Nautilus: From One Prompt to Plug-and-Play Robot Learning

Yufeng Jin[1*] Jianfei Guo[2*] Xiaogang Jia[4] Yu Deng[1] Zechu Li[1] Han Liu[3]
Weiran Liao[4] Vignesh Prasad[1] Mathias Franzius[5] Gerhard Neumann[4,6]
Georgia Chalvatzaki[1,7,8†]

Abstract

Robot learning research is fragmented across policy families, benchmark suites, and real robots; Each implementation is entangled with the others in a complex combination matrix, making it an engineering nightmare to port any single element. General-purpose coding agents may occasionally bridge specific setups, but cannot close this gap at scale because they lack the procedural priors and validation practices that characterize robotics research workflows. We propose NAUTILUS, an open-source harness that turns a single user prompt —for example, "Evaluate policy A with benchmark B" — into ready-to-use reproduction, evaluation, fine-tuning, and deployment workflows. NAUTILUS provides: plug-and-play agent skill sets with distilled priors from robotics research; typed contracts among policies, simulators/benchmarks, and real-world robots; unified interfaces and execution environments; and a trustworthy agentic coding workflow with explicit, automated validation, and testing at each milestone. NAUTILUS can not only automatically generate the required adapters and containers for existing implementations, but also wrap and onboard new or user-provided policies, simulators/benchmarks, and robots, all connected via a uniform interface. This expands cross-validation coverage without hand-written glue code. Like a nautilus shell that grows by adding chambers, NAUTILUS scales by extending its execution in chambered units, making it a research harness for scalability rather than a hand-curated framework, and aiming to reduce the engineering burden of cross-family reproduction and evaluation in the ever-growing robot learning ecosystem. Project website: https://yufengjin.github.io/nautilus/.

1 Introduction

Large Language Models (LLMs) are now widely used in code generation and software-engineering workflows [1, 2]. In robotics, LLMs have already been explored in task-level embodied pipelines, including policy code generation and reward design [3, 4]. Our setting is different: we study agentic coding systems as orchestrators of robot learning research workflows across repositories, simulators, benchmarks, and hardware interfaces, where even strong coding agents still often fail to reach an end-to-end reproduced result [5, 6]. We argue that this persistent failure is not primarily a model-capability problem, but a **harness problem**. *Harness engineering*—the design of everything around an LLM except the model itself, including orchestration, tools, verification protocols, guardrails, state, and procedural priors—has recently been identified as a central determinant of agentic project success [5–8]. Robot learning still lacks a general-purpose research harness.

*Equal contribution.
†[1]TU Darmstadt; [2]Chaser Robotics GmbH; [3]Independent Researcher; [4]KIT; [5]Honda Research Institute Europe; [6]FZI; [7]Hessian.AI; [8]Robotics Institute Germany.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

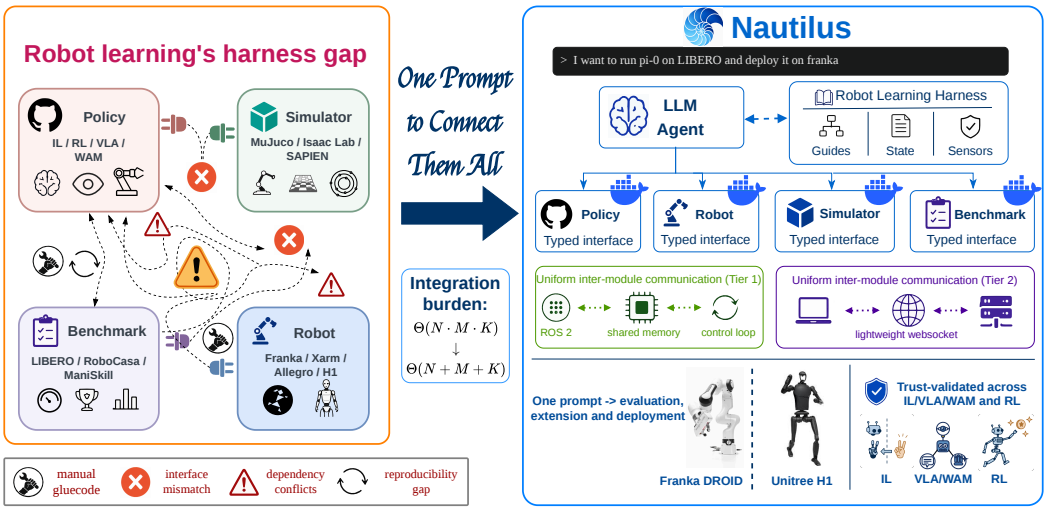


Figure 1: **The harness gap in robot learning research, and how NAUTILUS closes it.** Existing workflows repeatedly rebuild policy–benchmark–robot glue, yielding a field-scale integration burden of $\Theta(N \cdot M \cdot K)$. NAUTILUS introduces a shared robot learning harness: a substrate of typed contracts, chambered execution, and uniform transport, plus a content layer of Guides, Sensors, and State. This changes the unit of work from pairwise integration to reusable onboarding, reducing the burden to $\Theta(N+M+K)$.

Robot learning research spans three largely independent axes: policy families P (e.g., vision-language-action models (VLAs), imitation learning (IL), reinforcement learning (RL), and world action model (WAM)), benchmark suites B (e.g., LIBERO [9], RoboCasa [10], ManiSkill [11], ALOHA [12], etc.), and robot embodiments R (e.g., single-arm, bimanual, dexterous-hand, locomotion, humanoid, etc.), with cardinalities N, M, and K, respectively. Each non-trivial (P, B, R) cross-comparison typically requires a distinct hand-written integration layer, so docker container setup, observation adapters, smoke tests, and trust-validation procedures are repeatedly re-implemented across papers, labs, and reproduction efforts. At the field level, this induces a harness cost of $\Theta(N \cdot M \cdot K)$, whereas a shared harness would amortize the same burden to $\Theta(N+M+K)$. Existing systems address only part of the problem. Robot learning frameworks such as LeRobot [13], Isaac Lab [14], and robosuite [15] consolidate common workflows but remain coupled to fixed policies and benchmarks, while general-purpose coding agents such as Claude Code [16], OpenHands [17], and SWE-agent [18] provide the agent loop but not the robotics-specific procedural knowledge these workflows require.

