Towards the Harness of Embodied Agents

Qi Wang[†], Tianyi Wang[†], Chengyang Li[†], Shikun Ban, Yurun Chen,

Yizhong Ge, Jason Qin, Chengtai Li[†], Wentao Zhu[†]

Eastern Institute of Technology, Ningbo

[†] Core contributors

Technical Report, July 2026

Abstract

The success of coding agents has established the *harness* as a paradigm: what an agent achieves depends not on the model alone, but on the infrastructure around it. We ask whether the same paradigm extends to embodied agents in the physical world. We present Thea, a harness in which an agentic loop orchestrates robot capabilities, each wrapped as a callable tool. It inherits the core components of coding agents, modified as the physical world requires. The world, however, withholds two abilities that software grants for free: reading the state of the world, and judging the outcome of an action. To bridge these gaps, Thea introduces *Scene Graph as Context*, a persistent, symbolic representation of the world, and *Evaluation as Exit Codes*, which detects when an action should terminate, judges whether it succeeded, and on failure diagnoses the cause. Together they close the loop between the agent and the physical world. Rich behaviors then emerge from the composition of tools, and the closed loop carries long-horizon tasks to completion in real environments.

Project page
https://eit-hai.github.io/thea

Code
https://github.com/EIT-HAI/Thea

Contents

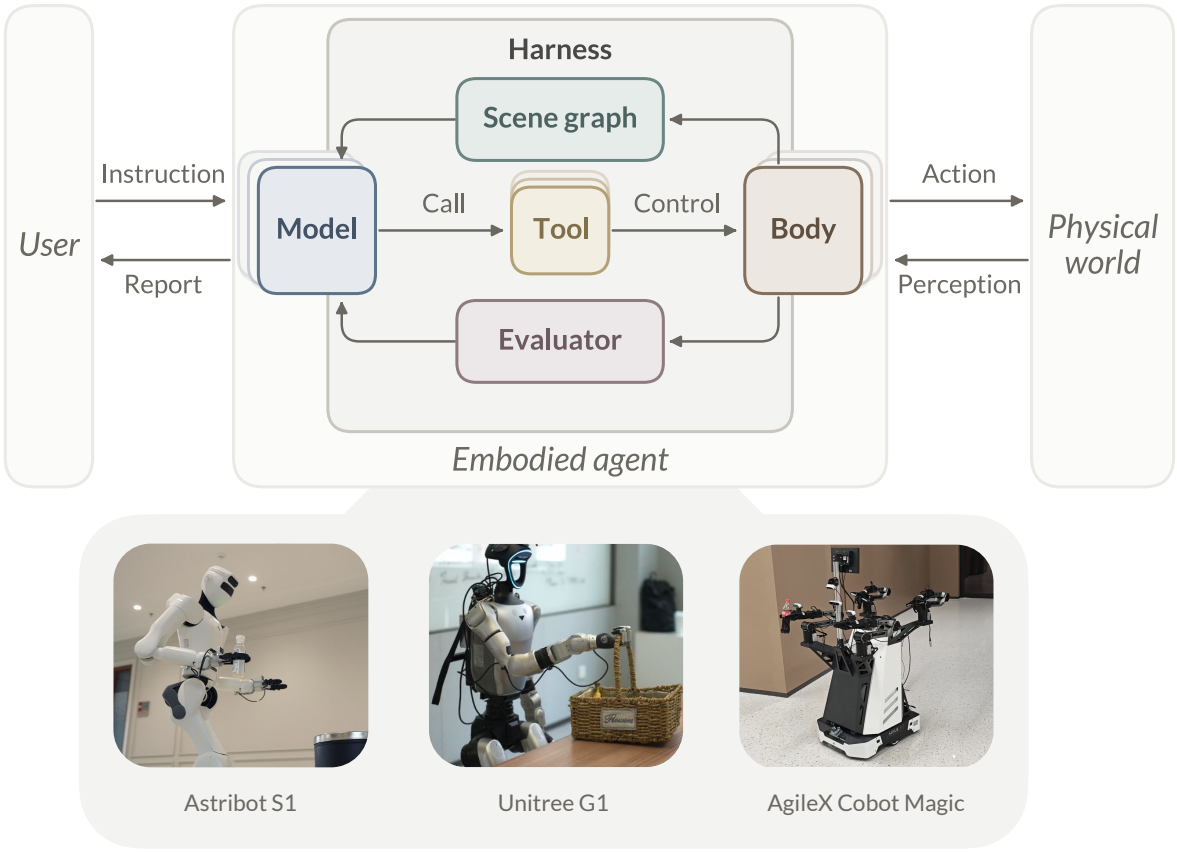


Figure 1. Overview of Thea. The harness wires a model and a body, each swappable, into an embodied agent. The model calls tools to control the body, and reads back the resulting world state from the scene graph and the outcome from the evaluator, closing the loop. The harness ports across embodiments, and we validate Thea on three different robot platforms.

1. Introduction

The rise of *coding agents* [1–3] has reshaped software engineering. Rather than produce correct code in a single pass, they write, test, read the error, fix, and test again, converging on correctness through a closed loop. The insight is not a better model, but a better *harness* [4, 5]: the surrounding infrastructure that grounds the model in its environment, mediates its actions through tools, and closes the loop between intent and outcome through evaluation and recovery. Complex behaviors emerge from the interaction between a simple agentic loop [6] and its tools. Plug in new tools and the same architecture that writes code also produces documents, queries external services, and conducts open-ended tasks with the user; capabilities arrive without changing the model or the loop. The harness has become a paradigm [7].

All of this, however, unfolds in digital environments. In the physical world, the building blocks are in place. Embodied foundation models now perform a range of tasks with growing competence, whether navigating cluttered rooms or manipulating everyday objects [8–10]. Yet a collection of strong atomic capabilities does not constitute a useful robot. What remains open is *orchestration*: weaving these capabilities into long-horizon

[1]: Anthropic (2026), *Claude Code by Anthropic: An Agentic Coding System*
[2]: OpenAI (2026), *Codex: AI Coding Partner from OpenAI*
[3]: Jimenez et al. (2024), *SWE-bench: Can Language Models Resolve Real-world Github Issues?*
[4]: Hashimoto (2026), *My AI Adoption Journey*
[5]: Lopopolo (2026), *Harness Engineering: Leveraging Codex in an Agent-First World*
[6]: Anthropic (2026), *How the Agent Loop Works*
[7]: Weng (2026), *Harness Engineering for Self-Improvement*

[8]: Brohan et al. (2023), *RT-1: Robotics Transformer for Real-World Control at Scale*
[9]: Team et al. (2024), *Octo: An Open-Source Generalist Robot Policy*
[10]: Black et al. (2025), $\pi_{0.5}$*: A Vision-Language-Action Model with Open-World Generalization*

behavior in a dynamic world shared with the people it serves. This is exactly the problem coding agents have proved tractable in the digital world. Their answer is not to wait for an omnipotent model, but to build a carefully engineered harness that amplifies what the model can achieve. Does the same paradigm, then, hold in the physical world?

An agent is, in essence, a closed loop of perception and action: it perceives the world, acts on it, and perceives the result. The architecture of coding agents provides a natural blueprint. At each turn the language model selects a tool (e.g. read a file, run a test, edit code), observes the result, and decides the next action. Each robot capability (e.g. navigation, manipulation) can likewise become a callable tool; a language model orchestrates them through the same agentic loop. The design decouples orchestration from execution. The agentic loop accesses policies and platforms only through their tool interfaces, remaining agnostic to their internals. Complex, long-horizon tasks need not be solved monolithically; the agentic loop addresses them step by step, and new behaviors emerge from flexible composition of tools.

At first glance, transferring this blueprint to the physical world should be a simple matter of redefining the tools. It is not. The physical world is far more complex than software. Closing the loop demands two abilities: reading the state of the world, and judging the outcome of an action. Software grants both for free; the physical world grants neither.

Gap 1: Readability of the World

A software environment is a *designed artifact*, built by humans for machines to read and execute. Source code is text; a language model can read it, grep it, diff it, reason about it. A coding agent perceives it simply by reading, because the environment *is already structured, symbolic, and persistent*. At any moment, the agent can take in the entire state, exact and complete.

The physical world is not a designed artifact. It simply exists. An embodied agent perceives by sensing. Its input is a 30 fps stream of RGB-D frames, continuous, high-dimensional, and unstructured. The agent sees only a local, momentary slice of the world, never the whole. It must piece the whole together frame by frame and hold it in memory; the world keeps no record to consult. A coding agent's "codebase" is free; a physical agent's must be constructed.

Gap 2: Verifiability of Outcomes

Software environments possess an elegant property that is easy to take for granted: every action has a natural *termination signal* and an explicit *success/failure judgment*. Processes exit. Commands return exit codes. Errors print to `stderr`. A coding agent runs a test and instantly knows `PASS` or `FAIL`, and on failure reads the stack trace to diagnose the cause. This closed loop is infrastructure provided for free by the operating system and the language itself. The physical world offers none of this. A Vision-Language-Action (VLA) policy outputs a continuous action stream with no built-in "done" signal. There is no exit code; the world does not report "grasp succeeded." The robot closes its gripper. Did it grasp the cup, or did the cup slip? Did the gripper close on air, or on the rim? A coding agent's "test suite" is free; a physical agent's must be constructed.

What appears "free" to a coding agent is the accumulated product of decades of software engineering. Formal grammars, type systems, and persistent file systems make the state of software readable; exit codes, stack traces, and test frameworks make the outcome of an action verifiable. None of it was built for coding agents; they arrived to find the infrastructure ready, and flourished. Physical environments carry no such inheritance; the missing infrastructure must be built. Nor are the two gaps an arbitrary pair. An agentic loop reaches its environment only through an interface with two directions: actions flow out, and information flows back. The outbound half is the one robotics has spent decades building, which is why policies wrap readily into tools. The inbound half carries exactly two signals: the state of the world, and a verdict on the last action, the same pair the classic agent–environment interface returns [11]. Hence exactly two gaps, one for each missing signal.

[11]: Sutton et al. (2018), *Reinforcement Learning: An Introduction*

We present Thea[1], a harness of embodied agents. Thea inherits the architecture of coding agents: the same agentic loop, capabilities exposed as tools, context, skills, and memory. On this foundation, it supplies the pieces the physical world is missing. *Scene Graph as Context* restores readability: a persistent, structured model of the scene that the language model can read and reason about as a coding agent reads source code. *Evaluation as Exit Codes* restores verifiability: it detects when an action should terminate, judges whether it succeeded, and on failure diagnoses the cause, closing the loop that the physical world otherwise leaves open.

1: From the Greek *thea*, "sight; view."

Through Thea, we demonstrate that with carefully adapted designs, the harness behind coding agents fits embodied agents well. It wires diverse foundation models, policies, and embodiments into a working whole. The resulting system exhibits the following properties, as shown in Figure 1:

- **Extensibility.** Capability is organized around tools. Adding a tool extends the agent, a new policy enters as one more tool, and complex behaviors emerge from their free composition rather than being hand-crafted.
- **Portability.** The model and the body are plug-and-play. A different model or a different embodiment requires no dedicated redesign, because nothing in the architecture is specific to either.
- **A bridge between user and world.** To the user, the agent is a natural interface. Instructions, questions, and explanations all flow through dialogue. To the world, it runs the classic perception–action loop autonomously, turn after turn. The agent bridges the two, translating the user's intention into fluent execution in the physical world.

2. Why the Harness Works

The paradigm of coding agents transfers, in principle, to the physical world. Each robot capability becomes a callable tool, and a language model orchestrates them through an agentic loop [12]. Our goal is to build the harness that makes this transfer work. Yet the harness touches only orchestration, not the components themselves. The same policy, called through the harness, produces the same distribution of outcomes on

[12]: Berman et al. (2026), *Claude Plays Robotics*

any single attempt. If no individual component becomes more likely to succeed, why should the system as a whole? We formulate this question and analyze what factors govern the gain and what they imply for design.

Consider an n-step task in which step i succeeds with probability p_i. Under open-loop execution the task succeeds only if every step does:

For a household robot, a long-horizon task may require navigating across rooms and interacting with multiple objects, leading to a large n.

$$P_{\text{open}} = \prod_{i=1}^{n} p_i \tag{1}$$

Each factor below one compounds the loss; the product decays geometrically with n.

Now suppose the harness wraps each step in a detect-and-retry loop, allowing up to k attempts per step. After each attempt, an evaluator classifies the outcome as success or failure with accuracy α: it returns the correct judgment with probability α and the wrong one with probability $1-\alpha$. A retry is triggered whenever the evaluator reports failure, whether correctly or not, with probability $r_i = p_i(1-\alpha) + (1-p_i)\alpha$. We seek $p_i^*(\alpha, k)$, the probability that step i produces a true success within k attempts. The step succeeds on attempt j ($j = 1, \ldots, k$) if all preceding $j-1$ attempts triggered retries (probability r_i^{j-1}) and the j-th attempt is a true success correctly passed (probability $p_i\alpha$). These events are mutually exclusive, so:

The four outcomes per attempt: true success correctly passed, $p_i\alpha$; true success misclassified and retried, $p_i(1-\alpha)$; true failure correctly detected and retried, $(1-p_i)\alpha$; true failure missed and passed through, $(1-p_i)(1-\alpha)$.

$$p_i^*(\alpha, k) \;=\; \sum_{j=1}^{k} r_i^{j-1} \cdot p_i\,\alpha \;=\; p_i\,\alpha \cdot \frac{1-r_i^{\,k}}{1-r_i} \tag{2}$$

Eq. 2 assumes each retry is an independent draw with the same p_i; we discuss this assumption further below.

