

From neural network demonstration to desk assistant

ALICE is a desk robotics project built around a purchased myPalletizer arm. I am developing the software that lets the system observe its surroundings, maintain an internal state, select a behavior, and request constrained movement. The intended result is a small, supervised desk companion rather than an industrial robot or a general-purpose autonomous assistant.

The original idea

The project began as a neural-network demonstration. A neural network transforms input through layers of numerical operations; I wanted to make those normally invisible intermediate activations visible. The early design connected recognition software to a Glass Brain display, so a viewer could see activity inside the model rather than only its final answer. Those demonstration plans fell through, and I reconsidered what I wanted the project to become.

Why I changed direction

DUM-E from Iron Man suggested a more personal use for the work: a physical desk partner for late nights. I could combine a desktop arm with locally runnable models and keep developing the visualization and control software around that idea. This was a change in purpose, not evidence that the original demonstration had already achieved reliable physical operation.

How the current system fits together

The camera and perception components provide observations. Shared state makes those observations available to behavior routines, which choose actions. An optional local language model interprets context and adjusts behavior multipliers; it does not directly invent motor commands. Movement requests pass through calibration and a controller that checks the number of joints and their allowed ranges. A dashboard exposes internal state and the neural-activation display when data is available.

My engineering work is in these software connections, their failure handling, and the interface. I did not design the commercial arm or train the underlying language model. The following pages show the implemented control boundaries and development failures. The offline interface and simulation tests are separate from physical demonstration evidence.

Desk robotics prototype

The dashboard is the software-facing view of ALICE. It brings together camera input, behavior state, neural activations, and an arm visualization. Running it with hardware disconnected lets me inspect the interface independently of the physical system; it does not demonstrate a completed robot action.