We propose NAUTILUS, which to our knowledge is the first domain-specific harness for robot learning research. Unlike a conventional robot learning framework, NAUTILUS is an agent-facing harness over typed policy, benchmark, and robot modules, while the underlying coding-agent loop is treated as replaceable. NAUTILUS is organized into two layers. The first is a *substrate* built on three engineering invariants—typed interface contracts, chambered execution, and a uniform inter-module transport. This substrate is a prerequisite: without typed contracts and chambered isolation, the agent has no stable surface to reason over and must relearn per-project glue code for each new (P, B, R) triplet. On top of this substrate sits a *content layer* that instantiates the Guides+Sensors framework of harness engineering [5] and extends it with State. In our formulation, **Guides** shape the agent's behavior before code generation, **Sensors** validate its outputs after generation, and **State**—an MCP-served, JSON-schema-validated contract registry—mediates between modules, reducing $(P \times B)$ integration cost from $\Theta(N \cdot M)$ to $\Theta(N+M)$. Figure 1 illustrates the harness gap in robot learning and how NAUTILUS closes it.

Our contributions are fourfold. **1)** We introduce a modular substrate based on typed interface contracts, chambered execution, and uniform inter-module transport, collapsing $(P \times B \times R)$ integration cost from $\Theta(N \cdot M \cdot K)$ to $\Theta(N+M+K)$ and giving the agent a stable, policy-agnostic surface to reason over. **2)** We distill the procedural priors of robotics research into reusable agent skills, turning tacit workflow knowledge into explicit guidance. **3)** We release NAUTILUS together with structured agentic workflows and evaluation tools for reproduction, cross-evaluation, finetuning, and deployment across four policy families (IL, VLA, RL, and WAM), six benchmark suites, and four embodiment classes; on real hardware, `nautilus-collect` provides evidence that the same Robot contract carries across

Table 1: **Comparison of NAUTILUS to representative robot learning platforms.** Rows are grouped into policy families, simulation benchmarks integrated under the typed Policy/Benchmark contracts, real-robot hardware interfaces (by embodiment class), and contracts demonstrably driven end-to-end by an LLM coding agent under our reproduction-validated protocol (§4.2). Several peer platforms expose stable ABCs that an expert programmer can extend; in the bottom block we reserve ✓ for contracts that an LLM agent has executed end-to-end against a verified registry. ✓ indicates first-class support, ✗ no support, ○ ad-hoc or community-contributed support without a stable interface.

Capability	**NAUTILUS**	LeRobot [13]	RoboTwin Platform [19]	robomimic [20]	robosuite [15]	Isaac Lab [14]	Lab-Arena [21]
Policy families							
Imitation learning (IL)	✓	✓	✓	✓	✗	○	✓
Vision-Language-Action (VLA)	✓	✓	✓	✗	✗	✗	✓
World Action Model (WAM)	✓	✗	✗	✗	✗	✗	✗
Reinforcement learning (RL)	✓	✓	○	✗	✓	✓	✓
Benchmarks (simulation)							
RoboCasa [10]	✓	✗	✗	○	✗	✗	✗
LIBERO [9]	✓	✓	✗	✗	✗	✗	✗
ManiSkill [11]	✓	✗	✗	✗	✗	✗	✗
RoboTwin [19]	✓	✗	✓	✗	✗	✗	✗
ALOHA [12]	✓	✗	✗	✗	✗	✗	✗
Real-robot hardware interfaces							
Single-arm	✓	✓	✓	✗	✗	✗	✗
Bimanual	✓	✓	✓	✗	✗	✗	✗
Dexterous hand	✓	✗	✗	✗	✗	✗	✗
Humanoid	✓	✓	✗	✗	✗	○	✗
Automatically extendable to							
New policies	✓	✗	✗	✗	✗	✗	✗
New benchmarks	✓	✗	✗	✗	✗	✗	✗
New robot interfaces	○	✗	✗	✗	✗	✗	✗

embodiments, with deployment validated on Franka and Unitree H1. **4)** We design NAUTILUS as an extensible community platform where researchers can reproduce and evaluate policies or benchmarks without modifying framework code, while new contributions enter the registry through 3–5 typed method contracts with commit- and image-pinned reproducibility and trust validation.

2 Related Work

LLM coding agents and harness engineering. Code-generating LLMs have evolved from prompt-centric methods to tool use and, more recently, to full agent loops [22–24, 18, 16, 17]. As these systems have matured, the main bottleneck has shifted from prompting to the surrounding scaffolding, formalized as *harness engineering* [5, 6]. Recent work studies this layer through abstractions such as Guides and Sensors and through unified evaluation scaffolds [5, 25, 6–8, 26, 27]. These systems provide strong general-purpose harnesses, but not the robotics-specific priors needed for robot learning workflows. NAUTILUS specializes this harness perspective through typed interfaces, chambered execution, and workflow-level validation.

Robot learning paradigms. Robot learning spans several policy families. *Imitation learning* (IL) fits policies to expert trajectories [12, 28–36], while *reinforcement learning* (RL) optimizes behavior through reward-driven interaction with the environment [37–41]. More recently, *vision-language-action* (VLA) models have emerged as a broader architectural class that maps multimodal inputs to robot actions [42–60]; most VLAs are trained with IL-style objectives, though recent work also uses RL fine-tuning [61, 62]. A related direction is the *world action model* (WAM), which routes action prediction through a learned dynamics model [63–74]. These families differ in training and deployment assumptions, but NAUTILUS targets their workflows rather than any single model class.

LLMs in robotics and embodied orchestration. Recent work has applied LLMs and VLMs inside robotics pipelines for policy code generation, reward design, and embodied task orchestration [3, 4, 75, 76]. These systems place an LLM within a policy, planning, or environment loop. NAUTILUS addresses a different layer of the stack: not execution-time orchestration for a single robot task, but research-time orchestration across repositories, simulators, benchmarks, and robot interfaces.

Benchmarks, infrastructure, and deployment stacks. Robot learning research is supported by a broad ecosystem of simulator and policy frameworks [13–15, 77, 20, 78–82], benchmark suites [9–11, 83–85, 19, 86], and large-scale datasets [87–89]. Recent integrations such as Isaac Lab-Arena [21] and evaluation suites such as RoboArena [90] and MolmoSpaces [91] connect parts of this stack

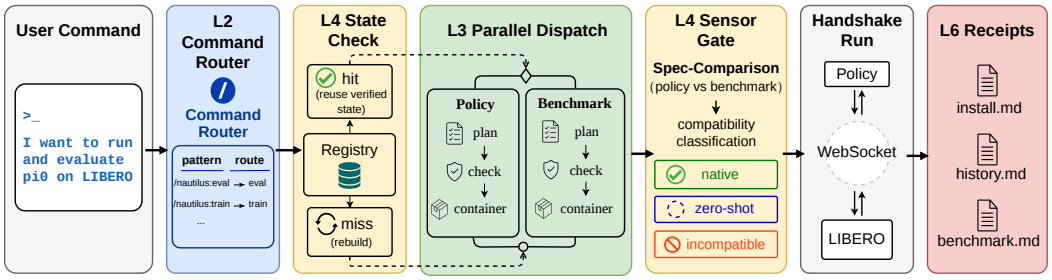


Figure 2: Execution trace for the running example π_0 on LIBERO (§3.1). The orchestrator routes a natural-language request to `> /nautilus:eval`; State, subagents, Sensors, and workspace artefacts then turn the request into a contract-checked, chambered, reproducible evaluation.

and support broader cross-policy comparison. Existing systems make important contributions at the level of individual frameworks, benchmark suites, or deployment pipelines, but are typically tied to particular simulator families, benchmark families, or robot stacks. NAUTILUS instead takes a contract-centric position, standardizing interfaces between policies, benchmarks, and robots to make cross-family composition and extension tractable without rewriting framework internals. Table 1 summarizes this comparison across coverage, hardware interfaces, and extensibility.

3 NAUTILUS

NAUTILUS is a Claude Code plugin that turns natural-language robot learning requests into executed reproduction, evaluation, finetuning, and deployment workflows. Its design has two layers. The first is a *substrate* of three engineering invariants—typed interface contracts, chambered execution, and uniform inter-module transport (§3.2). The second is a *content layer* of robotics-specific components—Guides, Sensors, State, and workspace artefacts (§3.3). The corresponding implementation stack is detailed in Appendix D.1 (Table 6).

3.1 Running example: π_0 on LIBERO

We use one running example throughout the section: a researcher says, in natural language, "I want to run and evaluate π_0 on the LIBERO benchmark." Rather than asking the researcher to locate the right repository entry points, align dependencies, and write the evaluation glue, the orchestrator interprets the request and routes it to `> /nautilus:eval policy=pi0 benchmark=libero`. The resulting run is the trace sketched in Figure 2: the orchestrator dispatches the task, State checks the registry, policy and benchmark modules are generated in parallel, their contracts are compared, a WebSocket smoke test is executed, and reproducibility receipts are written.

3.2 Substrate

NAUTILUS gives the agent a fixed substrate to build against. It has three invariants: *typed interface contracts*, which make policies, benchmarks, and robots legible through fixed schemas; *chambered execution*, which isolates generated artefacts so that dependency failures do not leak across modules; and *uniform inter-module communication*, which gives simulated and real-robot endpoints the same agent-visible observation/action transport.

Typed interface contracts. A robot learning experiment is represented by three typed interfaces: a policy, a robot embodiment, and a benchmark/environment. *Policy* exposes `infer(obs)` $\rightarrow$ `actions` with an optional `reset`. *Robot* exposes `reset`, `get_observation`, `apply_action`, and `safe_stop`. *Benchmark/Environment* exposes `reset(seed)` and `step(action)` together with the task definition, success criterion, and deterministic seeding used for the published-reference checks in §4.1. These contracts are the unit of integration: adding a new policy, benchmark, or robot requires satisfying the corresponding contract rather than modifying the rest of the framework. In Figure 2, the agent therefore connects π_0 and LIBERO through Policy and Benchmark/Environment contracts; for hardware experiments, a Robot contract fills the same role.

Chambered execution. Policy inference, simulator execution, and real-robot control often depend on incompatible CUDA, PyTorch, MuJoCo/Isaac/SAPIEN, and ROS 2 stacks. NAUTILUS therefore runs generated artefacts in disposable chambers: each chamber holds its own dependency environment while exposing the same external contract. A *policy container* serves model inference, while a *benchmark/environment container* hosts the simulator or task suite; real hardware occupies the same architectural slot from the agent's perspective. In the π_0–LIBERO example, chambering allows policy and benchmark modules to build in parallel without dependency failures leaking across them.

Uniform inter-module communication. Contracts and chambers define what each module exposes, but modules still need to exchange observations and actions. NAUTILUS separates inter-module communication into two tiers. *Tier 1 (robot-local)* keeps high-bandwidth sensing and control in the native robot stack (ROS 2/shared memory). *Tier 2 (off-robot)* is the only transport seen by the agent: a lightweight WebSocket between a policy container and a benchmark or real-environment endpoint. Simulators and robots therefore share one agent-visible observation/action schema, while ROS 2, shared memory, and other robot-local communication mechanisms remain below the agent boundary. In the π_0–LIBERO example, the WebSocket smoke test checks this shared Tier 2 interface rather than a pair-specific communication path.

3.3 Content layer

On top of the substrate, the content layer supplies robotics-specific procedures. We adopt the Guides+Sensors abstraction of harness engineering [5]: Guides steer work before generation, while Sensors validate outputs afterward. NAUTILUS adds State to persist typed contracts across runs, and workspace artefacts to record each run for reproducible hand-off. Operationally, **L1** is the base agent loop, **L2** routes user intent, **L3** executes task-specific subagents, **L4** validates artefacts and carries State, **L5** supplies domain references, and **L6** records receipts. The full six-layer implementation stack realising this content layer is given in Table 6 (Appendix D.1).

Guides. Guides tell the agent how to act before it writes code. Instead of inferring robot learning conventions from each repository, the agent is given the relevant routing rules, subagent roles, schemas, and validation protocols. They appear at three layers of Table 6: **L2** routes a request to a script or subagent workflow; **L3** runs subagents for policy servers, benchmark wrappers, training scaffolds, and robot integrations; **L5** provides the shared reference library they consult (full inventory in Appendix E). In the worked example, `> /nautilus:eval policy=pi0 benchmark=libero` therefore means "generate the policy and benchmark modules, then compare their registry specs before launch." A *spec* is the registry-side, machine-readable form of the typed contract in §3.2.

Sensors. Sensors validate what the agent generates. At the **L4** boundary, NAUTILUS runs six checks—pre-action filtering, render-time auditing, interface verification, spec comparison, tiered smoke testing, and cross-run verification—each of which either triggers a local correction or blocks the run; Appendix G gives the full inventory. In Figure 2, the key Sensor is spec comparison. It checks the π_0 Policy spec against the LIBERO Benchmark/Environment spec before launch, surfacing observation-key or action-shape mismatches before any heavyweight run. Registry promotion is handled separately by the curation gate of §3.4.

State. State is the **L4** persistent registry that lets NAUTILUS reuse verified module knowledge across runs. Instead of rediscovering repository layout, environment constraints, and policy/benchmark/robot interfaces from scratch, the agent retrieves registry entries that bind pinned code and environment artefacts to the typed specs of §3.2. These specs describe the module boundary: observation/action schemas, control mode, and benchmark criteria. In the π_0–LIBERO trace, State is checked immediately after routing. Verified entries support reuse and compatibility checks before launch; unverified modules are generated against the same spec format and can later enter curation. This turns policy–benchmark integration from $\Theta(N{\cdot}M)$ hand-written adapters into $\Theta(N{+}M)$ machine-readable specs and a shared dispatch protocol.

Workspace artefacts. The **L6** workspace layer records the evidence for a single run; it is not itself a verified registry entry. *Process logs* are append-only notes in the target repository, while *receipts* are regenerated summaries at the repository root (Appendix D.2). Together, they record the source commit, image ID, environment choices, benchmark protocol, key decisions, and rerun recipe. For π_0 on LIBERO, the receipt makes the run reproducible for a later user and provides evidence for a possible registry submission.

3.4 Registry curation and hardware support

The registry is a shareable index of reproducible entries, not a centralized service. Each entry pins source and environment artefacts, allowing any user to reproduce the same policy, benchmark, or robot setup. The `verified` status is reserved for benchmark entries, because benchmark protocols are far fewer in number and evolve more slowly than policy configurations. We mark a benchmark entry as `verified` only when it is supported by *cross-policy evidence*: several released policies, each run with its official checkpoint through the same wrapper, reproduce their published reference results with transparent deviations rather than a universal pass/fail threshold. §4.1 applies this criterion to the initial set of verified entries; curation mechanics are detailed in Appendices D.3 and F.2.

Hardware reuses the same substrate. `nautilus-collect` is a chambered data-collection platform that implements the Robot typed contract and exposes the same uniform transport. As a result, any policy wrapped once by NAUTILUS can be handed from a benchmark to a real Robot endpoint for deployment. The platform also covers robot data collection, training hand-off, and deployment. We validate this path with a VLA manipulation rollout on a single-arm Franka and an RL locomotion rollout on Unitree H1 (§4.3); details are in Appendices B and C.

4 Experiments

We organize experiments around three studies. §4.1 evaluates benchmark-wrapper fidelity by reproducing the reference results of public policies under their released checkpoints. §4.2 ablates the cost of new policy onboarding on a controlled testbed, isolating which components actually buy reusable extension. §4.3 closes the loop on real hardware. The fine-tuning study and the communication-layer microbenchmarks originally scoped alongside these studies are deferred (see §5).

4.1 Benchmark trust validation

RQ1. Are our generated benchmark wrappers faithful to the references they wrap?

Protocol. We instantiate the cross-policy evidence protocol introduced in §3.4. The unit of verification is a benchmark wrapper, not a single policy–benchmark pair. For each candidate benchmark entry, we run several released policies with their official checkpoints through the same generated wrapper and compare the resulting scores against the corresponding published references. The study covers imitation-learning, vision-language-action, and world-action-model policies across LIBERO, ManiSkill, RoboTwin, ALOHA, and RoboCasa. Because the reference sources do not report a common across-seed variance, we do not impose a universal pass/fail threshold. Instead, we report each reproduced score and its deviation from the corresponding reference. This study covers only the simulation row of Table 1; real-robot deployment is evaluated separately in §4.3.

Metrics. Table 2 reports, for each (method, benchmark) pair, the wrapped success rate, the reference value from the original publication, and the deviation Δ between them. No row is marked with a fixed threshold-based pass/fail label.

Findings. Across IL, VLA, and WAM policies, most wrapped results remain close to their published references, indicating that the benchmark adapters preserve the intended evaluation protocols. Table 2 is thus a fidelity check on benchmark wrappers, not evidence of onboarding new components: close agreement under released checkpoints is expected, but it is exactly the criterion a generated wrapper must meet before it becomes a reusable registry entry. Small deviations support reuse of the wrapper, while larger deviations are retained transparently for future calibration.

4.2 Ablation: from pairwise wiring to reusable extending

RQ2. Can NAUTILUS onboard new implementations and what is the cost?

§4.1 shows that, for existing policy — benchmark implementation, our generated wrappers faithfully reproduce the reference performance. The harder question we care about more is whether the same level of trust can extend beyond existing integrations.

In practice, we find that a strong coding agent can often make a specific local setup work: for example, wiring a policy implementation to a benchmark repository it does not yet support. However, this success is usually local — it is not sufficiently validated, deployable, or reusable across other

Table 2: **Cross-policy trust validation for benchmark wrappers.** Each row reports a (method, benchmark) pair: the success rate measured through our wrapper, the value reported in the original publication, and the deviation Δ. Rows are grouped by policy family — IL (blue), VLA (green), and WAM (orange).

Family	Method	Benchmark	Success rate	Reference	Δ
IL	Diffusion Policy [30]	LIBERO [9]	70.2	72.4	−2.2
		ManiSkill [11]	32.4	30.2	+2.2
		RoboTwin [19]	26.4	28.0	−1.6
		ALOHA [12]	75.8	77.5	−1.7
	ACT [12]	RoboTwin	30.0	29.7	+0.3
		ALOHA	72.8	72.3	+0.5
VLA	π_0 [47]	LIBERO	93.6	94.2	−0.6
		RoboTwin	42.6	46.4	−3.8
	$\pi_{0.5}$ [49]	LIBERO	97.0	96.8	+0.2
		RoboCasa [10]	18.6	16.9	+1.7
	OpenVLA [46]	LIBERO	78.2	76.5	+1.7
		ManiSkill	4.0	4.8	−0.8
	SmolVLA [92]	LIBERO	88.2	87.3	+0.9
WAM	cosmos-policy [93]	LIBERO	98.4	98.5	−0.1
		RoboCasa	66.7	67.1	−0.4
	Motus [73]	RoboTwin	86.9	87.0	−0.1

benchmarks. NAUTILUS aims for a different goal. Once a new component is onboarded through the shared interface, it is not only connected to the current target but also automatically compatible with all the verified implementations and workflows already in the platform.

This section therefore studies whether such platform-aware onboarding can be achieved at reasonable cost while remaining faithful to reference performance. We first introduce a controlled testbed that enables fair comparison between the raw coding-agent baseline and ablated NAUTILUS variants. Then we demonstrate with the components proposed in §3, NAUTILUS can achieve such extensibility at roughly the cost of a single one-off integration.

Testbed for extensibility A direct validation on genuinely unseen policy–benchmark combination is difficult to audit: without released checkpoints on the certain benchmark, success can only be judged after costly training.

We therefore construct a *leave-one-benchmark-out* controlled proxy. Starting from a public policy repository that already has a released checkpoint for a target benchmark, we carefully remove the repository's existing support for that benchmark and ask the agent to reconstruct the missing integration. This allows us to test a typical engineering task faced by robot learning researchers — connecting a custom policy repository to a not yet supported benchmark — while still retaining a task-level oracle: the resulting evaluation should match the reference success rate within the trust band established in §4.1. Details are provided in Appendix D.5.

We curate several pruned policy–benchmark pairs spanning IL and VLA policy families, and covering benchmark suites already trust-validated in §4.1. Detailed pairs are listed in Table 12. Each integration trial has a 2 h wall-clock budget and runs in a fresh sandbox that exposes only the pruned source trees: runtime network access passes through a deny-by-default proxy that blocks code-hosting and search hosts, and the agent's user-, session-, and project-level memory starts empty and is never visible to later trials. These mechanisms exclude leakage through the network and through persisted agent state, but not memorization of public code in model weights (Appendix D.6). Table 3 reports the headline numbers aggregated across the three pruned policy–benchmark pairs; the per-pair breakdown for both models appears in Table 12 in the appendix.

Metrics. Following Han et al. [94] on evaluating coding-agent performance, we measure and report: **Wrap success** (↑) measures whether the agent's generated integration loads the released

Table 3: **Comparison and ablations.** Four configurations are evaluated on the three pruned policy–benchmark pairs. Every cell reports paired values **Claude Opus 4.7 / Claude Sonnet 4.6**; the full per-pair breakdown appears in Table 12. Time, interaction, token, context, and cost columns are averaged over accepted trials and across pairs with at least one accepted trial.

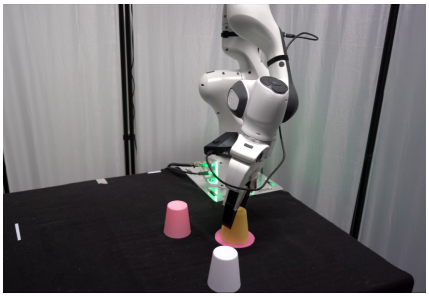
	Agent success (↑)		Agent inference	Interaction budget (↓)	Token/context budget (↓)		Monetary cost
Setting	**Wrap**	**Task**	**Time**	**Agent turns**	**Tokens**	**Peak ctx**	**USD**
Vanilla agent	100.0% / 98.8%	95.4% / 93.2%	16m30s / 14m21s	161.9 / 158.0	12.2M / 12.4M	123.1k / 130.0k	$5.10 / $5.19
NAUTILUS	100.0% / 100.0%	98.7% / 98.0%	23m51s / 17m00s	152.4 / 115.5	13.8M / 9.5M	152.1k / 136.0k	$6.36 / $4.76
NAUTILUS ablations							
w/o template	55.3% / –	55.1% / –	26m00s / –	607.5 / –	41.7M / –	219.2k / –	$74.94 / –
w/o verified image	100.0% / 100.0%	87.6% / 79.2%	25m18s / 15m40s	431.2 / 287.5	16.2M / 9.8M	135.5k / 123.9k	$14.80 / $9.75

checkpoint and completes the benchmark evaluation protocol end-to-end without manual intervention. **Task success** (↑) further requires the resulting score to lie within the trust band of §4.1. **Agent inference time** (↓) measures cumulative agent-side inference time spent on generation before the accepted run. **Interaction budget** (↓) reports the total number of agent turns required to reach a successful evaluation. **Token/context budget** (↓) reports total LLM token usage and peak context length. **Monetary cost** is the metered USD cost under the fixed pricing table of each integration trial.

Findings. Table 3 measures controlled reintegration of a removed policy–benchmark integration. Because the vanilla baseline is already strong, its value lies less in the final success numbers than in two observations: NAUTILUS produces shared-interface onboarding rather than one-off pairwise glue, and the ablations expose concrete failure modes when its provisions are removed. **Agent cost against the vanilla agent.** NAUTILUS raises task success on both backbones (98.7% vs. 95.4% on Opus, 98.0% vs. 93.2% on Sonnet) while delivering shared-interface onboarding rather than a one-off (P, B) wire-up, but its per-trial cost depends on the backbone. On `Claude Opus 4.7`, NAUTILUS costs *more* per accepted trial ($6.36 vs. $5.10; 13.8M vs. 12.2M tokens; +29k peak ctx), because the skill bodies of §3.3 are loaded before the first action and the validation calls of the Sensors stack (§3.3) add polling time; agent turns nonetheless drop (152.4 vs. 161.9), suggesting the benefit of the pre-defined interfaces of §3.2. On `Claude Sonnet 4.6`, NAUTILUS reduces turns, tokens, and cost (115.5 vs. 158.0 turns, 9.5M vs. 12.4M tokens, $4.76 vs. $5.19), while inference time and peak context still rise. The harness contribution thus grows as the underlying model becomes weaker, consistent with prior observations in agent-harness research that scaffolding gains are larger on weaker backbones [8, 5]. The largest cost differences appear against the two ablations below, which remove the interface template or the verified benchmark reuse, not against a vanilla one-off integration. **Without the typed interface template.** The `w/o template` setting strips the Tier-2 policy WebSocket schema of §3.2, collapsing the Opus run to 55.3% wrap success at 607.5 turns, 41.7M tokens, and $74.94, with the additional turns spent rediscovering the policy WebSocket transport and hand-rolling pair-specific IPC plumbing — precisely the per-project glue work that the substrate of §3.2 is designed to remove. **Without the verified registry.** The `w/o verified image` setting forces NAUTILUS to regenerate the benchmark side instead of looking it up in the State registry of §3.3, roughly doubling the workload, with Opus turns nearly tripling (152.4 to 431.2) and cost rising by 2.3×; the configuration nonetheless passes a non-trivial fraction of trials, indicating that the substrate alone supports unaided dual-side wrapping. **Failure modes.** The failed trials, with and without NAUTILUS, fall into three categories: contract mismatches, silent semantic errors (e.g., mismatched observation preprocessing or action definitions), and environment/dependency failures; most arise during task execution. Appendix K details two representative near-successes, a duplicated 180° image rotation and an overridden episode horizon, which raise no error yet yield a benchmark success rate of zero.

4.3 Real-robot deployment path

Protocol and demonstration. This section evaluates the real-robot deployment path of NAUTILUS, rather than reporting a new real-robot benchmark score. After a checkpoint has been wrapped for benchmark evaluation, the same policy artefact can be transferred to hardware by replacing the simulation back-end with a hardware container, while keeping both the policy-facing WebSocket endpoint and the platform-side container unchanged. We demonstrate this path on two deliberately heterogeneous systems: a VLA manipulation checkpoint deployed on a Franka arm and an RL



(a) $\pi_{0.5}$ wrapper deployed on a Franka Panda arm.



(b) RL-trained locomotion policy deployed on Unitree H1.

Figure 3: **Benchmark-to-robot deployment under a shared substrate.** A policy checkpoint wrapped for benchmark evaluation can be reused on hardware through the same policy-facing WebSocket endpoint and platform-side `Robot` contract. The deployment changes only the hardware container and launch namespace: **(a)** a reproduced $\pi_{0.5}$ wrapper on a Franka Panda arm, and **(b)** an RL-trained locomotion policy on Unitree H1.

locomotion policy deployed on a Unitree H1. Appendix H provides the hardware setup and the shared data-collection workflow.

For Franka, we take the reproduced $\pi_{0.5}$ wrapper from Table 2 and deploy it for a real pick-and-place rollout without modifying the policy code. This demonstrates that the same wrapper used in benchmark evaluation can be reused in the hardware deployment path. The released checkpoint, however, does not zero-shot solve our real setup. When task-level adaptation is required, `nautilus-collect` supports the same collect–fine-tune–redeploy workflow under the Robot contract.

For Unitree H1, we deploy an RL-trained locomotion policy through the same policy-facing endpoint and platform-side container. Compared with the benchmark setup, only the H1 hardware container and launch namespace are changed.

Human/agent division. In both deployments, the agent's integration work ends at the simulation pass; a human operator sets up and calibrates the hardware, launches the hardware container, and supervises safety (Table 10 in Appendix H). On Franka, the time from simulation pass to the first real-arm rollout was 1.5 h, mostly spent on Docker image build and pull, with zero changes to policy code. We therefore claim that an already wrapped policy can be moved to hardware, not real-robot automation or reduced hardware-side human labor.

Takeaway. These demonstrations test substrate portability, not cross-policy transfer or state-of-the-art robot performance. The point is that any checkpoint wrapped behind the shared `Policy` interface can be iterated from benchmark evaluation to real hardware without a separate policy-integration effort. Policy-family and embodiment-specific details remain confined below the `Robot` contract, while the agent-facing and policy-facing surfaces remain fixed.

5 Discussion and Future Work

Discussion. Our results support the central claim of §1: the current constraint on agentic robot learning automation lies in the harness around the agent model. With the backbone fixed, the ablations in §4.2 show that removing the typed interface template on OpenVLA$^{\setminus \mathrm{LIBERO}}$ drops Opus wrap success from 100% to 55.3% and raises the cost from $6.32 to $74.94, and removing verified benchmark reuse nearly triples agent turns; the full harness also helps the weaker backbone most. We read these results at the level of mechanisms: the harness does not make the agent smarter; it makes common integration failures surface early, loudly, and cheaply. Typed Policy/Benchmark/Robot contracts mitigate observation/action and control-mode mismatches; Sensors and spec comparison catch silent interface failures before launch; the State registry avoids rediscovering repository layouts, benchmark protocols, and pinned artefacts; chambered execution isolates incompatible dependency stacks; receipts make runs auditable; and the Robot contract lets a benchmark-wrapped policy move to heterogeneous hardware (§4.3) without rewriting the policy integration. As models improve, we

expect them to absorb generic debugging advice, but not shared interfaces, registry knowledge of policies and benchmarks released after their training, or verification that is independent of the agent's own reports.

The current system also has clear limits. First, coverage is limited to the workflow classes we implemented, namely reproduction, evaluation, deployment, and initial finetuning support; broader research workflows still depend on extending the skills library. Second, our cross-simulator unification is at the environment-API level rather than the physics layer, so simulator-specific effects should still be reported per simulator. Third, because NAUTILUS is implemented as a plugin on top of a commodity coding-agent harness, it inherits some upstream platform churn, which we mitigate but do not eliminate. Fourth, silent semantic errors that pass every interface check, such as a duplicated image rotation, are still exposed only by the task-level oracle. Fifth, our isolation protocol cannot exclude memorization of public code in model weights, and pruning tests the reconstruction of a removed integration rather than onboarding a genuinely unseen repository. Finally, on real hardware, setup, calibration, and safety remain human work.

Future work. The most important next step is an auto-research loop for robot learning. Today NAUTILUS automates the engineering loop around a given (P, B, R) triple; the natural extension is to let the agent propose experiments, run ablations, interpret results, and refine hypotheses using the same substrate. A practical step in this direction is to close the train–eval–deploy loop more fully, including automated hyperparameter selection, multi-seed sweeps, and reproduction-grade report generation. We also see automatic skill induction from agent–human traces as a promising way to expand coverage beyond hand-authored procedures, and broader hardware support plus human-in-the-loop safeguards as important system-level extensions. More broadly, the substrate+(Guides, Sensors, State) anatomy may transfer to adjacent scientific domains where generic coding harnesses still fail, though validating that claim requires a second domain-specific instantiation.

6 Conclusion

We argued that the current constraint on agentic automation of robot learning research is structural: a code-writing LLM cannot orchestrate research workflows at scale when the surrounding hardware/software stack offers no stable surface to target. NAUTILUS responds with a co-designed substrate—typed contracts, chambered execution, and uniform inter-module transport—together with a content layer of Guides, Sensors, State, and reproducibility artefacts. Across benchmark verification, reusable onboarding at roughly the cost of a one-off integration, and real-robot deployment demonstrations, the system turns fragmented research workflows into structured agentic procedures rather than pair-specific glue code. We release the platform, skills library, benchmark suite, and cross-policy evidence protocol as a community artefact at https://yufengjin.github.io/nautilus/. More broadly, NAUTILUS suggests a form of progress in which expert-built harnesses do not simply make models replace experts; they make domain expertise executable, reusable, and inspectable.

Acknowledgements and Disclosures

Funding. Funded by the European Union (ERC, SIREN, 101163933). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was further supported by the German Research Foundation (DFG) Emmy Noether Programme (CH 2676/1-2), the EU's Horizon Europe project "ARISE" (Grant no. 101135959), the German Federal Ministry of Education and Research (BMBF) project "RiG" (Grant no. 16ME1001), and Honda Research Institute Europe GmbH, which funds Yufeng Jin's doctoral research. The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) under the NHR project g108ea. NHR funding is provided by federal and Bavarian state authorities.

Competing interests. Jianfei Guo is employed by Chaser Robotics GmbH, and Mathias Franzius by Honda Research Institute Europe GmbH. The remaining authors declare no competing interests.

References

[1] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026.

[2] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

[3] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[4] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IEduRUO55F.

[5] Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026. doi: 10.20944/preprints202604.0428.v2. URL https://www.preprints.org/manuscript/202604.0428/v2.

[6] Nghi D. Q. Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned, 2026. URL https://arxiv.org/abs/2603.05344.

[7] Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses, 2026. URL https://arxiv.org/abs/2603.25723.

[8] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

[9] Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

[10] Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi, Ajay Mandlekar, and Yuke Zhu. Robocasa: Large-scale simulation of everyday tasks for generalist robots. *arXiv preprint arXiv:2406.02523*, 2024.

[11] Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse-kai Chan, et al. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *arXiv preprint arXiv:2410.00425*, 2024.

[12] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.

[13] Remi Cadene, Simon Alibert, Francesco Capuano, Michel Aractingi, Adil Zouitine, Pepijn Kooijmans, Jade Choghari, Martino Russi, Caroline Pascal, Steven Palma, Mustafa Shukor, Jess Moss, Alexander Soare, Dana Aubakirova, Quentin Lhoest, Quentin Gallouédec, and Thomas Wolf. Lerobot: An open-source library for end-to-end robot learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2602.22818.

[14] Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Muñoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, Lukasz Wawrzyniak, Milad Rakhsha, Alain Denzler, Eric Heiden, Ales Borovicka, Ossama Ahmed, Iretiayo Akinola, Abrar Anwar, Mark T. Carlson, Ji Yuan Feng, Animesh Garg, Renato Gasoto, Lionel Gulich, Yijie Guo, M. Gussert, Alex Hansen, Mihir Kulkarni, Chenran Li, Wei Liu, Viktor Makoviychuk, Grzegorz Malczyk, Hammad Mazhar, Masoud Moghani, Adithyavairavan Murali, Michael Noseworthy, Alexander Poddubny, Nathan Ratliff, Welf Rehberg, Clemens Schwarke, Ritvik

Singh, James Latham Smith, Bingjie Tang, Ruchik Thaker, Matthew Trepte, Karl Van Wyk, Fangzhou Yu, Alex Millane, Vikram Ramasamy, Remo Steiner, Sangeeta Subramanian, Clemens Volk, CY Chen, Neel Jawale, Ashwin Varghese Kuruttukulam, Michael A. Lin, Ajay Mandlekar, Karsten Patzwaldt, John Welsh, Huihua Zhao, Fatima Anes, Jean-Francois Lafleche, Nicolas Moënne-Loccoz, Soowan Park, Rob Stepinski, Dirk Van Gelder, Chris Amevor, Jan Carius, Jumyung Chang, Anka He Chen, Pablo de Heras Ciechomski, Gilles Daviet, Mohammad Mohajerani, Julia von Muralt, Viktor Reutskyy, Michael Sauter, Simon Schirm, Eric L. Shi, Pierre Terdiman, Kenny Vilella, Tobias Widmer, Gordon Yeoman, Tiffany Chen, Sergey Grizan, Cathy Li, Lotus Li, Connor Smith, Rafael Wiltz, Kostas Alexis, Yan Chang, David Chu, Linxi "Jim" Fan, Farbod Farshidian, Ankur Handa, Spencer Huang, Marco Hutter, Yashraj Narang, Soha Pouya, Shiwei Sheng, Yuke Zhu, Miles Macklin, Adam Moravanszky, Philipp Reist, Yunrong Guo, David Hoeller, and Gavriel State. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025. URL https://arxiv.org/abs/2511.04831.

[15] Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, Yifeng Zhu, and Kevin Lin. robosuite: A modular simulation framework and benchmark for robot learning. In *arXiv preprint arXiv:2009.12293*, 2020.

[16] Anthropic. Claude code: An agentic coding tool. GitHub, 2025. 2025b.

[17] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

[18] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

[19] Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Qiwei Liang, Zixuan Li, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025.

[20] Ajay Mandlekar, Danfei Xu, Josiah Wong, Soroush Nasiriany, Chen Wang, Rohun Kulkarni, Li Fei-Fei, Silvio Savarese, Yuke Zhu, and Roberto Martín-Martín. What matters in learning from offline human demonstrations for robot manipulation. In *Conference on Robot Learning (CoRL)*, 2021.

[21] NVIDIA Isaac Lab-Arena Contributors. Isaac lab-arena: Composable environment creation and policy evaluation for robotics, 2025. URL https://github.com/isaac-sim/IsaacLab-Arena.

[22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

[25] Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin,

Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vUaY1t64ZZ.

[26] Stephan Rabanser, Sayash Kapoor, Peter Kirgis, Kangheng Liu, Saiteja Utpala, and Arvind Narayanan. Towards a science of ai agent reliability, 2026. URL https://arxiv.org/abs/2602.16666.

[27] Leon Staufer, Kevin Feng, Kevin Wei, Luke Bailey, Yawen Duan, Mick Yang, A. Pinar Ozisik, Stephen Casper, and Noam Kolt. The 2025 ai agent index: Documenting technical and safety features of deployed agentic ai systems, 2026. URL https://arxiv.org/abs/2602.17753.

[28] Pete Florence, Corey Lynch, Andy Zeng, Oscar A Ramirez, Ayzaan Wahid, Laura Downs, Adrian Wong, Johnny Lee, Igor Mordatch, and Jonathan Tompson. Implicit behavioral cloning. In *Conference on robot learning*, pages 158–168. PMLR, 2022.

[29] Xiaogang Jia, Qian Wang, Anrui Wang, Han A. Wang, Balázs Gyenes, Emiliyan Gospodinov, Xinkai Jiang, Ge Li, Hongyi Zhou, Weiran Liao, Xi Huang, Maximilian Beck, Moritz Reuss, Rudolf Lioutikov, and Gerhard Neumann. Pointmappolicy: Structured point cloud processing for multi-modal imitation learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=ZR2mdBrhJX.

[30] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025.

[31] Zipeng Fu, Tony Z. Zhao, and Chelsea Finn. Mobile ALOHA: Learning bimanual mobile manipulation using low-cost whole-body teleoperation. In *8th Annual Conference on Robot Learning*, 2024. URL https://openreview.net/forum?id=FO6tePGRZj.

[32] Cheng Chi, Zhenjia Xu, Chuer Pan, Eric Cousineau, Benjamin Burchfiel, Siyuan Feng, Russ Tedrake, and Shuran Song. Universal manipulation interface: In-the-wild robot teaching without in-the-wild robots. *arXiv preprint arXiv:2402.10329*, 2024.

[33] Yanjie Ze, Gu Zhang, Kangning Zhang, Chenyuan Hu, Muhan Wang, and Huazhe Xu. 3d diffusion policy: Generalizable visuomotor policy learning via simple 3d representations. In *Proceedings of Robotics: Science and Systems (RSS)*, 2024.

[34] Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Charles Xu, Jianlan Luo, Tobias Kreiman, You Liang Tan, Pannag Sanketi, Quan Vuong, Ted Xiao, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An open-source generalist robot policy. In *Proceedings of Robotics: Science and Systems*, Delft, Netherlands, 2024.

[35] Siddhant Haldar, Zhuoran Peng, and Lerrel Pinto. Baku: An efficient transformer for multi-task policy learning. *arXiv preprint arXiv:2406.07539*, 2024.

[36] Zhi Hou, Tianyi Zhang, Yuwen Xiong, Haonan Duan, Hengjun Pu, Ronglei Tong, Chengyang Zhao, Xizhou Zhu, Yu Qiao, Jifeng Dai, and Yuntao Chen. Dita: Scaling diffusion transformer for generalist vision-language-action policy. *arXiv preprint arXiv:2503.19757*, 2025.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

[39] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[40] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

[41] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=fgFBtYgJQX_.

[42] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[43] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Tomas Jackson, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Kuang-Huei Lee, Sergey Levine, Yao Lu, Utsav Malla, Deeksha Manjunath, Igor Mordatch, Ofir Nachum, Carolina Parada, Jodilyn Peralta, Emily Perez, Karl Pertsch, Jornell Quiambao, Kanishka Rao, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Kevin Sayed, Jaspiar Singh, Sumedh Sontakke, Austin Stone, Clayton Tan, Huong Tran, Vincent Vanhoucke, Steve Vega, Quan Vuong, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. Rt-1: Robotics transformer for real-world control at scale. In *arXiv preprint arXiv:2212.06817*, 2022.

[44] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[45] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alex Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Lisa Lee, Tsang-Wei Edward Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspiar Singh, Anikait Singh, