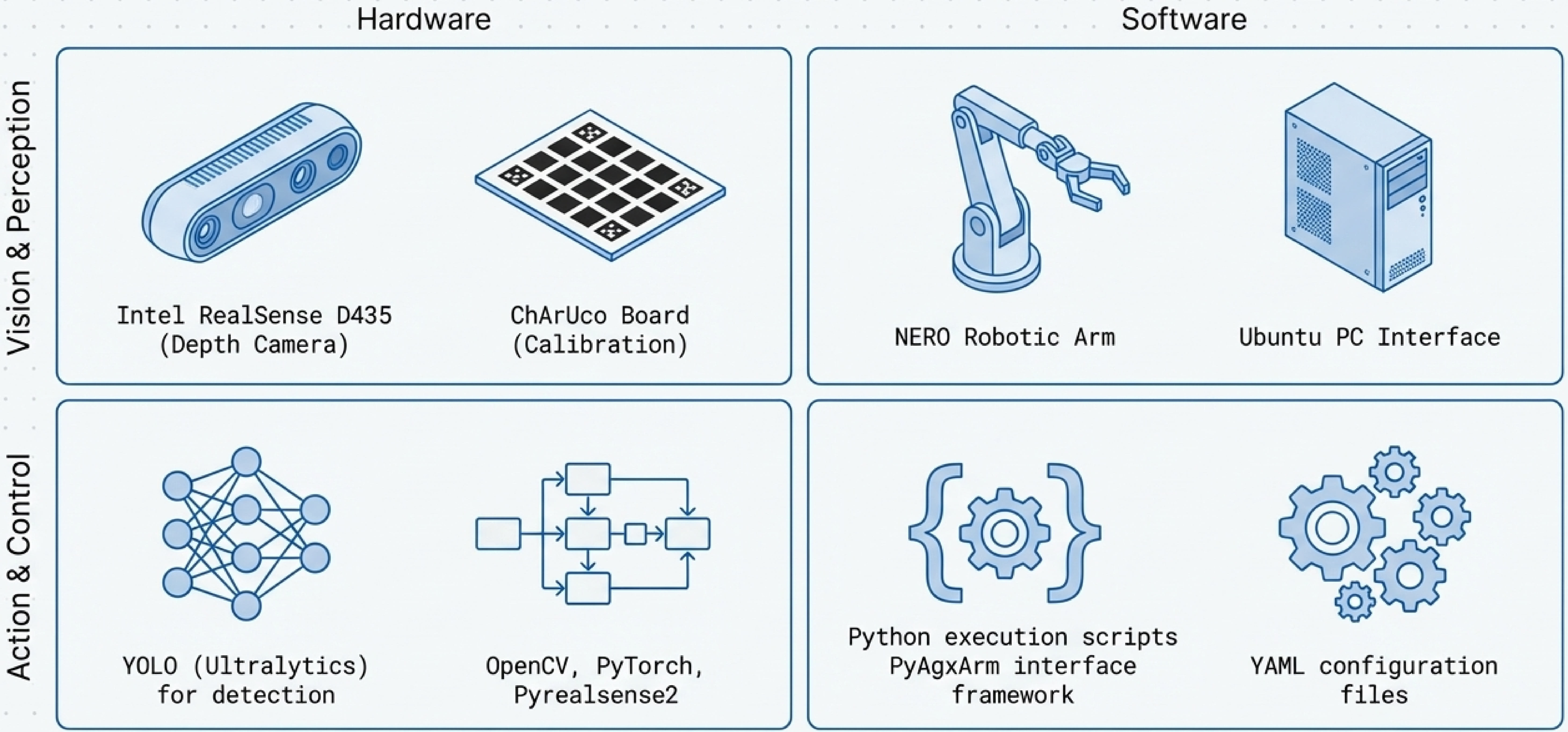
Locate

Pinpoint targets in 3D space.

Understand

Identify object classes.

The Technology Stack



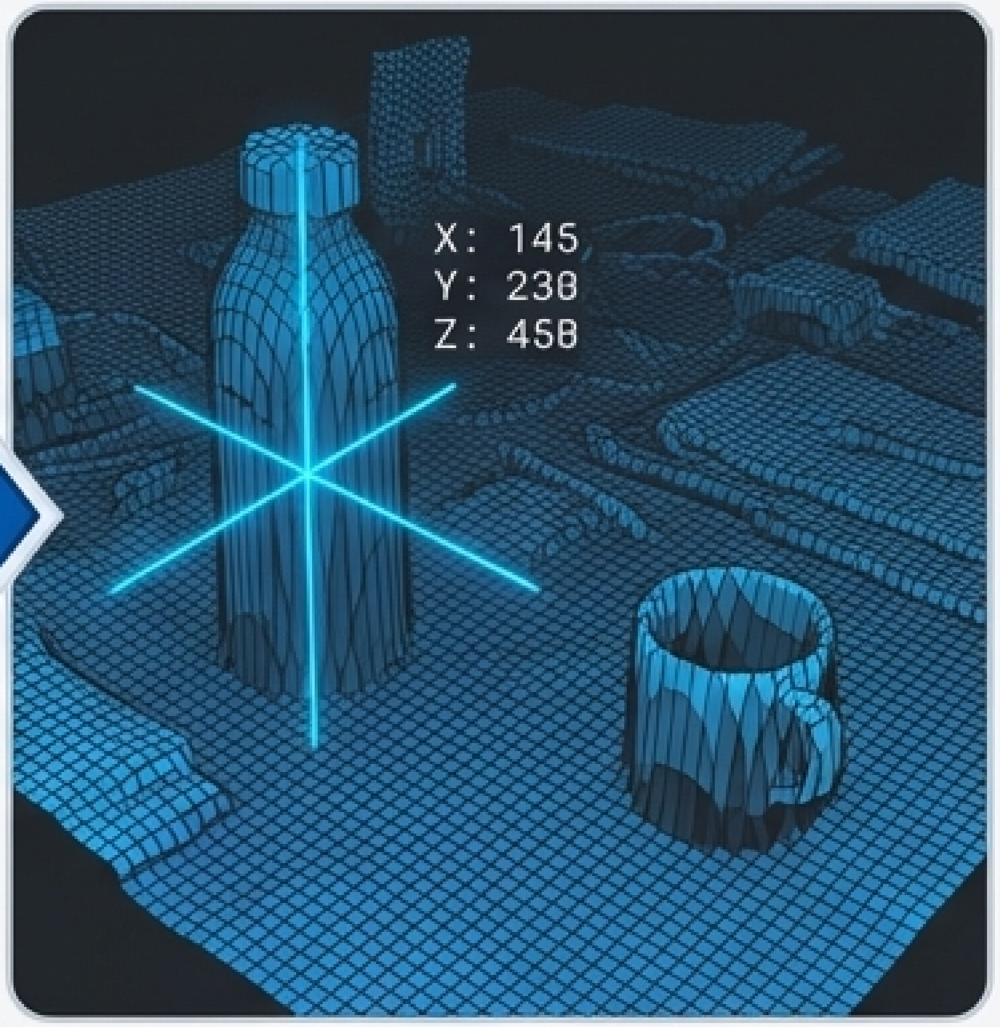
Module 1: Visual Perception

Role: See the desk, recognize targets (bottles, cups), and estimate positions.

Output: Target 3D coordinates calculated exclusively within the Camera's frame of reference.