Special cases: at $\alpha{=}1$, $r_i{=}1{-}p_i$ and $p_i^*{=}1{-}(1{-}p_i)^k$ (perfect evaluator, failure compressed exponentially in k);
at $\alpha{=}\frac{1}{2}$, $p_i^*{=}p_i(1-2^{-k}) \le p_i$ (random guessing never exceeds the single-attempt rate);
for sufficiently large k, p_i^* approaches the ceiling $\frac{p_i}{(2p_i-1)+(1-p_i)/\alpha}$, monotonically increasing in α, reaching 1 at $\alpha{=}1$.

Substituting into the product over all steps, the task reliability under the harness becomes:

$$P_{\text{harness}} \;=\; \prod_{i=1}^{n} p_i^*(\alpha, k) \;=\; \prod_{i=1}^{n} p_i\,\alpha \cdot \frac{1-r_i^{\,k}}{1-r_i} \tag{3}$$

Each factor p_i in the open-loop product is now scaled by a per-step multiplier $\alpha \cdot \frac{1-r_i^{\,k}}{1-r_i}$; even at moderate α, the gains are substantial, as illustrated in Figure 2.

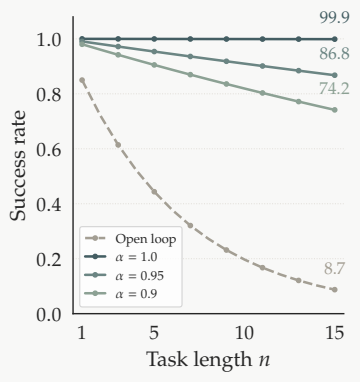


Figure 2. Task success rate as a function of task length n under open loop and under the harness at varying evaluation accuracy α. Per-step success probability $p{=}0.85$, with up to $k{=}5$ retry attempts per step.

The formula makes explicit what the harness should provide: *reliable evaluation* ($\alpha \to 1$). Each additional retry yields geometrically less improvement, and the limit is directly capped by α. In software, $\alpha \approx 1$ is taken for granted; in the physical world, it must be actively constructed and maximized, making it the primary design challenge. Also note that the formula rests on two simplifying assumptions, both conservative for a well-designed harness:

- **Retries within a step.** Eq. 2 assumes each retry is an independent draw with the same p_i. Naively re-executing the same policy without adaptation may fail repeatedly, making retries *worse* than independent draws. A well-designed harness, however, diagnoses the cause of failure and adapts the next attempt, for example by adjusting the robot's position or selecting a more suitable grasp policy, making retries progressively *more* likely to succeed.
- **Fixed plan between steps.** The formulation holds the plan fixed (same n-step sequence under both open loop and harness execution). In practice, a robot may encounter unexpected situations

that a fixed plan cannot anticipate: a door that was open during planning is now closed, or an object has been moved. The harness reads the world state before each step and re-plans adaptively, correcting course and effectively raising p_i itself.

For a well-designed harness, both effects work in its favor; the formulation is therefore a lower bound on the true advantage. The design of the harness presented in the following sections is informed by these considerations.

3. System Architecture

This section is organized by a single question: what form does each component of a coding agent take in the physical world? The answer has the following parts. An *agentic loop* (§3.1) drives the agent. At each turn the *model*, a vision-language model, reads the state of the world and issues a tool call for the harness to execute; we reserve *policy* for the low-level controllers that tools invoke. *Context engineering* (§3.2) determines what the model actually reads at each turn. A *tool protocol* (§3.3) defines how each capability is packaged so that the loop can execute it uniformly. A *skill system* (§3.4) injects domain knowledge into the context when the situation demands it. A *memory system* (§3.5) lets the agent accumulate what it learns, at every scope from a single task to the lifetime of a deployment. Finally, *safety* (§3.6) bounds what the agent may do at all. Each component is inherited from the architecture of coding agents, and each is reshaped in some way by the demands of the physical world. The components that the physical world requires, with no counterpart to inherit, follow in §4.

3.1. Agentic Loop

Listing 1 shows the agentic loop. Each turn, the harness gathers the context, the model reads it and picks the next action, and the harness executes the tool call it names and appends the result; a response with no tool call ends the task. This is the same loop that drives coding agents: the same accumulating messages, the same termination convention. The control flow fits in these few lines.

Everything the model reads is curated in `build_context`, from durable instructions to physical evidence gathered fresh every turn. Section 3.2 unpacks this function, and the world state it draws on is the subject of §4.1.

As in coding agents, every capability is exposed as a *tool*. Navigation, manipulation, evaluation, and user interaction are all entries in the same flat *tool registry*, and the loop has no special-cased logic for any of them. The model issues exactly one *tool call* per turn, because physical actions and their consequences are hard to predict. This *reactive* principle (execute one tool call, observe the outcome, then decide the next) lets the agent adapt to a world it cannot fully anticipate. The absence of a tool call closes the task, and the final text reports its outcome.

Under this design, *evaluation* is also a tool, but its invocation is not left to the model's discretion. In coding agents, evaluation is invisible. A shell

Time units used throughout.

turn — one cycle of the loop: a model call plus the tool execution it requests.

task — one user instruction handled to completion; it opens a run of the loop, spans as many turns as needed, and ends when the model returns no tool call.

session — one continuous engagement with the user; tasks follow one another within it, and the accumulated messages carry across them (§3.2).

`build_context` — assembles everything the model reads, by lifetime; expanded in §3.2 (Listing 2).

`update_memory` — consolidates the finished task's record into durable memory; the subject of §3.5 (Listing 7).

```python
def agent_loop(instruction, session) -> str:
    """Perceive, decide, act, repeat until done."""
    messages = session.messages
    messages.append(instruction)

    while True:
        # perceive: gather what the model reads
        context = build_context(session)

        # decide: the model picks the next action
        response = llm(context)
        messages.append(response)
        if not response.tool_calls:  # done
            update_memory(session)
            return response.text

        # act: execute one tool call per turn
        call = response.tool_calls[0]
        messages.append(execute(call))
```

Listing 1. Agentic loop.

command returns an exit code and a stack trace, and the model reads the result like any other tool output. The physical world provides no such signal, so Thea makes evaluation an explicit tool, and the harness triggers it structurally. A post-execution hook invokes the evaluator after every manipulation (§4.2).

3.2. Context Engineering

Context engineering curates what the model reads. The context window is a finite, attention-limited resource, and what fills it largely determines what the agent does [13]. Unlike coding agents, which can often recover context by reading durable files and preserving a growing transcript, Thea must also refresh physical evidence whose validity changes with robot motion and external events. Context engineering in Thea therefore assigns each model-visible input both a representational role and a lifetime, so stable operating knowledge, current physical evidence, and historical trace occupy distinct parts of the context window: resident, refreshed, and accumulated. Listing 2 unpacks `build_context` from Listing 1, Table 1 lists the contents of each lifetime, and Figure 3 shows how they evolve across turns.

[13]: Anthropic (2025), *Effective Context Engineering for AI Agents*

Resident context holds the System Prompt, Memory, Embodiment Profile, and Tool Definitions. Accumulated context holds Instructions, Task Notes, Model Responses, and Tool Results. These two lifetimes behave just as they do in coding agents. Instructions stay, and the record grows. Refreshed context holds the Scene Graph Brief and Observations, and is where the difference above lands. Observations are the body's current sensor readings: camera images and clearance measurements, the LiDAR-detected distances to nearby obstacles. They expire quickly, and their durable content is distilled into the scene graph, a holistic

```python
def build_context(session) -> Context:
    session.messages.append(task_notes())
    return Context(
        # resident: loaded at task start
        session.system_prompt, session.memory,
        session.profile, session.tool_schemas,
        # refreshed: pulled fresh, kept latest
        scene_graph.latest(), observe(),
        # accumulated: the growing record
        session.messages,
    )
```

Listing 2. Context assembly for one turn.

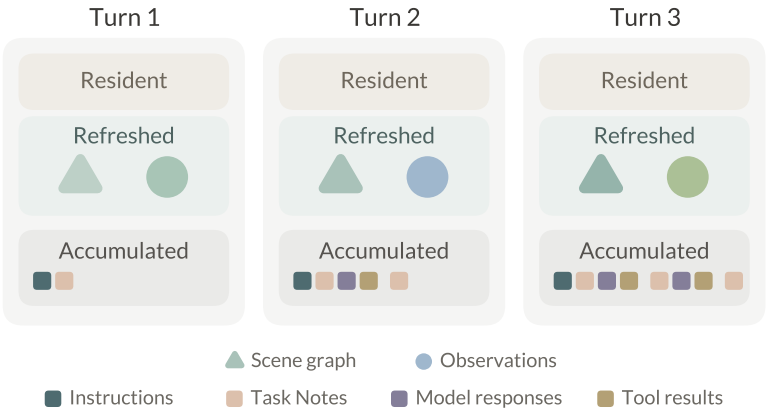


Figure 3. Context evolution across ordinary turns. Resident context stays stable, refreshed context is replaced before each model decision, and accumulated context grows. Compaction and task-end expiry are not shown.

representation of the world, so the latest of each is all a decision needs. Context-truncation experiments in robot manipulation point the same way. Models rely on the recent past far more than on a broad accumulated history [12]. Appendix B expands each item to its full model-facing structure.

[12]: Berman et al. (2026), *Claude Plays Robotics*
Each lifetime has its own source: the resident blocks are loaded at task start, the refreshed values are pulled from the sensor side, and the accumulated record lives and grows in the session.

Resident. Resident context states the durable conditions for the next decision. The System Prompt states the agent's role and its operating rules: one tool call per turn, which evidence source serves which purpose, and when to re-observe instead of trusting stale context. Memory contributes durable knowledge carried across sessions, such as preferences, conventions, and lessons, while tool-specific experience is keyed separately (§3.5). The Embodiment Profile gives the model the body's cameras, frames, and physical limits in the same place every time (§4.3). Tool Definitions are the model-visible side of each tool's contract (§3.3); the post-condition never enters the model's context, and only the evaluator reads it (§4.2). When a tool has accumulated experience, the harness appends a compact summary of it to that tool's description at task start (§3.5).

Table 1. Context lifetimes for one turn: resident context stays stable, refreshed context is replaced before each decision, and accumulated context grows through ordinary turns. Compaction may replace older history, and Task Notes expire with the active task.

Lifetime	Contents
Resident	**System Prompt**. The agent's role and operating rules.
	Memory. Durable memory from `MEMORY.md`: preferences, conventions, and lessons.
	Embodiment Profile. Description of the active embodiment: body, cameras, frames, and physical limits.
	Tool Definitions. `name`, `description` (with the tool's experience summary appended), and `inputSchema`.
Refreshed	**Scene Graph Brief**. Compact rendering of the current global world state.
	Observations. Camera images and clearance measurements.
Accumulated	**Instructions**. Messages from the user.
	Task Notes. Concise active-task working record.
	Model Responses. Model-generated messages, including text and previous tool calls.
	Tool Results. Return envelopes from executed tools.

Refreshed. Refreshed context states what the harness currently claims about the world, at two scales. The scene graph supplies the global scale: a structured summary of what is where, distilled from everything the robot has observed so far. The graph itself lives in the harness; what enters the context is its compact rendering, the scene graph brief, regenerated with source and freshness metadata before each model call (§4.1). Observations supply the local scale: current camera images and clearance measurements. These distances report the nearest obstacles forward, backward, left, and right for local motion decisions, not the semantic distance to a target. The two complement each other. Observations are raw and instantaneous, valid for this decision only, while the scene graph is organized and persistent, yet what it asserts is always about the present. Together they let the model ground every action in the world as it is, not as it was when some earlier tool ran.

Accumulated. Accumulated messages preserve what has happened within the session. When a new instruction arrives inside the same session, the harness appends it to existing messages rather than opening a separate transcript. Before each decision, the harness also appends Task Notes as a concise active-task working record. Inside the loop, Model Responses add model-generated text and previous tool calls, and Tool Results add compact return envelopes from executed tools. These entries help recovery by recording what was tried and what came back. Compaction touches only this lifetime. As a conversation nears the model's context window, older messages are summarized into a shorter record, while resident and refreshed context are preserved [13].

[13]: Anthropic (2025), *Effective Context Engineering for AI Agents*

```python
class Tool(Protocol):
    schema: dict  # the MCP tool definition:
                  # {name, description, inputSchema},
                  # introspected at registration from
                  # the file's signature and docstring
    def call(self, args: dict) -> dict:
        # executes the tool file's function body
        ...  #    {"success": True,  **value}
             # or {"success": False, "reason": reason}
```

Listing 4. The tool protocol: one interface and one result envelope for every tool.

3.3. Tool Protocol

Every capability in Thea is a tool: a `name`, a `description`, and an `inputSchema`, the tool *definition* of the Model Context Protocol [14]. Before a call, this definition is everything the model knows about what a tool does, which is why coding agents treat this agent-computer interface with the same care as an interface built for people [15–17]. What the physical world changes is how much this interface must specify. The rest of this section unpacks the protocol: the contract a tool file carries, how it registers, the envelope every call returns, who executes it, and the hooks it passes through.