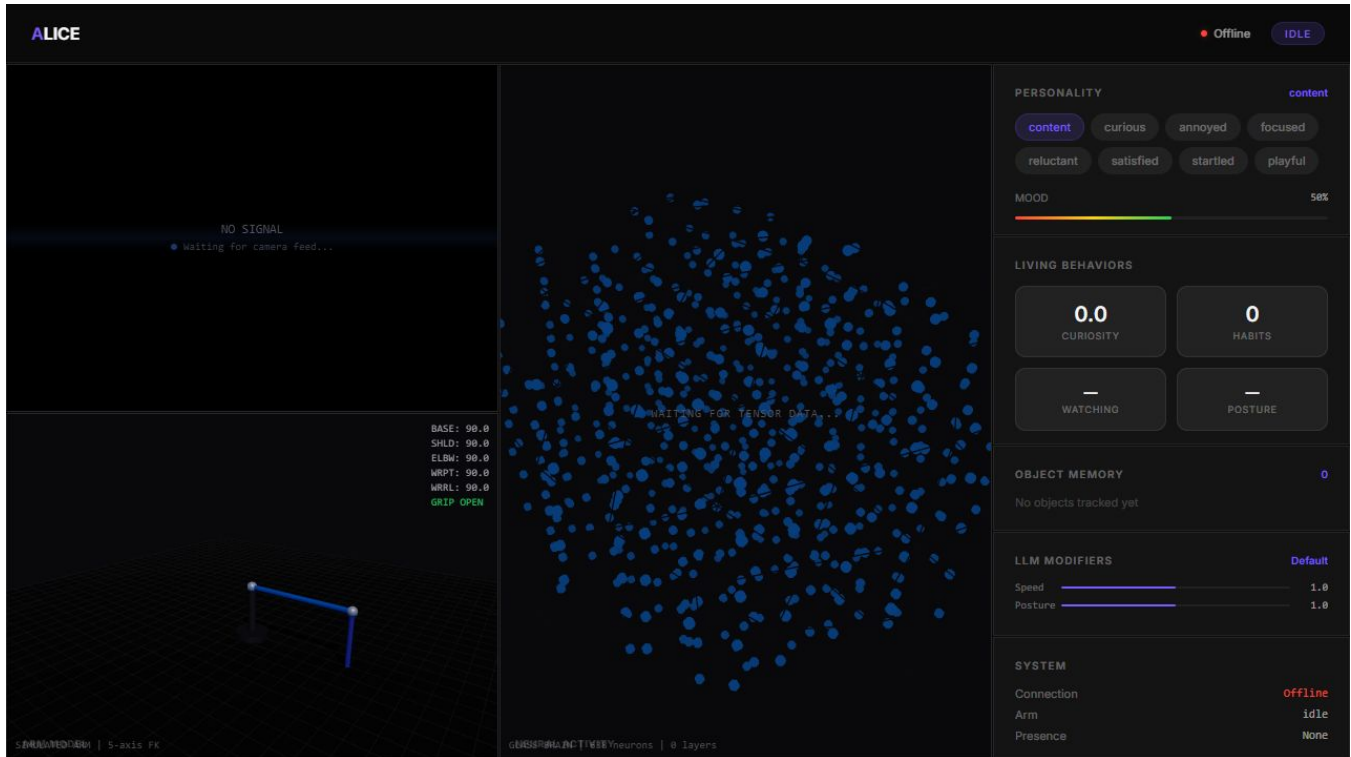


Figure 1. Local dashboard captured 18 October 2026 with hardware disconnected. Values are defaults, not live measurements. The generic five-axis display is not a calibrated model of the four-axis myPalletizer used in this project.

Contribution and hardware

I integrated a commercially manufactured myPalletizer arm rather than designing its mechanical hardware. My contribution is the software connecting observations to behavior and constrained motion, including the dashboard, simulation, calibration path, and recovery handling.

Separating interpretation from motion

Camera observations enter shared state; behavior routines request movement; calibration maps targets into arm coordinates. The controller checks four-axis commands and clamps joint angles. The optional local language model adjusts behavior multipliers, not joint commands. The following pages examine this boundary, calibration, and a truncated model response. Physical grasp accuracy and collision avoidance still require separate measurement; ALICE remains supervised.

Joint command validation

An object detection can suggest a target that the arm cannot safely reach. I placed a shape check and per-joint bounds in the controller before updating its target state. The same constrained tuple reaches the simulated movement or serial command path.

```
171     def _clamp_angles(self, angles: Tuple[float, ...]) -> Tuple[float, ...]:
172         """Clamp each joint angle to its per-joint limits."""
173         clamped = []
174         for i, a in enumerate(angles):
175             lo, hi = self.JOINT_LIMITS[i]
176             clamped.append(max(lo, min(hi, a)))
177         return tuple(clamped)
... intervening source lines omitted ...
183         if len(angles) != self.AXIS_COUNT:
184             raise ValueError(f"Expected {self.AXIS_COUNT} angles, got {len(angles)}")
185
186         clamped = self._clamp_angles(angles)
187         self._target_position = ArmPosition.from_tuple(clamped)
188         self._state = ArmState.MOVING
189
190         if self.simulate:
191             self._simulate_movement(speed)
192         else:
193             self._send_angles(clamped, speed)
```

Source screenshot of hardware/arm_controller.py, lines 171-177, 183-193. Omitted lines are marked.

Listing 1. The controller checks the number of axes and clamps joint angles before dispatch. The screenshot shows selected source lines; the gap is explicitly marked.

Regression cases

Input	Expected software behavior
Four angles outside the allowed ranges	Clamp to the configured joint bounds
Three angles for a four-axis arm	Raise ValueError before dispatch

On 18 October 2026, all fourteen controller tests passed in simulation mode. They cover connection state, movement, selected joint bounds, invalid axes, and a malformed command. The clamping fixture requests (-200, 200, -100, 200); another expects ValueError for three angles. These results verify software behavior, not physical position.

Test reference: tests/test_arm_controller.py, test_move_to_clamps_angles and test_move_to_wrong_axis_count_raises.

Calibration and manipulation sequence

The calibration code fits a planar affine mapping from image coordinates to workspace coordinates derived from recorded arm angles. It then uses inverse kinematics to obtain a joint target. At least four calibration points are required before fitting the transform.

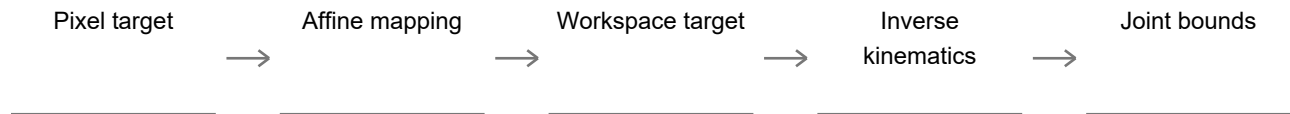


Figure 2. Implemented coordinate conversion. The mapping depends on camera placement, arm dimensions, and the calibration samples; its existence does not establish physical accuracy.

Manipulation routine

The pick-and-place routine approaches above the object, descends, closes the gripper, lifts, travels above the destination, releases, and retracts. Different heights are converted to arm coordinates at each stage. Delays separate the open-loop commands.

```

65     # 7. Release
66     gripper.open()
67     await asyncio.sleep(cfg.release_delay)
68
69     # 8. Retract to safe height
70     retract = calibration.pixel_to_arm(place_px[0], place_px[1], cfg.z_safe)
71     arm.move_to(retract)
72     await asyncio.sleep(cfg.move_delay)
73
74     return True
75
76 except Exception as e:
77     logger.error(f"Pick-and-place failed: {e}")
78     gripper.open()
79     return False
  
```

logic/arm_routines.py, lines 65-79. Working copy inspected 15 October 2026; HEAD ebdc803c.

Listing 2. Normal release and retraction, followed by exception handling. On an exception, the routine logs the failure, opens the gripper, and returns False.

Unresolved measurement

A returned True reports that the software sequence reached its end. It does not independently confirm a grasp or placement. Physical testing must measure offsets and repeatability under a fixed camera and arm setup, including targets near the workspace boundary.

Local model response truncation

ALICE's movement interpreter calls Llama 3.2 3B through a local Ollama server. It sends a compact description of mood, curiosity, nearby objects, and presence, then asks for eight JSON multipliers controlling speed, pauses, posture, and related behavior. The model modulates predefined routines; it does not generate arbitrary joint commands.

Local endpoint and response budget

```
111 model: "llama3.2:3b"
112 base_url: "http://localhost:11434"
113 interval_s: 4.0          # seconds between LLM modulation queries
114 timeout_s: 0.5           # max wait per query (defaults if exceeded)
115 num_predict: 60          # max output tokens (full JSON needs ~54)
```

alice.yaml, lines 111-115. Local source view captured 18 October 2026; HEAD ebdc803c.

The request passes that output-token budget to Ollama and posts to the local generation endpoint. The excerpt below begins inside the request payload; preceding model, prompt, and system fields are omitted.

```
263     "stream": False,
264     "options": {
265         "temperature": self._temperature,
266         "top_p": 0.8,
267         "num_predict": self._num_predict,
268     },
269 }).encode("utf-8")
270
271 req = Request(
272     f"{self._base_url}/api/generate",
273     data=payload,
274     headers={"Content-Type": "application/json"},
275     method="POST",
276 )
277
```

logic/llm_interpreter.py, lines 263-277. Local source view captured 18 October 2026; HEAD ebdc803c.

Why the failure was silent

Commit 0046b56 records an approximately 54-token response being cut off by a 40-token budget. Without a complete JSON object, the parser returned neutral multipliers of 1.0. ALICE's hardcoded behavior kept running, concealing the loss of model-driven modulation.

Repair and limits

I raised num_predict from 40 to 60. The commit reports successful parsing in all thirteen checked contexts, but only 46 percent contextual accuracy: fixing the response length did not fix the model's interpretation. These are historical software checks, not physical trials. The current configuration disables this optional interpreter; the excerpts show its implementation, not a claim that it is running now.