1. RealSense RGB Stream

2. YOLO 2D Bounding Box

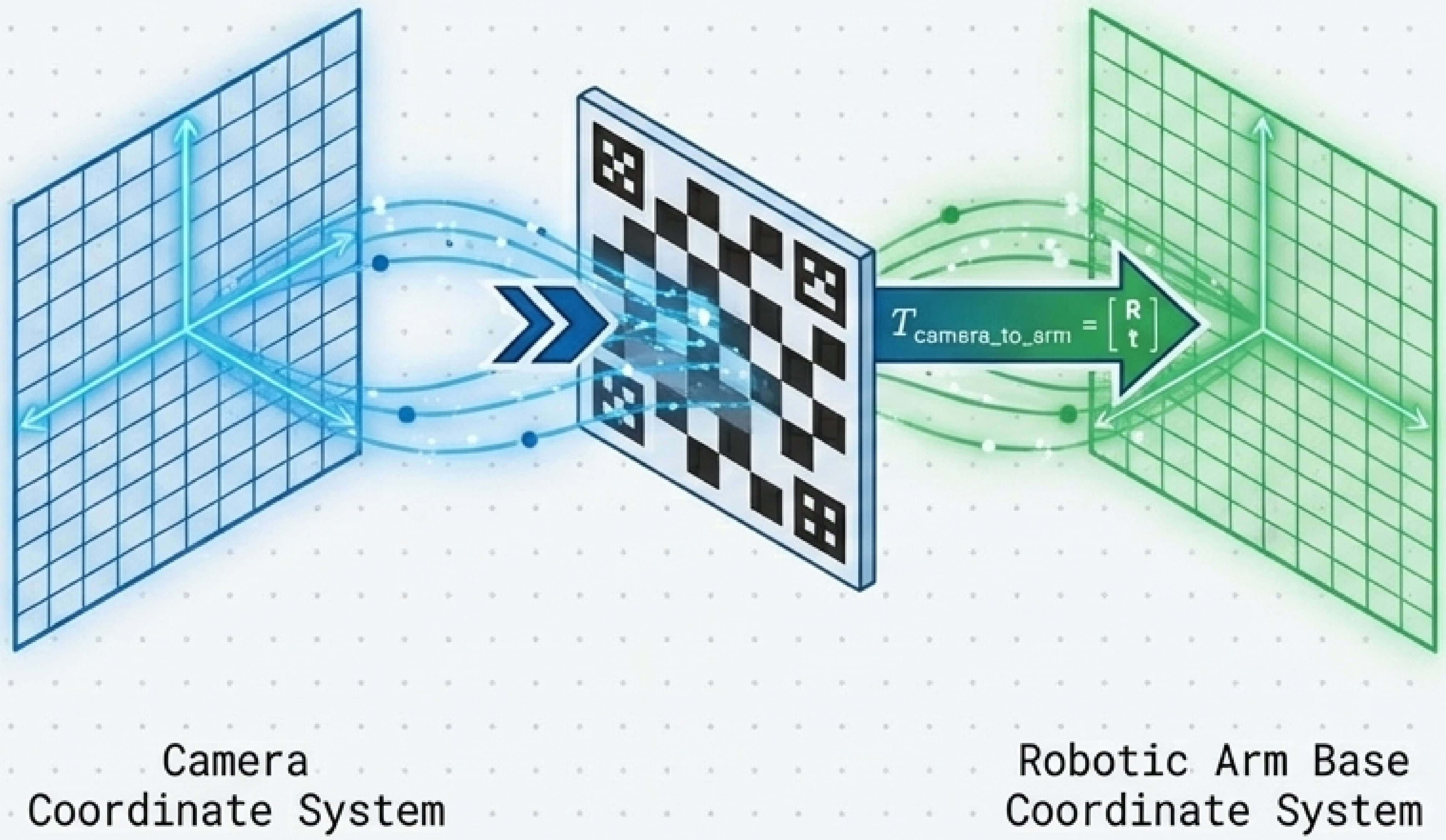


3. Depth Map 3D Coordinate Extraction

Bridging Two Worlds: Calibration

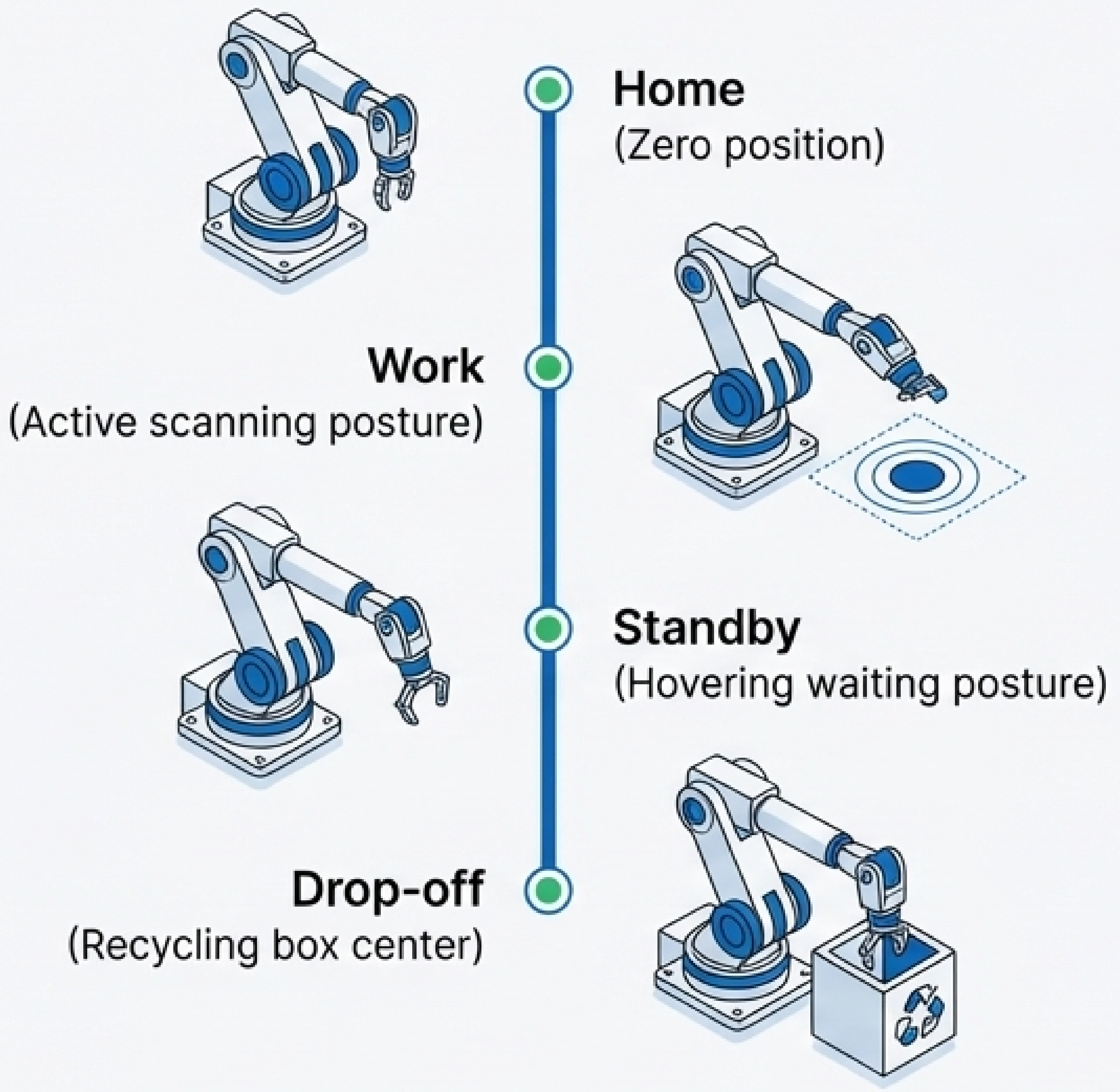
The Problem:
The camera and the arm do not inherently share the same spatial understanding.