[14]: Anthropic (2024), *Introducing the Model Context Protocol*

[15]: Yang et al. (2024), *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*
[16]: Anthropic (2024), *Building Effective Agents*
[17]: Anthropic (2025), *Writing Effective Tools for AI Agents—Using AI Agents*

The contract. A coding tool rarely needs to explain when it will fail, but a physical tool wraps a policy with a success distribution, so its description states preconditions (e.g. "target within reach, gripper empty"), the character of the underlying policy, and what to try when it fails. Each tool file also binds its own *post-condition*, a few sentences stating what the world should look like if this particular call succeeded; at evaluation time the evaluator retrieves it by tool name, so the success criterion travels with the tool rather than with the prompt, and the model never sees it (§4.2). In effect, each tool file carries the tool's full contract: the `inputSchema` says how to call it, the `description` says when, and the post-condition says how its outcome will be judged. Listing 3 shows the shape of one such contract as deployed.

```python
def pick_up(
  object_name: Annotated[str,
    Field(description="...")],
  ...,
) -> dict:
  """Run CaP-X pick-object;
  returns a run id.

  When to use: ...
  Readiness: ...
  Do not use: ...
  Result: ...
  """

# success criterion: fetched by
# evaluator, never shown to model
POST_CONDITION = ("...")
```

Listing 3. A deployed tool file (`pick_up`), contents elided; it registers as the `Tool` of Listing 4.

Registration. A one-line registration, `server.tool()(pick_up)`, turns the file of Listing 3 into the `Tool` of Listing 4: the function name, the docstring, and the signature compile into the three fields of `schema` (`name`, `description`, `inputSchema`), and the body is what `call()` executes. Listing 4 also shows the envelope every call returns; who executes it, and the hooks a call must pass through, are traced by Figure 4.

The envelope. Every tool returns the same envelope: a `success` flag, a value when it succeeds, and a `reason` when it fails. The value is, in the loop's vocabulary, the model's next observation; for a policy-backed tool it can be as thin as a run handle, an identifier for the ongoing policy execution (a *run*), with the substantive evidence arriving when the

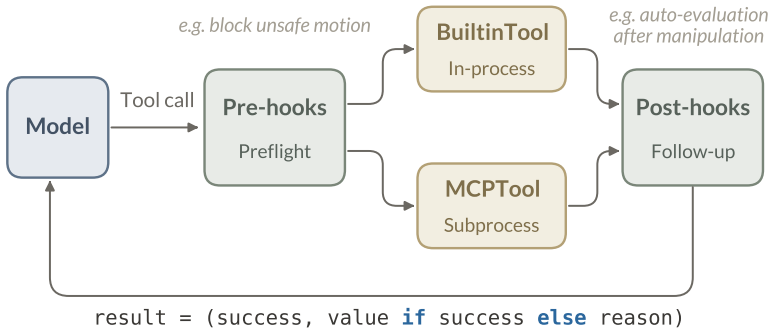


Figure 4. The model issues a tool call. Pre-hooks preflight the call, one backend executes it, and post-hooks trigger follow-up work before the result returns to the model.

post-hook brings in the evaluator (§4.2). Failure never raises an exception; a failed call returns a `reason` such as "the robot is too far from the target to grasp", the closest thing a physical action has to `stderr`. How verdicts and reasons are produced is the subject of §4.2; what the agent does with them is deliberately unscripted. Recoveries emerge from the model recombining the same tools, not from hand-designed recovery routines.

Execution. Who executes a call depends on its weight. Lightweight queries run in-process (`BuiltinTool` in Figure 4), while heavyweight capabilities (e.g. the navigation stack, the manipulation policies) each run in their own subprocess speaking the Model Context Protocol [14] (`MCPTool`). The registry presents both as one flat list, and the model cannot tell them apart. This indifference is what portability rests on. A policy backend can crash, restart, or be upgraded behind the same interface, and nothing above the protocol notices; the same subprocess boundary is what lets one harness drive backends written against three different robot stacks.

[14]: Anthropic (2024), *Introducing the Model Context Protocol*

Table 4 (Appendix A) lists the tool registry deployed on one embodiment.

Hooks. A call never travels straight from the model to the tool (Figure 4). Each call passes through deterministic interception points that the harness owns and the model cannot skip: pre-execution hooks that preflight the call (and may rewrite or block it), and post-execution hooks that trigger follow-up work; an intercepted call returns through the ordinary result channel like any other failure. The two that matter most are a safety check that reads fresh clearance measurements before any base motion (§3.6) and a hook that invokes the evaluator automatically after every manipulation (§4.2). One discipline decides where each rule lives: rules about a single tool go into its description; rules the model must never be trusted to follow go into hooks; neither belongs in the system prompt, which stays short and tool-agnostic.

3.4. Skills

Tools give the model capabilities; skills give it knowledge. Concretely, a skill is a directory holding a `SKILL.md` file: YAML front matter carrying a `name` and a `description`, an instruction body in free-form markdown,

and optionally bundled resources such as reference files or scripts. Loading follows the progressive disclosure of coding agents [18], in three levels: the name and description of every registered skill stay resident in the context at a cost of tens of tokens each, the body enters the context through `load_skill` only when the model judges that the task at hand matches a description, and bundled resources are read only if the instructions call for them. Nothing in this mechanism is specific to robots; it transfers unchanged.

What the physical world sharpens is the boundary between skills and tools. A tool is a contract: schema validation, safety checks, permissions, and evaluation all attach at the call boundary (§3.3). A skill is advice: text the model reads and may weigh. In a world with no sandbox and no undo, whatever acts must sit on the contract side, so a manipulation policy is always wrapped as a tool and never delivered as a skill that teaches the model how to invoke it; on the advice side the guarantees would have nothing to attach to. Skills are left holding exactly what text is good at: knowledge. Which knowledge is decided by elimination: rules about a single tool go into its description, behavior needed on every turn goes into the System Prompt (§3.2), lessons the system accumulates on its own go into memory (§3.5), and what is left falls to skills. In practice this ranges from the operating sequence of a particular appliance to the house rules of a particular workspace. Listing 5 shows the shape of one deployed in Thea: a page of boundary judgments for tidying a desk, knowledge at the level of convention rather than of operation. As a skill is only text, the skill system extends by editing a file: a new appliance, a new house rule, with no retraining and no code change.

[18]: Anthropic (2025), *Equipping Agents for the Real World with Agent Skills*

```
---
name: tidy-workspace
description: Rules for tidying
  a desk: what to remove, what
  to keep, and how to report.
---
## Working order
- ...
## What to remove / what to keep
- ...
## Wrap-up
- ...
```

Listing 5. A skill is a markdown file: the deployed `tidy-workspace/SKILL.md`, contents elided.

3.5. Memory

Memory in Thea is a lifecycle rather than a single store.

Task Notes. Inside a running task, the agent first needs a notepad: a place that keeps what has been done and what remains, so that a long task never depends on the model re-deriving its own progress [13]. In Thea this notepad is Task Notes, a concise record that preserves continuity inside one task. A task begins when the user gives an instruction and ends when the model returns no tool call. At task start, the harness resets Task Notes. After each tool result, it appends a brief event. The notes carry the current goal, phase, and timeline. They are the working record for the active task, not a session transcript and not live world state.

Durable stores. Beyond a single task, the agent needs durable memory. Some of what it learns is general: who the user is and how things are usually done. Some is specific to one tool: how it succeeds and how it fails, lessons that make the next call better. Coding agents have little of the tool-specific kind: `grep` does exactly what its description says, every call, so the tool itself leaves nothing to learn. A tool that wraps a policy is different. Its description promises only a tendency, and where it actually succeeds or fails must be learned from use [19] (§3.3). Thea therefore keeps two durable stores: `MEMORY.md`, the same cross-session memory file a coding agent keeps, and `tool_experience/`, one file per tool. Both are written at task end, when the harness consolidates the completed Task

[13]: Anthropic (2025), *Effective Context Engineering for AI Agents*

```
Task Notes
Task summary:
- Goal: current instruction
- Current phase: ...
Timeline:
- task_start: ...

Memory
- [user-preference] User gives
robot tasks in Chinese.

Tool Experience
 open_drawer
Success:
- Lower drawer pulled outward
  with a visible gap.
Failure:
- If distance is reported or no gap
  appears, move forward before
retrying.
```

Listing 6. Abridged memory artifacts: a Task Notes shape, a Memory entry, and success/failure tool experience.

[19]: Qu et al. (2025), *From Exploration to Mastery: Enabling LLMs to Master Tools via Self-Driven Interactions*

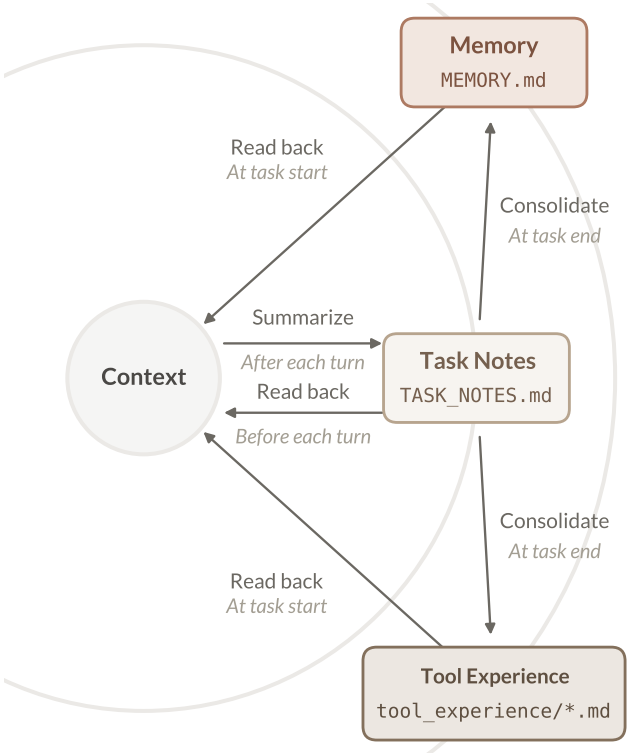


Figure 5. Memory lifecycle. Each store cycles around the Context hub at its own period. Within a task, the harness summarizes each turn's events into Task Notes and reads the snapshot back into the accumulated messages before each turn. At task end, consolidation writes accepted cross-task entries to `MEMORY.md` and accepted tool-scoped lessons to that tool's experience file. At task start, both are read back into resident context, Memory directly and tool experience as a summary inside each tool's description.

Notes, splitting entries by where they will return to the model: cross-task knowledge to Memory, tool-scoped lessons to that tool's experience file. The consolidation call scores each candidate, and the harness writes only those above a confidence threshold. Evaluator verdicts (§4.2) enter this pipeline only as evidence in the trace, never as direct writes. A single verdict is a hypothesis, and whether the agent actually recovered from it shows only in the completed trajectory. Listing 7 shows the consolidation step, and Figure 5 draws the full cycle.

Read paths. The three read-back arrows differ in what the return buys. Task Notes expire with the task; until then, each snapshot keeps the model current on its own progress. Memory returns as resident context, so durable knowledge is visible on every decision without an extra tool call. Tool experience is keyed by tool identity rather than by session or user; the harness appends a summary of each tool's file to that tool's description, so the lessons sit beside its preconditions, failure modes, and recovery hints, exactly where the model weighs whether to call it. Description quality largely determines call quality [17], so the memory lifecycle does more than remember. What it writes and reloads is, in effect, the interface between the model and its tools. The agent improves its own interface with use.

[17]: Anthropic (2025), *Writing Effective Tools for AI Agents—Using AI Agents*

```python
def update_memory(session) -> None:
    """Consolidate a finished task into durable memory."""
    # one model call reads the completed Task Notes and
    # splits them: cross-task knowledge vs tool lessons
    memory, lessons = llm(CONSOLIDATE, session.task_notes)
    append_memory("MEMORY.md", memory)  # user preferences
    for lesson in lessons:              # how a grasp fails
        path = f"tool_experience/{lesson.tool}.md"
        append_memory(path, lesson.entry)
```

Listing 7. Memory consolidation, run once at task end.

3.6. Safety

Coding agents secure themselves with permission rules and an operating-system sandbox [20]: the rules tell the agent what it *should* do, with denials enforced below the model, and the sandbox bounds what it *can* do. Neither is available in the physical world. No sandbox contains a physical action (simulation covers part of this need, but not the deployed world), and many actions cannot be undone. Safety must therefore be enforced before the action.

[20]: Anthropic (2025), *Claude Code Sandboxing*

Thea builds safety into the machinery rather than into the model's behavior [21]. Deterministic checks live in the hooks and the execution pipeline of §3.3, and the model can neither skip nor persuade them. The main one is a *safety filter* on base motion. Before the model decides, the filter puts four-direction clearance measurements into the refreshed context, so the model plans with nearby obstacles in view. Before the robot moves, its hook obtains a fresh reading, blocks navigation when no immediate direction is admissible, and clamps direct translations to the admissible distance. The loop itself is conservative: physical actions run one at a time (§3.1), a failure budget halts the loop instead of letting it run away, and when unsure the agent can stop and ask the user (`query_user`, §3.7). Safety is one of the deepest differences between coding agents and embodied agents in the physical world; Thea makes an initial attempt, and much remains open: force limits, safety around people, and deployment beyond the lab.

[21]: Ahn et al. (2024), *Autort: Embodied foundation models for large scale orchestration of robotic agents*

3.7. User Interaction

Coding agents keep the user reachable throughout a task. The model can ask a blocking clarification question or surface progress, and autonomy is defined with structured returns to the human for information or judgement [16]. Thea carries this over as two tools whose endpoint is a person. `query_user` asks and waits: for a choice among lookalike targets, for confirmation of a borderline action, for permission before touching something personal. `notify_user` tells without waiting: that a long action is starting, that something unexpected happened, that the task is beyond what the robot can do. When to call either is a tool choice like any other in the agentic loop (§3.1).

[16]: Anthropic (2024), *Building Effective Agents*

In the harness, the two tools compose with everything else. Because

they are ordinary tool calls, everything that shapes tool choice shapes them too. A question comes as a last resort rather than a reflex, after the scene graph and the current observations have been consulted, and it can attach the graph's candidate refs and views so the answer returns as evidence the rest of the task can ground on (§4.1.2). A skill can state when a notification is warranted for its task, and Memory keeps settled answers as standing preferences so the same question is not asked twice. Keeping the user in the loop is what keeps an embodied agent adaptive to the unknown and the unexpected: it neither guesses nor fails silently.

4. Bridging the Gaps

The previous section carries the harness across layer by layer. Every layer has a digital counterpart to start from, and the work is adaptation. This section builds what has no counterpart. Gaps 1 and 2 (§1) name the two absences that break the loop itself: the physical world is neither readable nor verifiable by construction, so the infrastructure a coding agent inherits for free must here be built. The scene graph restores readability, giving the loop a world it can reference (§4.1). The evaluator restores verifiability, approximating the exit codes the world does not issue (§4.2). A third absence has no named gap because coding agents never meet it. They have no body. An embodied agent does, and what it can sense, reach, and do depends on which body it runs on. The Embodiment Profile describes the body and the boundaries of its capabilities (§4.3).

4.1. Scene Graph as Context

We restore world readability through two linked layers (Figure 6). An object-centric scene graph maintains a persistent, symbolic world state across turns. A decision-facing interface carries that state into the model's context: a compact scene graph brief by default, with queries keyed by object ref when more detail is needed. §4.1.1 defines the world-state representation and its update process. §4.1.2 explains how the model reads that state, confirms an object ref, and carries the ref into tool calls.

4.1.1. Persistent World State

A coding agent inherits a named and searchable environment. It can search for a symbol, read the path that comes back, and hand the same path to its next tool call. A physical agent starts with none of this. Its visual and depth observations describe one view at one moment, robot state and tool feedback arrive on separate channels, and none of these signals organizes the world into entities the loop can reference again. Yet to continue a task, the loop must know which entity it is pursuing, where that entity was last observed, and how much to trust that evidence. We therefore turn fleeting signals into persistent symbols. An object-centric scene graph ties a stable ref to each object's coarse position and freshness, turn after turn [22–24].

[22]: Gu et al. (2024), *Conceptgraphs: Open-vocabulary 3d scene graphs for perception and planning*
[23]: Rana et al. (2023), *Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning*
[24]: Yoneda et al. (2024), *Statler: State-maintaining language models for embodied reasoning*

Graph structure. The scene graph organizes world state into nodes, edges, and graph metadata, following 3D scene-graph representations that bind semantic entities to spatial structure [25]. An object node carries a ref such as `cup_2`, which the harness derives from the node's `label` and `id`; behind the ref, the node stores a coarse 3D bounding box, confidence, freshness, and a link to visual evidence. A container (e.g. a cabinet or a drawer) keeps its state and tracked contents on the same node, and a robot node tracks pose and holding state. Edges encode `on`, `inside`, `holding`, and `near` relations among object, container, and robot nodes; a `near` edge, for example, carries the measured distance between its endpoints. Graph metadata records provenance, coordinate frame, and update time. Listing 8 shows one recorded object node.

[25]: Armeni et al. (2019), *3d scene graph: A structure for unified semantics, 3d space, and camera*

Recorded object node

```
{
  "id": 2,
  "label": "cup",
  "center": [3.91, -0.18, 1.08],
  "extent": [0.099,
   0.133, 0.096],
  "confidence": 1.0,
  "last_seen_time":
   1778162862.957,
  "image_path":
   ".../2.jpg"
}
derived ref: cup_2
```

Listing 8. A recorded object node, abridged.

Graph maintenance. A persistent picture must be kept true, and two kinds of change reach it: what the robot sees, and what the robot does. The perception backend supplies observation-derived state from visual and depth evidence, refreshing object nodes, coarse positions, confidence, freshness, and spatial relations, echoing dynamic scene-graph construction and structured representations used in real-world ObjectNav systems [26–28]. Execution-derived updates record what physical actions change, such as grasping an object or opening a container; action-conditioned scene graphs likewise use interactions to reveal previously hidden structure [29]. In our harness, this input is gated. After a tool finishes, the evaluator checks its post-condition (§4.2), and only a confirmed outcome may edit the graph. A confirmed pickup, for example, marks the robot as holding the object and removes it from where it lay; a confirmed cabinet opening records that container as open. The harness resolves both inputs against object refs and merges them into a single graph. Together they keep the scene graph an up-to-date picture of the world.

[26]: Rosinol et al. (2020), *3D dynamic scene graphs: Actionable spatial perception with places, objects, and humans*
[27]: Takmaz et al. (2023), *OpenMask3D: Open-Vocabulary 3D Instance Segmentation*
[28]: Zhu et al. (2026), *SysNav: Multi-Level Systematic Cooperation Enables Real-World, Cross-Embodiment Object Navigation*
[29]: Jiang et al. (2024), *RoboEXP: Action-Conditioned Scene Graph via Interactive Exploration for Robotic Manipulation*

4.1.2. Accessing the World State

Scene graph brief. The full graph is too much to hand the model every turn. The context window is a finite, attention-limited resource (§3.2), and most graph detail is irrelevant to the next decision. The harness therefore renders a scene graph brief into the refreshed context, with frequently needed information present by default and the rest fetched on demand. Nearly every decision must know which objects the world offers, where each one lies, and how far to trust that evidence, so the brief lists every object ref with its coarse position, confidence, and freshness. The robot's own pose and holding state condition every action choice, and tracked container contents keep known but unseen objects addressable, so both enter the brief. Detail that only particular decisions touch, such as an object's extent, its relation edges, and its stored images, stays in the graph and is retrievable by ref. The brief is compact because it selects fields rather than pruning nodes; the boundary need not be exact, since anything omitted costs one extra query, not a failure. The harness regenerates the brief before each call while preserving the graph across calls, so the model reads a current global picture without reconstructing world state from accumulated messages [13]. Listing 9 shows an abridged

[13]: Anthropic (2025), *Effective Context Engineering for AI Agents*

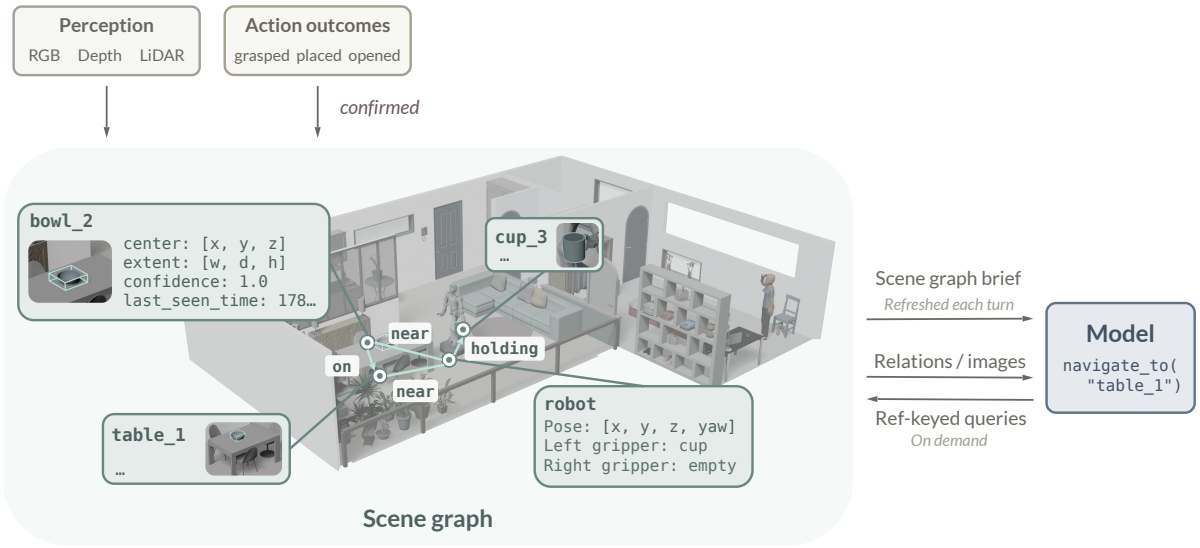


Figure 6. Maintaining and reading the world state.[1] Observation-derived updates and evaluator-confirmed action outcomes are resolved against object refs and merged into one graph, carried into the next turn. The scene graph brief is pushed into the refreshed context every turn. Ref-keyed queries fetch omitted evidence on demand.

brief.

1: Scene layout designed with Sweet Home 3D. Includes 3D models and textures distributed under free licenses.

```
Scene graph brief:
 objects=10;
 stamp=1778162862.957
Robot: pose=[0,0,0,0],
 gripper empty
Observed objects:
 - cup_2 at (3.91,-0.18,1.08);
  conf=1
 - laptop_5 at (5.21,1.18,1.29);
  conf=1
 ...
```

Listing 9. Scene graph brief rendered before a decision.

Ref-keyed queries. What the brief omits, the model fetches itself. A small set of query tools reads the full graph on demand, keyed by ref, so every answer attaches to an entity the graph already names rather than introducing a new one. `get_object_relations` returns the typed relations of a ref, and `get_image` returns the visual evidence stored on its node. For example, the graph names objects at the class level, so when the user asks for "the red cup" and the brief lists three cups, no ref settles the instruction by name; the model calls `get_image` on the candidates and reads the color from their stored views, evidence it needs for this one decision and not as standing context. A spatial cue such as "the cup by the desk" resolves similarly through `get_object_relations`. The design ends at the tools; what follows is the model's own behavior. It chooses which query to issue and when, and asks through `query_user` when no gathered evidence settles what the user means. It treats an empty lookup as absence of evidence, not absence of the object, and obtains a fresh observation or inspects a likely container before concluding that the object is not there. Composing these moves, the model resolves a vague phrase into a symbolic instance and hands its ref to the next action tool, `navigate_to(target="cup_2")`; the binding keeps naming the same entity through evaluation and retry until new evidence breaks it.

A confirmed ref settles which entity the robot pursues and where it approaches; it does not promise that manipulation is ready from the current pose. That judgment falls to later evidence. Near the object, a fresh observation grounds visibility and alignment, and after each action the evaluator rules on the outcome (§4.2). The scene graph supplies identity and coarse location, nothing more. It makes the world readable and addressable without becoming a readiness or success oracle.

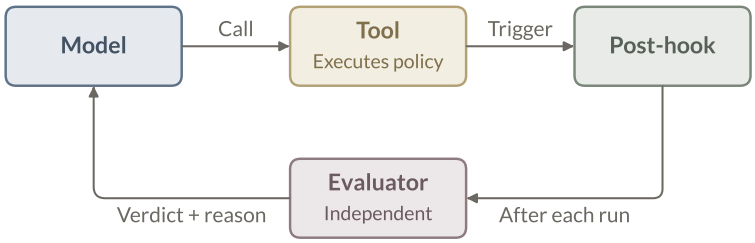


Figure 7. The harness triggers the evaluator structurally after every manipulation. The evaluator reads only the current observation and the per-tool post-condition, then returns a three-state verdict with a reason to the model.

4.2. Evaluation as Exit Codes

In the paradigm of coding agents, the environment itself tells the agent how its last action went: commands return exit codes, tests fail with stack traces, and the agent debugs against these signals [30]. The physical world provides no such infrastructure. A robot that has just attempted a grasp receives no signal telling it whether the task is still in progress, has succeeded, or has failed, let alone why: a slipped grip and a collision both go unreported. Supplying this missing verdict is neither optional nor easy. The reliability analysis in §2 shows that evaluator accuracy α directly bounds the end-to-end task success rate: a harness is only as reliable as its judge. The difficulty also inverts across worlds. In coding agents a single step judges itself, and the hard questions begin only at the level of whole trajectories; in the physical world, judging even one action is already the open problem.

[30]: Anthropic (2026), *Demystifying evals for AI agents*

Thea therefore makes evaluation an explicit module of the harness. An evaluator judges each action's outcome and returns the verdict with a structured failure reason, the exit code and the stack trace that the world never provides. Designing it comes down to three questions, answered in turn below and illustrated in Figure 7: *when* to judge, since not even the end of an action announces itself; *who* judges, since a self-report of success cannot be the verdict; and *what* the verdict carries, since a bare boolean gives replanning nothing to work with.

Timing. When to judge depends on the family of the policy behind the tool. For end-to-end models [10, 31], no signal marks completion, so the evaluator judges the run online, at each action-segment boundary: `process` means the run should continue, `success` that the post-condition has been satisfied, and `failure` that execution should stop, with a step budget as the backstop. For Coding-as-Policy [32] approaches, the generated program terminates on its own and reports its progress through control flow, so the same contract degenerates to the two terminal states.

[10]: Black et al. (2025), $\pi_{0.5}$*: A Vision-Language-Action Model with Open-World Generalization*

[31]: Black et al. (2024), π_0*: A Vision-Language-Action Flow Model for General Robot Control*

[32]: Fu et al. (2026), *CaP-X: A framework for benchmarking and improving coding agents for robot manipulation*

Independence. Who judges is settled structurally. The harness places the evaluator as an independent component after tool execution, triggered by a post-hook rather than by the model's explicit invocation. Its inputs are restricted to two things: the current observation, and the per-tool post-condition (what the world should look like after a successful call) [33]. It reads neither the model's reasoning nor its self-report, the reasoning-blind discipline Anthropic applies to its own action classifier [34]. A

[33]: Meyer (2002), *Applying'design by contract'*

[34]: Anthropic (2026), *How We Built Claude Code Auto Mode: A Safer Way to Skip Permissions*

model grading its own work is an unreliable judge [35], and a biased one [36].

The verdict. What the verdict carries determines what recovery can follow. A bare status already closes the loop, but only blindly: the agent can retry, yet cannot adapt. For example, if a grasping action fails, the agent needs to know whether the failure was caused by failing to reach the target, missing the object, object occlusion, or dropping the object after grasping; each cause calls for a different recovery. Beyond the status, the evaluator therefore returns evidence and the failure reason; Listing 10 shows the contract returned for one such failed grasp. The division is deliberate [37, 38]. The evaluator only judges the outcome and explains the failure, and what to do next is left to the agent, which holds the full context the evaluator never sees. Downstream, a `success` gates the scene graph update after the action (§4.1), and the failure reasons feed the tool experience consolidated at task end (§3.5).

[35]: Huang et al. (2024), *Large Language Models Cannot Self-Correct Reasoning Yet*

[36]: Panickssery et al. (2024), *LLM Evaluators Recognize and Favor Their Own Generations*

```
{
  "status": "failure",
  "evidence": [
    "the bottle remains on the table
    ",
    "the gripper closed without
    lifting it"
  ],
  "failure_reason": "the robot
    stopped too far from the bottle
    to grasp it"
}
```

Listing 10. The evaluator contract for one failed grasp.

[37]: Liu et al. (2023), *REFLECT: Summarizing Robot Experiences for Failure Explanation and Correction*

[38]: Guo et al. (2024), *Doremi: Grounding language model by detecting and recovering from plan-execution misalignment*

4.3. Embodiment Profile

A coding agent never needs to be told about a body. It works through the shell and the file system, an interface that is the same everywhere and, when in doubt, can simply be asked. A robot's body takes no such questions, and nothing about it is standard. Morphology, sensing geometry, mobility, and reach vary across platforms, and jointly determine what evidence is available, how spatial measurements should be interpreted, and which actions are feasible. The Embodiment Profile states these body-specific facts explicitly, as a compact, replaceable document. At deployment, the harness loads the profile selected for the active embodiment and retains it throughout the session. Replacing the profile changes the body-specific context as a unit while task instructions retain the same form. This is what makes one harness portable across embodiments. Localizing embodiment knowledge in one document keeps it consistent across decisions and makes adaptation easier to inspect, maintain, and validate.

The Embodiment Profile organizes stable body-specific information into three complementary sections, summarized in Table 2.

Table 2. The three sections of the Embodiment Profile, their stored items, and their role in model decisions.

Profile item	Stored information	Decision support
Operational Envelope		
Base Footprint	Radius or dimensions of the body's ground footprint.	Relates available space to the area occupied by the body.
Base Mobility	Supported base translations and rotations.	Rules out motion commands that the base cannot execute.
Reachable Workspace	End-effector reach and operating height.	Determines whether an object or location is physically reachable.
Perception Configuration		
Sensor Modalities	Types of evidence available to the model.	Identifies whether a decision can use images or distance evidence.
Model-Visible Views	Named mounted camera views exposed to the model.	Identifies the views that the model can request.
Base-Relative Positions		
Camera Positions	Fixed camera positions relative to the base center.	Situates visual evidence on the active body.
Initial Gripper Positions	Gripper positions in the initial posture relative to the base center.	Situates the manipulation interfaces on the active body.

Operational Envelope records the body's stable limits for motion and manipulation. The base is whatever carries the body, a wheeled chassis or a humanoid's legs, and its entries record what the base can do, not how it does it. Perception Configuration describes the evidence made available to the model, such as color images and clearance measurements. Base-Relative Positions records where sensing and manipulation interfaces are located relative to the base center. Taken together, Operational Envelope constrains feasible actions, Perception Configuration identifies the evidence available for a decision, and Base-Relative Positions situates that evidence and the manipulation interfaces on the active body.

5. Experiments

We evaluate Thea on real robots in real-world environments. The experiments explore three questions. The first is how the full harness compares with alternative execution architectures as task complexity increases. The second is how reliably the evaluator, on which closed-loop execution depends, judges the state of an ongoing task. The third is what capabilities emerge from the composition of tools in complete deployments across different embodiments.

5.1. Task Setup

Our experiments span three robots with distinct embodiments. The main quantitative experiments run on the Astribot S1, a wheeled dual-arm humanoid that combines an actuated head and torso, an omnidirectional mobile base, and two gripper end effectors, 25 degrees of freedom in total. We additionally explore Thea on AgileX Cobot Magic, which pairs two arms with a mobile base, and on Unitree G1, which carries BrainCo Revo 2 dexterous hands with six degrees of freedom per hand. These deployments test whether the same harness can compose capabilities exposed by different physical interfaces.

Table 3. Real-world tasks of increasing difficulty.

Level	Task	Task description
L1	Short-horizon manipulation	Pick up one tabletop object and place it into a basket.
L2	Long-horizon manipulation	Pick up all tabletop objects and place them into a basket.
L3	Navigation and manipulation	Navigate to one table, retrieve the instructed object, and deliver it to a target table.

To evaluate Thea and the baselines across tasks of increasing difficulty, we organize the experiments into three levels, L1–L3, as summarized in Table 3. L1 is a simple pick-and-place task that isolates a single manipulation cycle on a fixed tabletop. L2 preserves the fixed-table setting but extends the task horizon. The agent must repeatedly locate, pick, and place every object on the table into a basket while tracking progress across multiple manipulation cycles. L3 is the most challenging task and couples navigation with manipulation across two workspaces, requiring the agent to alternate between the two as it travels to the source, identifies and retrieves the instructed object, navigates to the destination, and completes the delivery. Across tasks, we vary object categories and placements. For each method, we conduct 20 independent trials on L1 and L2 and 15 independent trials on L3.

We compare Thea with the following systems: end-to-end policies [10, 39, 40], a Coding-as-Policy baseline, and a hierarchical planning method based on SayCan [41]. The Coding-as-Policy baseline revises its robot-control program over multiple turns using execution traces and structured text describing the initial scene, subsequent visual changes, and task completion. This follows the CaP-Bench M3 setting in CaP-X [32]. We use GPT-5.5 with reasoning effort set to `high` as the decision model for Thea, CaP-X, and SayCan. For perception, Thea builds on the Astribot S1 SDK and SysNav [28]. Each end-to-end policy is trained or fine-tuned on 200 collected demonstrations per task. The Coding-as-Policy and hierarchical planning methods use the same tools and underlying implementations as Thea (Table 4, Appendix A). For the orchestration baselines [32, 41], this setup controls for the underlying robot capabilities and isolates how each architecture selects and composes tools, and how it recovers when they fail. Two of the end-to-end baselines, ACT and $\pi_{0.5}$, are the same policies that back Thea's manipulation tools (Table 4). The differences reported below therefore reflect how complete each architecture's orchestration is, not how capable the underlying policies are.

[10]: Black et al. (2025), $\pi_{0.5}$*: A Vision-Language-Action Model with Open-World Generalization*

[39]: Zhao et al. (2023), *Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware*

[40]: Wu et al. (2026), *From Foundation to Application: Improving VLA Models in Practice*

[41]: Ahn et al. (2022), *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances*

[32]: Fu et al. (2026), *CaP-X: A framework for benchmarking and improving coding agents for robot manipulation*

[28]: Zhu et al. (2026), *SysNav: Multi-Level Systematic Cooperation Enables Real-World, Cross-Embodiment Object Navigation*

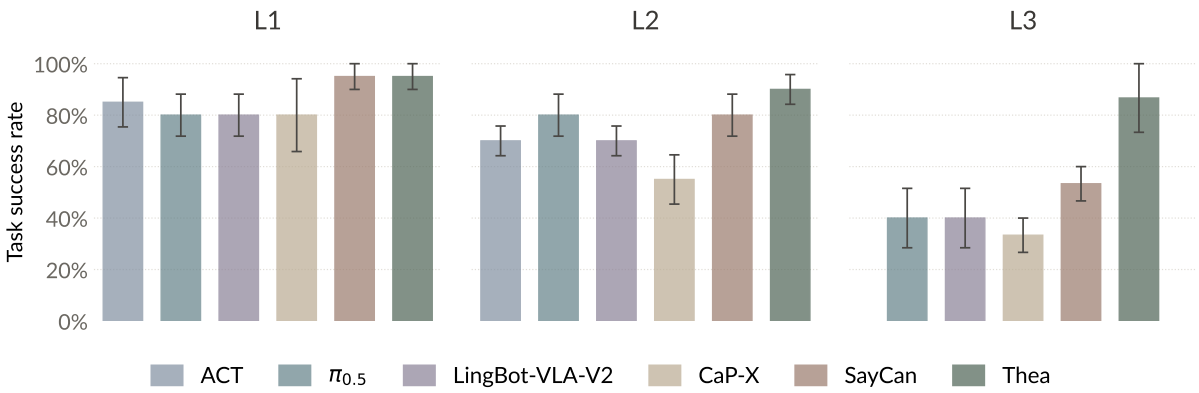


Figure 8. Task success rates of the baselines and Thea across tasks of increasing difficulty. Error bars show one standard error across runs. ACT is omitted from L3 because it is not language-conditioned.

5.2. Scaling with Task Complexity

Figure 8 reports the task success rates of ACT [39], LingBot-VLA-V2 [40], $\pi_{0.5}$ [10], CaP-X [32], SayCan [41], and Thea across L1–L3. Beyond absolute performance, the comparison shows how each architecture scales as task horizon and compositional complexity grow. A trial counts as successful only if every instruction-specified object reaches its target location and the robot completes all required navigation and manipulation stages; outcomes are judged by human annotators from recorded trajectories, independently of the online evaluator.

Thea achieves the highest success rate at all three levels. On L1 the gap among methods is small. On L2, Thea evaluates each grasp and retries the failed ones instead of carrying an invalid sequence forward. On L3, the agent calls `navigate_to` and then `move_base` to bring the robot to a pose better suited to the next manipulation, and adjusts the pose again when a manipulation fails. For the end-to-end policies (ACT, LingBot-VLA-V2, $\pi_{0.5}$), a failure in the middle of a task goes undetected, and the run does not recover. CaP-X generates and revises programs across multiple turns; SayCan selects the next skill with no verdict on the last. The gap that widens from L1 to L3 follows the reliability arithmetic of §2, where per-step failures compound with task length. It also reflects key properties of the harness. The model selects and composes tools as the task requires, and the evaluator closes the loop, making outcomes observable and failures recoverable.

5.3. Evaluator Accuracy

The recovery behind the results above rests on the evaluator, as the analysis of §2 suggests. We therefore measure how reliably the evaluator judges the state of an ongoing physical task. The test set consists of 90 trajectory checkpoints collected from Astribot S1, AgileX Cobot Magic, and Unitree G1, sampled in equal numbers after successful completion, mid-execution, and after unrecovered failures. Two human annotators labeled each checkpoint independently from the complete trajectory, resolving disagreements by replay and discussion. We use Qwen3.7-Plus [42] as the evaluator model throughout our experiments. At test

[42]: Qwen Team (2026), *Qwen3.7-Plus: Multimodal Agent Intelligence*

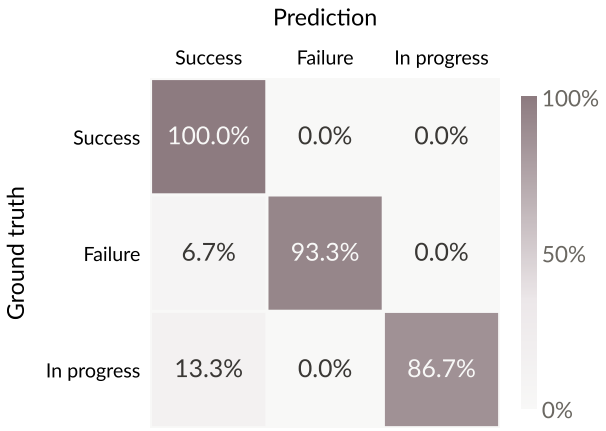


Figure 9. Evaluator predictions against ground-truth success, failure, and in-progress states across three robot embodiments.

time the evaluator sees only its evaluation prompt, including the per-tool post-condition (the expected world state after the call), and the synchronized camera observations.

Figure 9 reports the classification performance for each state separately, with an average accuracy of 93.3%. The remaining errors all run in one direction: 6.7% of failed and 13.3% of in-progress checkpoints are judged successful, while no successful checkpoint is misjudged. These errors are the harmful kind, because a false success either forfeits a needed retry or cuts an action short. Reducing this error mode is an important next step. Substituting the measured accuracy ($\alpha = 0.93$)[1], per-step success $p = 0.8$, and up to $k = 5$ retries into the formulation of §2, a five-step task completes with probability 0.33 in open loop and 0.91 under the harness, broadly consistent with the trend in Figure 8.

1: The three-state average, taken as an approximation of the binary α defined in §2.

5.4. Emergent Capabilities

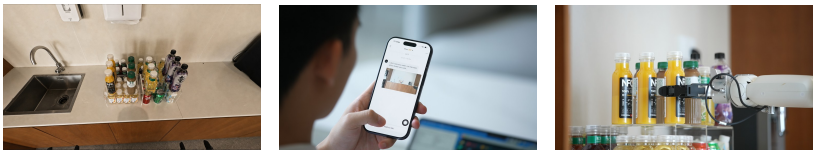
This subsection moves from the fixed tasks above to longer, open-ended tasks in more complex, realistic settings. Thea runs with the full harness, every tool and component available, and we observe what the agent as a whole composes out of them. The demonstrations below sample the capabilities that emerge, and Appendix C provides the execution logs of the demos.

Long-horizon task composition. Across these deployments, Thea carries user requests through to completion over extended sequences of physical interaction. The difficulty concentrates where navigation and manipulation alternate, because each manipulation depends on where the previous move left the robot. Thea composes the alternation from its tools (Figure 10a): it reads the target's position from the scene graph, approaches with `navigate_to`, refines the base pose with `move_base` within the measured clearances, and hands each manipulation's outcome to the evaluator, whose verdict decides whether to adjust and retry or move on. Figure 10 shows representative trajectories, each composed at run time from the same general tools; none follows a script prepared for the task.

(a). The robot executes a long-horizon sequence of navigation and manipulation, composed at run time, to retrieve and deliver a power bank.

(b). The robot actively searches the cabinet drawers to locate the power bank when the target is absent from the scene graph.

(c). The robot uses the evaluator's failure reason to reposition relative to the object and complete the grasp after the initial attempt fails.



(d). The robot queries the user for an alternative when the requested water is unavailable, then retrieves the selected drink.

(e). The three robots share the same harness, with embodiment-specific profiles and tool implementations.

Figure 10. Demonstrations of emergent capabilities enabled by the shared harness across long-horizon task composition, active perception, failure recovery, user interaction, and cross-embodiment portability.

Active perception. Active perception takes two forms. When a requested object appears nowhere in the scene graph, Thea explores the environment. It navigates to a cabinet, opens and inspects its drawers in sequence (Figure 10b), and writes the previously occluded contents into the graph. When the evidence lies outside the current view, Thea adjusts the view instead, calling `tilt_head` to bring the relevant region into the camera. The two operate at different scales, moving the body and moving the camera, but follow the same logic: perception is an action the model chooses, and the new perception informs the next decision.

Failure recovery. Physical execution does not always satisfy the intended post-condition on the first attempt. When it does not, the evaluator returns the failure with a reason, and Thea decides the next call from that reason and the surrounding evidence (Figure 10c), perhaps adjusting

the base pose before retrying, or switching to a tool backed by a different policy. When a retry succeeds, the consolidation at task end writes the recovery into that tool's experience.

User interaction. THEA keeps the user reachable throughout the task. A request may prove vague, a choice may hinge on the user's preference, or progress may be worth surfacing; `query_user` and `notify_user` serve these moments, and the model calls them when the need arises. In the beverage scene (Figure 10d), the user asks for a bottle of water and the scene graph holds none; rather than grasping a substitute of its own choosing, THEA asks the user which of the drinks it does see to bring, and retrieves the one the user selects.

Cross-embodiment portability. The same harness runs on all three embodiments, Astribot S1, AgileX Cobot Magic, and Unitree G1 (Figure 10e). Nothing in the loop refers to a particular body; what is body-specific enters through the Embodiment Profile and the tool implementations behind the registry. Moving to a new robot therefore means writing its profile and its tools, while the loop above them, interpreting requests, selecting tools, weighing evaluator verdicts, carries over unchanged.

6. Related Work

6.1. Coding Agents

Modern coding agents succeed at managing an environment end to end at scale [1, 2, 43]. Their shared design has been systematized as a harness [7]: a reactive loop that re-decides from the latest observed state [44], a tool registry the model selects from at runtime [45], recovery that reads a failure trace and retries [46], managed context [47], and memory that persists across sessions [48]. We transplant these patterns into the physical world, adapting each component along the way (§3). Two properties of the software environment have no physical counterpart. A code repository is readable, and its tests are verifiable by construction. The physical world provides neither, so we build both (§4).

6.2. Embodied Foundation Models

Embodied foundation models supply atomic capability in the physical world along two complementary lines. Vision-Language-Action (VLA) models map observation and instruction end to end to actions, from the RT series [49] through open generalist policies [50] to flow-matching architectures [10, 31]. World Action Models (WAMs) grow out of world models, predictors of how the world evolves, and fold action emission into the prediction network. DreamZero, a recent instance, drives a robot in closed loop from a single image-to-video diffusion model [51]. Both lines are the tools our harness calls, not the layer it replaces. A recent evaluation across control interfaces finds that frontier models mostly fail when driving joints directly but act capably when supervising pretrained policies, with the interface mattering as much as the model itself [12].

[1]: Anthropic (2026), *Claude Code by Anthropic: An Agentic Coding System*
[2]: OpenAI (2026), *Codex: AI Coding Partner from OpenAI*
[43]: Wang et al. (2025), *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*
[44]: Yao et al. (2023), *ReAct: Synergizing Reasoning and Acting in Language Models*
[45]: Schick et al. (2023), *Toolformer: Language Models Can Teach Themselves to Use Tools*
[46]: Shinn et al. (2023), *Reflexion: Language Agents with Verbal Reinforcement Learning*
[47]: Packer et al. (2023), *MemGPT: Towards LLMs as Operating Systems*
[48]: Wang et al. (2024), *Voyager: An Open-Ended Embodied Agent with Large Language Models*
[49]: Zitkovich et al. (2023), *Rt-2: Vision-language-action models transfer web knowledge to robotic control*
[50]: Kim et al. (2024), *Openvla: An open-source vision-language-action model*
[51]: Ye et al. (2026), *World action models are zero-shot policies*

Today's best policies succeed on a single step with probability p of roughly 0.8 to 0.9 [10, 50], so an n-step task succeeds with only p^n (§2), which collapses even for modest n; recovery and evaluation at the harness level close this gap without waiting for single-step perfection. This is why our design separates atomic capability from orchestration explicitly.

6.3. Orchestration for Embodied Agents

Language-model orchestration over robot skills begins open-loop. SayCan selects affordance-grounded skills step by step but nothing judges the skill just executed [41], and Code as Policies emits a one-shot program [52]. Inner Monologue comes closest to closing the loop, threading environment feedback back through the language model, but the feedback channel is hand-picked and there is no general mechanism to decide whether a step has actually succeeded [53]. Later work supplies individual pieces of the loop, a scene graph the planner reads [23] and a failure judge that rules on outcomes [54]. Dual-system VLAs take a different route and pair a slow vision-language reasoner with a fast low-level controller [55–58]. Most recently, concurrent systems bring coding agents to robotics: RoboClaw automates data collection through self-resetting action pairs [59], CaP-X benchmarks coding agents on manipulation and improves them by scaling test-time interaction [32], Guava searches the harness design space for manipulation and distills the result into a compact model for deployment [60], ENPIRE has coding agents self-improve policies on real robots through automated reset, rollout, and verification [61], and ASPIRE discovers reusable skills by writing and repairing control code [62]. Our work carries the paradigm of coding agents over as a whole. The harness is itself the deployed system, closing the loop from instruction to completion in the physical world.

7. Discussion

This work took an exploratory step towards the harness of embodied agents, realizing the paradigm of coding agents as one working system, Thea. Each component was adapted to the physical world, from the loop and tools to context and memory. The two properties the world does not grant, readability and verifiability, were rebuilt as the scene graph and the evaluator.

We offer Thea as evidence that embodied AI is entering the paradigm shift that software agents have already undergone, where the layer between the model and its environment has come to matter as much as the model itself [5, 7, 15]. In fact, an embodied agent needs such a system, rather than just a capable model, even more than a coding agent does, as what it faces is far more complex. The world changes and surprises on its own schedule, observations are always partial, executions fail and cannot be rolled back, and the body's limits shape what is possible.

We believe this shift will fundamentally change how embodied agents are built:

- Reliability becomes a systems problem. It can be engineered in the harness without touching model weights, and the gain compounds

[41]: Ahn et al. (2022), *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances*

[52]: Liang et al. (2023), *Code as policies: Language model programs for embodied control*

[53]: Huang et al. (2022), *Inner monologue: Embodied reasoning through planning with language models*

[23]: Rana et al. (2023), *Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning*

[54]: Duan et al. (2024), *Aha: A vision-language-model for detecting and reasoning over failures in robotic manipulation*

[55]: Bjorck et al. (2025), *Gr00t n1: An open foundation model for generalist humanoid robots*

[56]: Figure AI (2025), *Helix: A Vision-Language-Action Model for Generalist Humanoid Control*

[57]: Shi et al. (2025), *Hi Robot: Open-Ended Instruction Following with Hierarchical Vision-Language-Action Models*

[58]: Gemini Robotics Team (2025), *Gemini Robotics: Bringing AI into the Physical World*

[59]: Li et al. (2026), *RoboClaw: An Agentic Framework for Scalable Long-Horizon Robotic Tasks*

[32]: Fu et al. (2026), *CaP-X: A framework for benchmarking and improving coding agents for robot manipulation*

[60]: Liu et al. (2026), *Guava: An Effective and Universal Harness for Embodied Manipulation*

[61]: Xiao et al. (2026), *ENPIRE: Agentic Robot Policy Self-Improvement in the Real World*

[62]: Lu et al. (2026), *ASPIRE: Agentic /Skills Discovery for Robotics*

[5]: Lopopolo (2026), *Harness Engineering: Leveraging Codex in an Agent-First World*

[7]: Weng (2026), *Harness Engineering for Self-Improvement*

[15]: Yang et al. (2024), *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*

with the model's own progress, since a stronger model raises what the same harness delivers.
- Capability becomes additive. A new policy enters as one more callable tool, and a new body as one more embodiment profile, with nothing retrained around them, so policy builders and system builders can advance in parallel.
- The loop becomes a training target. Reading a scene graph, choosing among tools, admitting failure and retrying are today the work of a general language model. They are learnable behaviors, and natural objectives for the next generation of embodied foundation models. VLAs and WAMs enter the loop as tools today, and they can evolve towards the objectives the loop sets.

Necessary as we have argued each component of the harness to be, we do not claim any of the specific implementations to be final. Indeed, each has its own limitations. The scene graph is limited by the perception beneath it. Positions are coarse, and associating observations of the same object over time still admits errors. The evaluator approximates the exit code it replaces, and its verdicts carry false positives and false negatives that a return value does not. And the loop takes time to decide. Every decision passes through a language model, and the system does not yet act in real time.

Each of these limits marks a direction the framework can grow. For the scene graph, we can explore representations of the world that are more efficient and more durable, tracking entities through change, towards a household memory of where things are and who moved them. The evaluator will always carry an error rate, but the error rate can be trained down, with large collections of judged outcomes, with the habit of gathering more evidence before ruling, and in the long run jointly with the policies it judges. A compact evaluation model, trained for this one job and run locally, could then rule on every action without a round trip to a frontier foundation model. A robot, more than a coding agent, has reason to run its models on board, so a compact model trained for the loop, accelerated at the edge, would bring decisions towards real time. Memory also changes what the agent is to the people it serves. It can grow more personalized and more proactive over time, learning a household's habits and offering help before being asked. Such familiarity raises new questions of privacy, trust, and initiative that deserve as much care as capability. And deployment itself feeds improvement. Interaction with the real world produces exactly what policy learning consumes, ready for the agentic self-improvement engines built by recent work [61, 62], closing a flywheel from experience to policies to stronger single steps. We envision Thea growing with each turn of this flywheel, into part of the infrastructure that embodied agents take for granted.

[61]: Xiao et al. (2026), *ENPIRE: Agentic Robot Policy Self-Improvement in the Real World*
[62]: Lu et al. (2026), *ASPIRE: Agentic /Skills Discovery for Robotics*

References

[1] Anthropic. *Claude Code by Anthropic: An Agentic Coding System*. https://www.anthropic.com/product/claude-code. Accessed: 2026-05-28. 2026 (cited on pages 3, 26).

[2] OpenAI. *Codex: AI Coding Partner from OpenAI*. https://openai.com/codex/. Accessed: 2026-05-28. 2026 (cited on pages 3, 26).

[3] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, et al. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *International Conference on Learning Representations (ICLR)*. 2024 (cited on page 3).

[4] Mitchell Hashimoto. *My AI Adoption Journey*. https://mitchellh.com/writing/my-ai-adoption-journey. Blog post, February 5, 2026. Feb. 2026 (cited on page 3).

[5] Ryan Lopopolo. *Harness Engineering: Leveraging Codex in an Agent-First World*. Feb. 2026. URL: https://openai.com/index/harness-engineering/ (visited on 06/11/2026) (cited on pages 3, 27).

[6] Anthropic. *How the Agent Loop Works*. 2026. URL: https://code.claude.com/docs/en/agent-sdk/agent-loop (visited on 06/20/2026) (cited on page 3).

[7] Lilian Weng. *Harness Engineering for Self-Improvement*. July 2026. URL: https://lilianweng.github.io/posts/2026-07-04-harness/ (visited on 06/11/2026) (cited on pages 3, 26, 27).

[8] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, et al. "RT-1: Robotics Transformer for Real-World Control at Scale". In: *Robotics: Science and Systems (RSS)*. Robotics: Science and Systems Foundation, 2023 (cited on page 3).

[9] Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, et al. "Octo: An Open-Source Generalist Robot Policy". In: *arXiv preprint arXiv:2405.12213* (2024) (cited on page 3).

[10] Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, et al. "$\pi_{0.5}$: A Vision-Language-Action Model with Open-World Generalization". In: *Conference on Robot Learning (CoRL)*. 2025 (cited on pages 3, 19, 22, 23, 26, 27, 34).

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018 (cited on page 5).

[12] Shmuel Berman, Michael Ilie, Jia Deng, and Daniel Freeman. *Claude Plays Robotics*. 2026. URL: https://www.anthropic.com/research/claude-plays-robotics (visited on 07/14/2026) (cited on pages 5, 9, 26).

[13] Anthropic. *Effective Context Engineering for AI Agents*. Sept. 29, 2025. URL: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents (visited on 07/07/2026) (cited on pages 8, 10, 13, 17).

[14] Anthropic. *Introducing the Model Context Protocol*. Nov. 25, 2024. URL: https://www.anthropic.com/news/model-context-protocol (visited on 07/05/2026) (cited on pages 11, 12).

[15] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024 (cited on pages 11, 27).

[16] Anthropic. *Building Effective Agents*. Dec. 19, 2024. URL: https://www.anthropic.com/engineering/building-effective-agents (visited on 07/05/2026) (cited on pages 11, 15).

[17] Anthropic. *Writing Effective Tools for AI Agents—Using AI Agents*. Sept. 11, 2025. URL: https://www.anthropic.com/engineering/writing-tools-for-agents (visited on 07/05/2026) (cited on pages 11, 14).

[18] Anthropic. *Equipping Agents for the Real World with Agent Skills*. Oct. 16, 2025. URL: https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills (visited on 07/05/2026) (cited on page 13).

[19] Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, et al. "From Exploration to Mastery: Enabling LLMs to Master Tools via Self-Driven Interactions". In: *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2025 (cited on page 13).

[20] Anthropic. *Claude Code Sandboxing*. Oct. 20, 2025. URL: https://www.anthropic.com/engineering/claude-code-sandboxing (visited on 07/08/2026) (cited on page 15).

[21] Michael Ahn, Debidatta Dwibedi, Chelsea Finn, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Karol Hausman, et al. "Autort: Embodied foundation models for large scale orchestration of robotic agents". In: *arXiv preprint arXiv:2401.12963* (2024) (cited on page 15).

[22] Qiao Gu, Ali Kuwajerwala, Sacha Morin, Krishna Murthy Jatavallabhula, Bipasha Sen, Aditya Agarwal, et al. "Conceptgraphs: Open-vocabulary 3d scene graphs for perception and planning". In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 5021–5028 (cited on page 16).

[23] Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. "Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning". In: *arXiv preprint arXiv:2307.06135* (2023) (cited on pages 16, 27).

[24] Takuma Yoneda, Jiading Fang, Peng Li, Huanyu Zhang, Tianchong Jiang, Shengjie Lin, et al. "Statler: State-maintaining language models for embodied reasoning". In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 15083–15091 (cited on page 16).

[25] Iro Armeni, Zhi-Yang He, JunYoung Gwak, Amir R Zamir, Martin Fischer, Jitendra Malik, et al. "3d scene graph: A structure for unified semantics, 3d space, and camera". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 5664–5673 (cited on page 17).

[26] Antoni Rosinol, Arjun Gupta, Marcus Abate, Jingnan Shi, and Luca Carlone. "3D dynamic scene graphs: Actionable spatial perception with places, objects, and humans". In: *arXiv preprint arXiv:2002.06289* (2020) (cited on page 17).

[27] Ayça Takmaz, Elisabetta Fedele, Robert W. Sumner, Marc Pollefeys, Federico Tombari, and Francis Engelmann. "OpenMask3D: Open-Vocabulary 3D Instance Segmentation". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023 (cited on page 17).

[28] Haokun Zhu, Zongtai Li, Zihan Liu, Kevin Guo, Zhengzhi Lin, Yuxin Cai, et al. "SysNav: Multi-Level Systematic Cooperation Enables Real-World, Cross-Embodiment Object Navigation". In: *IEEE Robotics and Automation Letters* (2026) (cited on pages 17, 22, 34).

[29] Hanxiao Jiang, Binghao Huang, Ruihai Wu, Zhuoran Li, Shubham Garg, Hooshang Nayyeri, et al. "RoboEXP: Action-Conditioned Scene Graph via Interactive Exploration for Robotic Manipulation". In: *Conference on Robot Learning (CoRL)*. 2024 (cited on page 17).

[30] Anthropic. *Demystifying evals for AI agents*. Jan. 9, 2026. URL: https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents (visited on 07/29/2026) (cited on page 19).

[31] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, et al. "π_0: A Vision-Language-Action Flow Model for General Robot Control". In: *CoRR* abs/2410.24164 (2024). DOI: 10.48550/ARXIV.2410.24164 (cited on pages 19, 26).

[32] Max Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, et al. "CaP-X: A framework for benchmarking and improving coding agents for robot manipulation". In: *arXiv preprint arXiv:2603.22435* (2026) (cited on pages 19, 22, 23, 27, 34).

[33] Bertrand Meyer. "Applying'design by contract'". In: *Computer* 25.10 (2002), pp. 40–51 (cited on page 19).

[34] Anthropic. *How We Built Claude Code Auto Mode: A Safer Way to Skip Permissions*. Mar. 25, 2026. URL: https://www.anthropic.com/engineering/claude-code-auto-mode (visited on 07/14/2026) (cited on page 19).

[35] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, et al. "Large Language Models Cannot Self-Correct Reasoning Yet". In: *International Conference on Learning Representations (ICLR)*. 2024 (cited on page 20).

[36] Arjun Panickssery, Samuel R. Bowman, and Shi Feng. "LLM Evaluators Recognize and Favor Their Own Generations". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024 (cited on page 20).

[37] Zeyi Liu, Arpit Bahety, and Shuran Song. "REFLECT: Summarizing Robot Experiences for Failure Explanation and Correction". In: *Conference on Robot Learning (CoRL)*. 2023 (cited on page 20).

[38] Yanjiang Guo, Yen-Jen Wang, Lihan Zha, and Jianyu Chen. "Doremi: Grounding language model by detecting and recovering from plan-execution misalignment". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2024 (cited on page 20).

[39] Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware". In: *Robotics: Science and Systems (RSS)*. 2023 (cited on pages 22, 23, 34).

[40] Wei Wu, Fangjing Wang, Fan Lu, He Sun, Shi Liu, Yunnan Wang, et al. "From Foundation to Application: Improving VLA Models in Practice". In: *arXiv preprint arXiv:2607.06403* (2026) (cited on pages 22, 23).

[41] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, et al. "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances". In: *Conference on Robot Learning (CoRL)*. 2022 (cited on pages 22, 23, 27).

[42] Qwen Team. *Qwen3.7-Plus: Multimodal Agent Intelligence*. May 2026. URL: https://qwen.ai/blog?id=qwen3.7-plus (cited on page 23).

[43] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, et al. "OpenHands: An Open Platform for AI Software Developers as Generalist Agents". In: *International Conference on Learning Representations (ICLR)*. 2025 (cited on page 26).

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, et al. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *International Conference on Learning Representations (ICLR)*. 2023 (cited on page 26).

[45] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, et al. "Toolformer: Language Models Can Teach Themselves to Use Tools". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023 (cited on page 26).

[46] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. "Reflexion: Language Agents with Verbal Reinforcement Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023 (cited on page 26).

[47] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, et al. "MemGPT: Towards LLMs as Operating Systems". In: *arXiv preprint arXiv:2310.08560* (2023) (cited on page 26).

[48] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, et al. "Voyager: An Open-Ended Embodied Agent with Large Language Models". In: *Transactions on Machine Learning Research (TMLR)* (2024) (cited on page 26).

[49] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, et al. "Rt-2: Vision-language-action models transfer web knowledge to robotic control". In: *Conference on Robot Learning (CoRL)*. PMLR. 2023, pp. 2165–2183 (cited on page 26).

[50] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, et al. "Openvla: An open-source vision-language-action model". In: *arXiv preprint arXiv:2406.09246* (2024) (cited on pages 26, 27).

[51] Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, et al. “World action models are zero-shot policies”. In: *arXiv preprint arXiv:2602.15922* (2026) (cited on page 26).

[52] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, et al. “Code as policies: Language model programs for embodied control”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 9493–9500 (cited on page 27).

[53] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, et al. “Inner monologue: Embodied reasoning through planning with language models”. In: *arXiv preprint arXiv:2207.05608* (2022) (cited on page 27).

[54] Jiafei Duan, Wilbert Pumacay, Nishanth Kumar, Yi Ru Wang, Shulin Tian, Wentao Yuan, et al. “Aha: A vision-language-model for detecting and reasoning over failures in robotic manipulation”. In: *arXiv preprint arXiv:2410.00371* (2024) (cited on page 27).

[55] Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, et al. “Gr00t n1: An open foundation model for generalist humanoid robots”. In: *arXiv preprint arXiv:2503.14734* (2025) (cited on page 27).

[56] Figure AI. *Helix: A Vision-Language-Action Model for Generalist Humanoid Control*. https://www.figure.ai/news/helix. Accessed: 2026-07-25. Feb. 2025 (cited on page 27).

[57] Lucy Xiaoyang Shi, Brian Ichter, Michael Robert Equi, Liyiming Ke, Karl Pertsch, Quan Vuong, et al. “Hi Robot: Open-Ended Instruction Following with Hierarchical Vision-Language-Action Models”. In: *International Conference on Machine Learning (ICML)*. Vol. 267. Proceedings of Machine Learning Research. PMLR, 2025, pp. 54919–54933 (cited on page 27).

[58] Gemini Robotics Team. “Gemini Robotics: Bringing AI into the Physical World”. In: *arXiv preprint arXiv:2503.20020* (2025) (cited on page 27).

[59] Ruiying Li, Yunlang Zhou, Yuyao Zhu, Kylin Chen, Jingyuan Wang, Sukai Wang, et al. “RoboClaw: An Agentic Framework for Scalable Long-Horizon Robotic Tasks”. In: *arXiv preprint arXiv:2603.11558* (2026) (cited on page 27).

[60] Haowen Liu, Xirui Li, Shaoxiong Yao, Peng Shi, Tianyi Zhou, Jia-Bin Huang, et al. “Guava: An Effective and Universal Harness for Embodied Manipulation”. In: *arXiv preprint arXiv:2606.18363* (2026) (cited on page 27).

[61] Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian Fu, Haoru Xue, et al. “ENPIRE: Agentic Robot Policy Self-Improvement in the Real World”. In: *arXiv preprint arXiv:2606.19980* (2026) (cited on pages 27, 28).

[62] Runyu Lu, Yubo Wu, Ethan Kou, Max Fu, Wenli Xiao, Ajay Mandlekar, et al. “ASPIRE: Agentic /Skills Discovery for Robotics”. In: *arXiv preprint* (2026) (cited on pages 27, 28).

A. Tool Registry

Table 4 lists the Astribot S1 tool registry. Its entries follow the protocol of §3.3: physical capabilities use deployment backends, while lightweight context and interaction tools run in-process. Slash-separated lists are the admissible values; quoted strings are free-form descriptions; refs like `desk_24` name Scene Graph entities.

Table 4. Tools deployed on the Astribot S1, grouped by function. `evaluate_run` is registered alongside them but invoked by the harness through a post-hook, never selected by the model (§4.2).

Tool	Description	Backend	Parameter	Value
Navigation				
`navigate_to`	Approach an object or location.	SysNav [28]	`target`	e.g. "drink area"/desk_24
`move_base`	Translate or rotate within safety bounds.	Astribot S1 SDK	`direction`	forward/backward/left/ right/turn_left/turn_right
			`distance_m`	0.03–0.80 m
			`angle_deg`	3–30°
Manipulation				
`pick_up`	Grasp a localized object.	CaP-X/ACT/$\pi_{0.5}$ [10, 32, 39]	`obj`	e.g. "orange juice"/bottle_2
`place`	Place the held object on a table or in a box.	CaP-X/ACT/$\pi_{0.5}$	`target`	e.g. "table"/table_2
`open_drawer`	Open a drawer.	CaP-X/ACT/$\pi_{0.5}$	`container_ref`	e.g. cabinet_87
			`drawer_level`	higher/lower
`close_drawer`	Close a drawer.	CaP-X/ACT/$\pi_{0.5}$	`container_ref`	e.g. cabinet_87
			`drawer_level`	higher/lower
`trash_drop`	Drop the held object into a bin.	CaP-X/ACT/$\pi_{0.5}$	`trash_ref`	e.g. trash_can_54
Perception				
`tilt_head`	Tilt the head camera to a given pitch angle.	Astribot S1 SDK	`pitch_deg`	0–60°
`get_object_relations`	Query an object's spatial and semantic relations.	Scene Graph	`obj`	e.g. cup_2
			`relation` *(optional)*	on/inside/holding/near
`get_image`	Retrieve stored images of an object.	Scene Graph	`obj`	e.g. cup_2
Skills				
`load_skill`	Load the instructions of a registered skill.	Thea	`name`	e.g. tidy-workspace
User Interaction				
`query_user`	Request and await clarification.	Lark	`question`	e.g. "No plain water; which drink should I bring?"
			`candidate_refs` *(optional)*	e.g. bottle_3
			`observation_views` *(optional)*	e.g. torso_rgbd
`notify_user`	Send a non-blocking update.	Lark	`message`	e.g. "I found an empty orange-juice bottle on desk..."
			`notification_type` *(optional)*	progress/warning/ completion

B. Context Structure

Section 3.2 introduces three context lifetimes. Listing 11 expands the context items defined in Table 1. The System Prompt appears in full; other entries show structure without task-specific values.

Resident
stable across turns

System Prompt

```
You are Thea, an embodied agent that completes physical tasks
by calling tools.
- Choose one next tool call per turn unless the task is
  complete.
- After each tool result, update the plan from the latest
  evidence.
- Use the Scene Graph Brief for object refs and coarse global
  state. Use the latest Observation for current local
  visibility, alignment, and clearance.
- Treat the latest Observation as current physical evidence
  and earlier tool results as historical execution evidence.
- Follow tool descriptions for preconditions, argument
  semantics, failure modes, and recovery hints.
- Do not invent object identifiers, measurements, action
  outcomes, or user intent. Ask the user when the instruction
  and current evidence are genuinely insufficient.
- Wait for the current tool to finish before choosing the
  next tool.
```

Memory

```
{Preferences: [entry, ...], Conventions: [entry, ...],
  General Lessons: [entry, ...]}
```

Embodiment Profile

```
Operational Envelope: {Base Footprint; Base Mobility;
Reachable Workspace}
Perception Configuration: {Sensor Modalities; Model-Visible
Views}
Base-Relative Positions: {Camera Positions; Initial Gripper
Positions}
```

Tool Definitions

```
[{name, description [+ experience summary],
  inputSchema: {type, properties, [required]}}, ...]
```

Refreshed
replaced before each decision

Scene Graph Brief

```
Graph metadata: freshness=...; provenance=...;
coordinate_frame=...; updated_at=...
Robot: pose=(...); holding=[obj, ...]
Objects: - obj; position=(...); confidence=...;
freshness=...; container_state=...; contents=[obj, ...]
```

Observations

```
Base clearance: forward=...m, backward=...m, left=...m,
right=...m
Visual evidence: view, ...  [image blocks follow]
```

Instructions

```
[{content: instruction}, ...]
```

Task Notes

```
Task summary: Goal=instruction; Current phase=phase
Timeline: task_start; tool: outcome [reason]
```

Model Responses

```
[{content,
  tool_calls[]: {id, type: function, function: {name,
arguments}}
}, ...]
```

Tool Results

```
[{tool_call_id,
  content: {success: true, tool-defined value fields} or
{success: false, reason}
}, ...]
```

```
Compaction may replace an older accumulated prefix; resident
and refreshed context remain unchanged.
```

Accumulated
grows through ordinary turns

Listing 11. Context structure for one decision, the concatenation of the three lifetimes of Table 1 in one model call.

C. Demo Execution Logs

The first three logs were collected on the Astribot S1, and the final log on the Unitree G1 with the same harness. Turn counts follow the definition in §3.1. Final responses contain no tool calls and are therefore shown without turn numbers.

C.1. Beverage Retrieval

The user asks for a bottle of water to be brought to a table. The beverage area contains no unique water bottle, and Thea asks the user to select an available drink. After the user chooses orange juice, the robot delivers it.

User request. "Bring me a bottle of water and place it on the table."

Turn 1. Reasoning: The user requests a drink, so the pantry is the next search location. Navigate there before selecting the target bottle.

```
1  navigate_to(target="pantry")
```

Navigation. The robot reaches the pantry.

Turn 2. Reasoning: The navigation result confirms arrival and alignment at the pantry, and the refreshed clearance evidence permits a 0.80 m forward step. Advance to improve the torso-camera view and manipulation distance.

```
move_base(direction="forward", distance_m=0.80)
```

Turn 3. Reasoning: The fresh torso view shows no unambiguous plain water bottle but includes several alternative drinks. Query the user rather than treating the navigation anchor as the grasp target.

```
query_user(
    question="No plain water; which drink should I bring?")
```

User response. Orange juice.

Turns 4–6. Reasoning summary:[1] The user selects orange juice. Apply two leftward corrections followed by a 0.28 m forward step to bring the selected bottle into the grasp envelope.

Base motions 3 moves

```
move_base(direction="left", distance_m=0.30)
move_base(direction="left", distance_m=0.15)
move_base(direction="forward", distance_m=0.28)
```

Turn 7. Reasoning: The user response and local alignment establish orange juice as the target, but no reliable object reference is available. Bind it by name, execute the grasp, and verify control.

```
pick_up(obj="orange juice")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Grasp verified. The evaluator marks the pick as successful.

Turn 8. Reasoning: The grasp evaluator confirms control of the orange juice. The destination is `desk_24`; navigate there while retaining the object.

```
navigate_to(target="desk_24")
```

Navigation. The robot reaches `desk_24` with the drink.

Turns 9–12. Reasoning summary: The navigation result confirms arrival at `desk_24` with the drink held. Apply a 0.58 m approach followed by 0.05 m, 0.04 m, and 0.03 m corrections to reach the placement pose.

Base motions 4 moves

```
move_base(direction="forward", distance_m=0.58)
move_base(direction="forward", distance_m=0.05)
move_base(direction="forward", distance_m=0.04)
...
```

1: Each `move_base` is one turn, decided after the previous completes. Folded runs show the move count and one reasoning summary, here and in the logs that follow.

The manipulation calls in these logs run self-terminating Coding-as-Policy backends, so their evaluations take the two terminal states of the contract in §4.2.

Turn 13. Reasoning: The orange juice remains controlled, the final base correction has completed, and the destination is `desk_24`. Place the drink there and verify the resulting object state.

```
place(target="table")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Placement verified. The evaluator marks the placement as successful.

Final response. The drink has been delivered.

C.2. Power-Bank Search

The user asks for a power bank. The power bank is not visible in the scene graph, and the robot searches the cabinet drawers in order. After finding the power bank in the lower drawer, the robot delivers it.

User request. "The computer is out of power. Find me a power bank and place it on the desk."

Turn 1. Reasoning: The requested power bank is absent from the visible scene graph. The drawers in `cabinet_87` may contain objects outside the current view, so navigate there and inspect them in order.

```
navigate_to(target="cabinet_87")
```

Navigation. The robot reaches `cabinet_87`.

Turns 2–7. Reasoning summary: The navigation result confirms arrival at `cabinet_87`. Apply a 0.55 m approach, a 0.17 m rightward correction, and four centimeter-scale adjustments to enter the drawer manipulation pose.

Base motions 6 moves

```
move_base(direction="forward", distance_m=0.55)
move_base(direction="right", distance_m=0.17)
move_base(direction="forward", distance_m=0.05)
...
```

Turn 8. Reasoning: The cabinet is now within manipulation range. Start with the upper drawer and open it to inspect its contents.

```
open_drawer(drawer_level="higher", container_ref="cabinet_87")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Upper drawer opened. The evaluator verifies that the upper drawer opens successfully.

Turn 9. Reasoning: The refreshed head-camera observation shows the upper drawer open and the power bank absent. Close the searched compartment before continuing to avoid leaving an obstruction.

```
close_drawer(
    drawer_level="higher",
    container_ref="cabinet_87")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Upper drawer closed. The closure is verified successfully.

Turn 10. Reasoning: The upper drawer closure is verified. Continue the ordered search by opening the lower drawer and exposing its contents.

```
open_drawer(drawer_level="lower", container_ref="cabinet_87")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Lower drawer opened. The evaluator verifies that the lower drawer opens successfully.

Turn 11. Reasoning: The refreshed head-camera observation shows the power bank inside the open lower drawer, and the gripper is empty. The object is reachable, so pick it and verify control.

```
pick_up(obj="power bank")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Pickup verified. The wrist observation confirms that the power bank is held securely.

Turn 12. Reasoning: The pickup evaluation confirms that the power bank is held, and the updated scene state removes it from the cabinet. Close the lower drawer before leaving the search location.

```
close_drawer(drawer_level="lower", container_ref="cabinet_87")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Lower drawer closed. The cabinet search concludes with both compartments closed.

Turn 13. Reasoning: The power bank is held and both drawers are closed. The destination is `desk_96`; navigate there while retaining the object.

```
navigate_to(target="desk_96")
```

Navigation. The robot reaches `desk_96` while holding the power bank.

Turns 14–16. Reasoning summary: The navigation result confirms arrival at the destination with the power bank held. Apply a 0.80 m approach followed by two 0.03 m corrections to enter the surface-placement envelope.

Base motions 3 moves

```
move_base(direction="forward", distance_m=0.80)
move_base(direction="forward", distance_m=0.03)
move_base(direction="forward", distance_m=0.03)
```

Turn 17. Reasoning: The final base correction has completed, the scene still marks the power bank as held, and the destination is `desk_96`. Place the object on its surface and verify the result.

```
place(target="table")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Placement verified. The power bank is placed successfully on `desk_96`.

Final response. The power bank has been delivered.

C.3. Desk Cleanup

This episode starts without a user instruction. A standing monitor waits until the workstation is unoccupied while objects remain on the desk. Thea loads the workspace-cleaning rules, identifies an empty orange-juice bottle as trash while leaving the laptop and charging power bank untouched, and sends one non-blocking notification before acting. The robot then drops the bottle into the trash bin, and Thea sends a completion notification.

`monitor_scene` is an ordinary tool, the standing counterpart of `query_user`: where `query_user` waits for the user to answer, `monitor_scene` waits for the world to change. The agent gives it a natural-language condition, and the call returns once the condition is observed to hold. The returned event then takes the place of a user instruction, giving the task its goal. Only this log uses the tool.

Standing monitor condition. "The workstation is unoccupied, and objects remain on the desk."

Turn 1. Reasoning: Keep the desk unchanged while the workstation is occupied. Set up the standing monitor and wait until the workstation is unoccupied while items remain on the desk.

```
monitor_scene(
    condition="workstation unoccupied and objects on desk")
```

Monitor event. The user is absent. The desk contains an empty orange-juice bottle, a laptop connected to a power bank, and several other items.

Turn 2. Reasoning: The returned monitor event identifies several desk objects, but visual presence alone does not justify discarding them. Load the workspace-cleaning rules.

```
load_skill(name="tidy-workspace")
```

Classification. The confirmed empty bottle is trash. The laptop, charging power bank, and items whose state is uncertain must remain untouched.

Turn 3. Reasoning: The rules and current observation identify only the empty orange-juice bottle as discardable; the remaining items must stay untouched. Notify the user, then begin the cleanup.

```
notify_user(
    message="I found an empty orange-juice bottle on your
             desk. I will discard it and leave the other
             items untouched.")
```

Notification. The one-way message is delivered, and execution continues without awaiting a reply.

Turns 4–7. Reasoning summary: The target is fixed and the approach is clear. Advance 0.20 m, tilt the head by 45 degrees to refresh the local evidence, then apply 0.05 m and 0.03 m corrections to bring the bottle into the pickup envelope.

Base motions 4 actions

```
move_base(direction="forward", distance_m=0.20)
tilt_head(pitch_deg=45)
move_base(direction="forward", distance_m=0.05)
...
```

Turn 8. Reasoning: The refreshed view and incremental alignment place the confirmed empty bottle within reach while the gripper remains empty. Execute the grasp and verify control.

```
pick_up(obj="empty orange-juice bottle")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Pickup verified. The wrist observations show the bottle secured between the gripper fingers.

Turn 9. Reasoning: The grasp evaluator shows the empty bottle secured between the gripper fingers. The disposal destination is `trash_can_54`; navigate there while retaining the bottle.

```
navigate_to(target="trash_can_54")
```

Navigation. The robot reaches `trash_can_54` with the bottle.

Turns 10–12. Reasoning summary: The navigation result confirms arrival in the target area, but the bin is not yet centered for release. Apply a 0.55 m approach, a 15-degree left turn, and a final 0.05 m correction to reach the release pose.

Base motions 3 moves

```
move_base(direction="forward", distance_m=0.55)
move_base(direction="turn_left", angle_deg=15)
move_base(direction="forward", distance_m=0.05)
```

Turn 13. Reasoning: The empty bottle remains held, `trash_can_54` is unambiguous, and the final alignment succeeds. Release the bottle into the bin and verify the resulting state.

```
trash_drop(trash_ref="trash_can_54")
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Drop verified. The bin opening is centered, both grippers are empty, and no bottle is visible outside the bin.

Turn 14. Reasoning: The drop evaluator confirms that both grippers are empty and no bottle is visible outside the bin, satisfying the cleanup goal. Notify the user that the bottle has been discarded.

```
notify_user(
    message="The empty orange-juice bottle has been
             discarded.")
```

Completion notification. The cleanup result is delivered to the user.

Final response. The empty orange-juice bottle has been discarded.

C.4. Basket Delivery

The user (Jason) asks for snacks to be brought to friends. The robot packs snacks into a basket, carries it into the living room, walks up to Jason, and hands over the basket.

Routine low-level calls (controller checks, camera refreshes, setup queries, and the locomotion steps inside navigation) are folded and excluded from the turn numbering.

User request. "I am about to meet some friends. Bring some snacks over for them."

Turn 1. Reasoning: The workspace is ready. Place the snack in the basket and wait for the manipulation to finish before lifting the basket.

```
pack_snacks()
```

Packing workflow repeated controller updates folded

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Placement verified. The snack is packed in the basket.

Turn 2. Reasoning: Packing is complete. Lift the loaded basket, retract the arm for walking, and prepare it for navigation.

```
lift_basket()
```

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Lift verified. The basket is ready to be carried.

Turn 3. Reasoning: The basket is secured for transport. Follow the doorway route into the living room before selecting a recipient.

```
navigate_to(living_room)
```

Navigation. G1 reaches the living room while holding the basket.

Turn 4. Reasoning: The living-room view shows Jason seated in a white shirt and light trousers. Navigate to him while retaining the basket.

```
navigate_to(Jason)
```

Navigation. G1 brings Jason into front view while retaining the basket.

Turns 5–8. Reasoning summary: The recipient is now in front but not yet close enough for the handoff. Continue in short increments.

Carry motions 4 moves

```
basket_carry_move(direction="forward", distance_m=3.0)
basket_carry_move(direction="forward", distance_m=1.5)
...
```

Navigation. Four short forward motions bring G1 close to the seated recipient.

Turn 9. Reasoning: The recipient is close in front of G1, and the basket remains held. Extend the arm to the handoff pose and open the hand, but do not infer transfer from the controller state alone.

```
hand_over()
```

Handoff. The right arm extends, and the hand opens successfully.

Evaluator. The harness invokes `evaluate_run()` after the manipulation.
Release verified. The basket is released and rests securely with the recipient.