The Solution:
Eye-to-Hand calibration calculates the exact mathematical transformation matrix between what the camera sees and where the arm lives.



Orchestrating Movement

Spatial Keyframes (YAML)



Process Control Logic

```
Connect to NERO via PyAgxArm
↓
Read Current Hardware Posture
↓
String Vision + Calibration Data
↓
Run Built-in Safety Check
↓
Send Execution Command
```

The central scripts string vision, calibration, and movement into a single continuous operational loop.

Iterative Engineering: The Fake Grasp

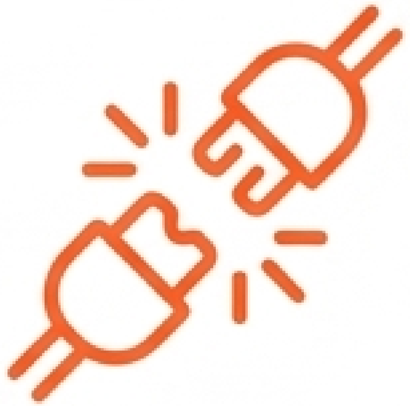
Why fake it? True grasping introduces immense friction. By simulating the grab and release, engineers can perfect vision, positioning, and routing with zero hardware risk.

Traditional Approach: Real Grasp	Project Standard: Fake Grasp
Requires perfect hardware tuning	Focuses entirely on precise motion trajectory
High risk of physical collision	Perfectly safe to test in any environment
Complex mechanical error debugging	Allows pure visual validation of spatial logic
Mandates a physical gripper or OmniHand	Completely hardware-agnostic at the end-effector

Built-In Safety Protocols

Z-Axis Limits

The end-effector target altitude must strictly remain `>= 0.10m` at all times to categorically prevent physical desk collisions.



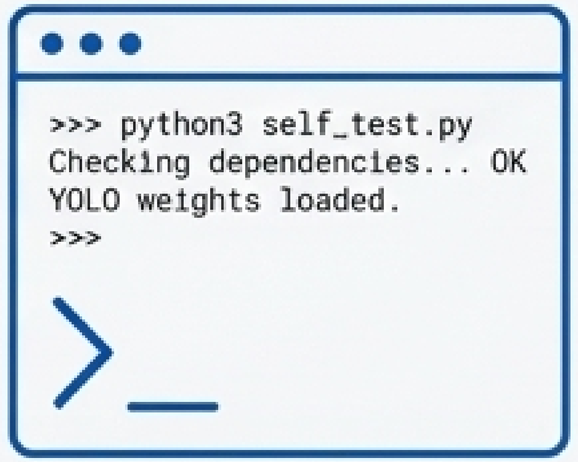
Communication Checks

The system autonomously halts all routing commands if the CAN bus interface (`can0`) is detected as down or non-responsive.

Dry-Runs Required

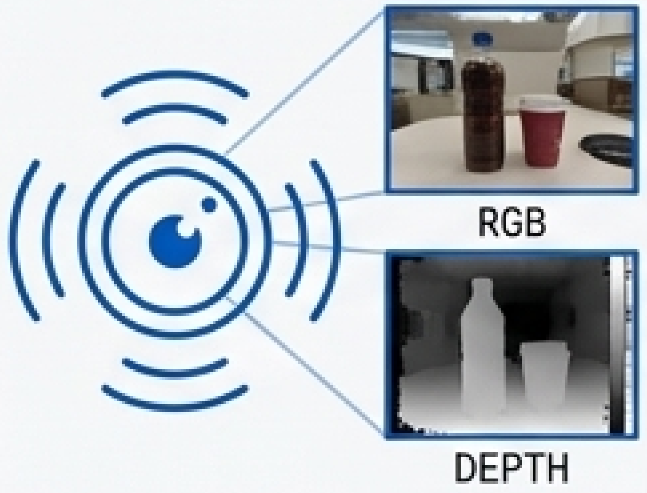
All physical movements must be completely verified safely in simulation before adding the `--go` execution flag to the terminal.

Pipeline Phase 1: Vision & Environment



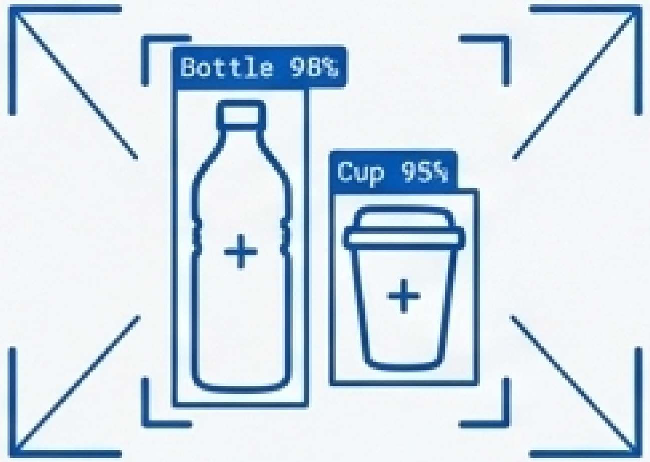
Step 1: Software Self-Test

Verify Python dependencies, YOLO weights, and dummy config files without any hardware attached.



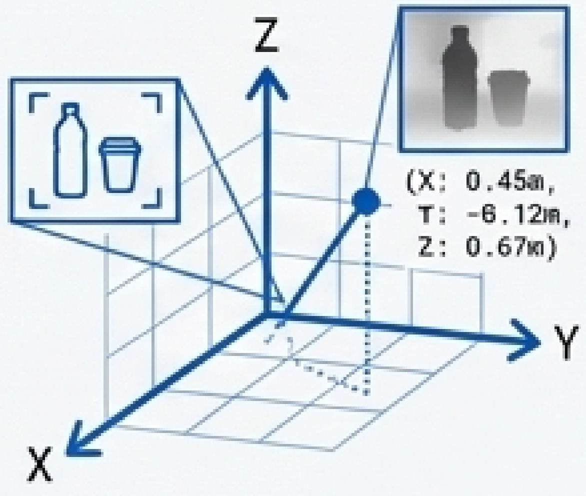
Step 2: Camera Stream

Connect RealSense D435 hardware; systematically verify RGB color mapping and depth mapping feeds.



Step 3: 2D Detection

YOLO inference identifies bottles and cups, actively drawing center-point bounding boxes on the feed.

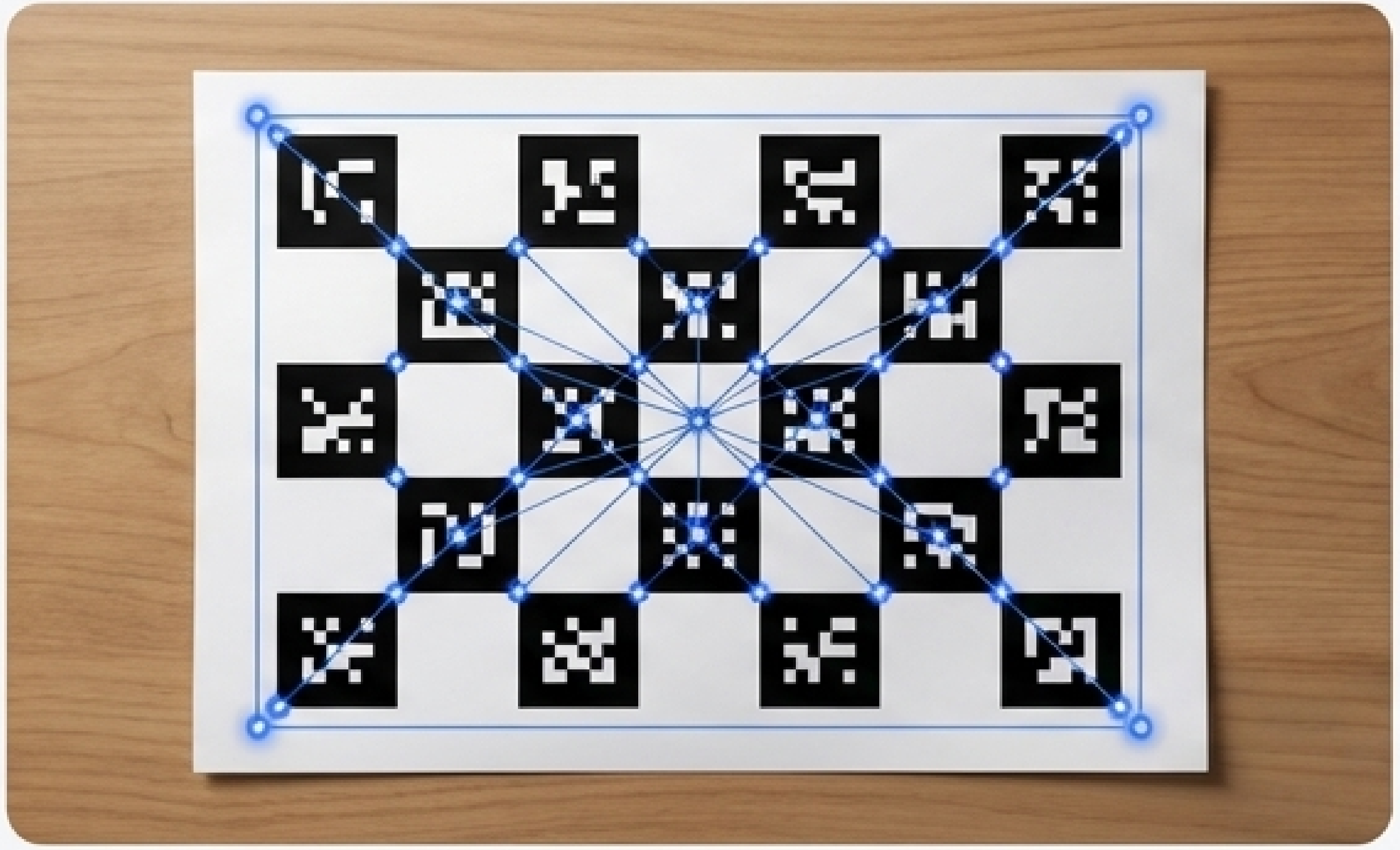


Step 4: 3D Positioning

Combine YOLO 2D detection with Depth map data to output precise XYZ coordinates strictly in the camera's spatial framework.

Pipeline Phase 2: Hardware Sync

Step 5: Hand-Eye Calibration



Print standard ChArUco board at strictly 100% scale. Run baseline calibration scripts to mathematically merge the independent camera and arm coordinate systems.

Step 6: Arm Initialization

```
user@host:~$ can_init --interface can0
[INFO] CAN bus can0 interface initialized.
[INFO] Pinging NERO arm controller...
[SUCCESS] NERO arm controller responded on can0.
[INFO] Starting dry-run sequence...
[INFO] Movement 1: +X, +Y, +Z (reachability check) ... OK
[INFO] Movement 2: -X, -Y, -Z (baseline pose error check) ... OK
user@host:~$
```

Wake up the CAN bus network (can0). Run isolated mechanical movement dry-runs to confidently verify spatial reachability and baseline pose error margins.

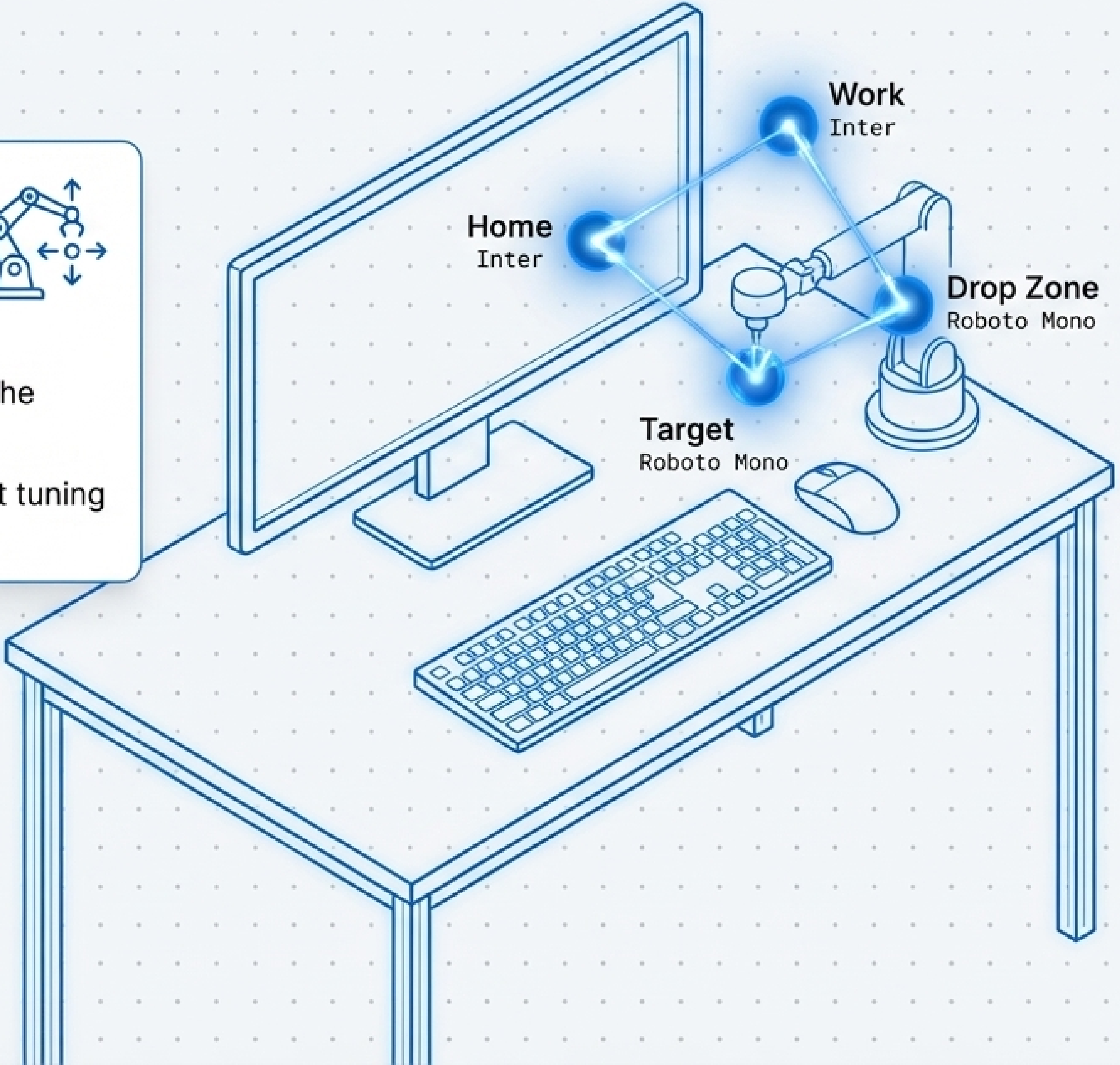
Pipeline Phase 3: Mapping the Workspace

Step 7: Visual Hover Validation

Translate target coords to base coords.

Arm hovers directly above the target.

Supports manual XYZ offset tuning via keyboard (a/d, w/s, r/v).

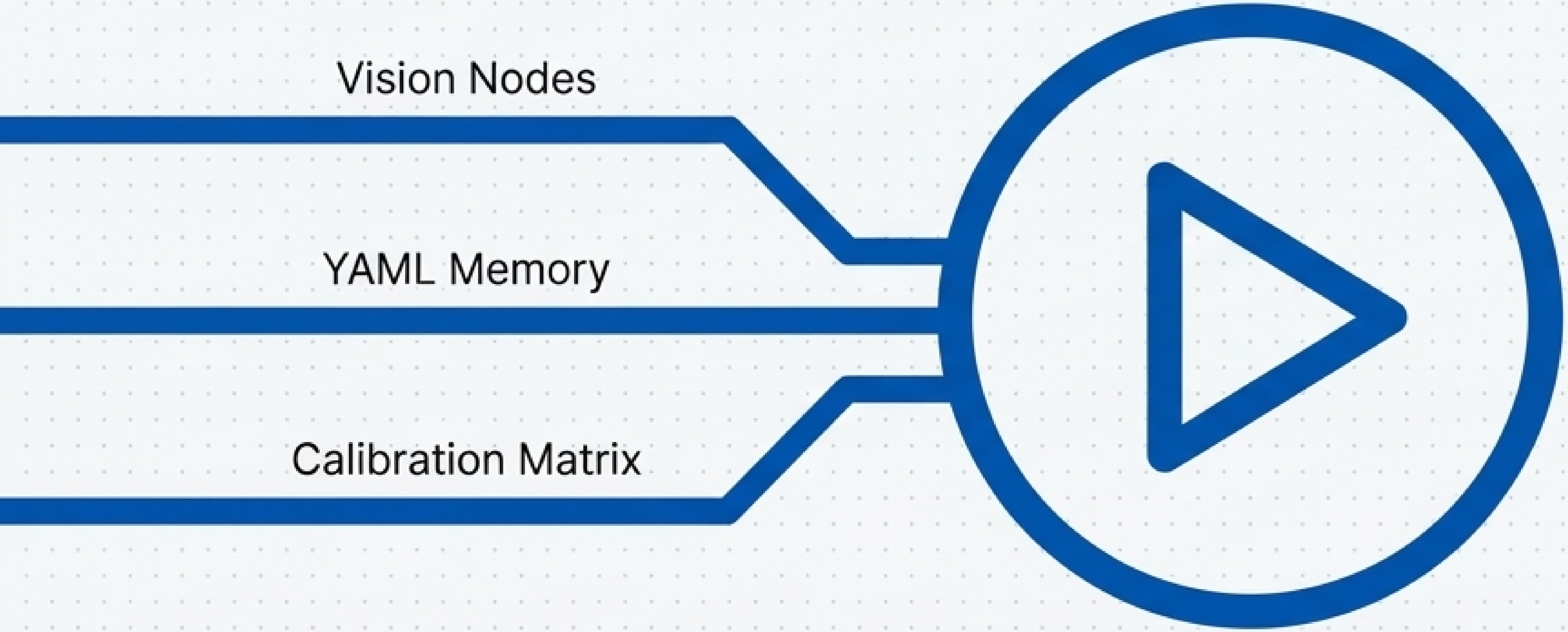


Step 8: Record Key Poses

Save absolute physical coordinates directly to YAML files.

This includes Task Poses (home, work, standby) and Drop Poses (Recycling box centers).

Pipeline Phase 4: Full Cycle Control



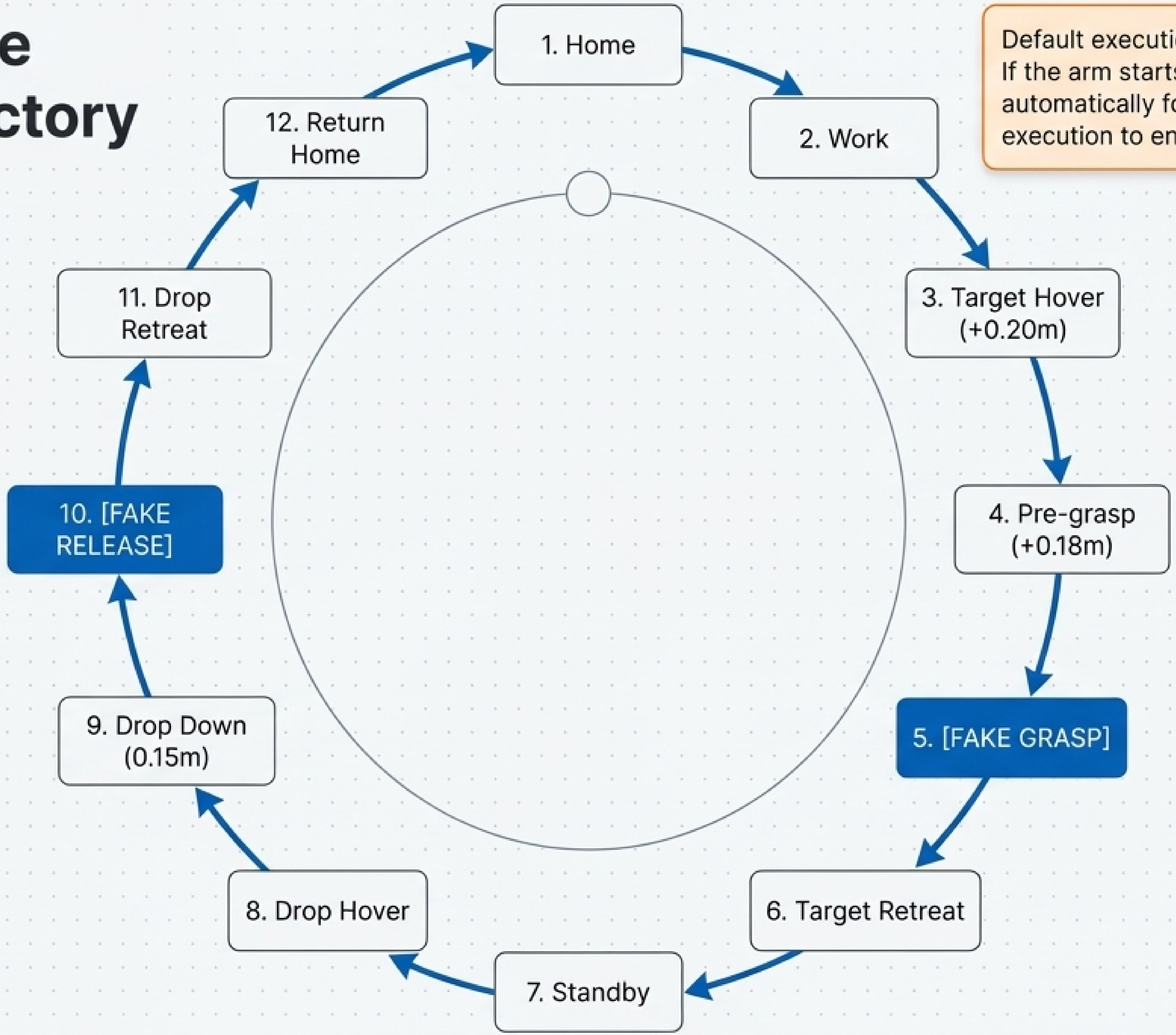
Step 9: Fake Grasp Cycle

Execute the complete, multi-stage, zero-risk movement script charting the arm through 12 specific waypoints.

Step 10: Master Control Skeleton

Final system integration into a single entry-point Python script dictating automated pick-and-place routing.

The Complete Motion Trajectory



Default execution posture: `rx=90`, `ry=-90`, `rz=0`. If the arm starts off-center, the script automatically forces a return to Home before execution to ensure spatial safety.

Beyond the Code: What This Project Teaches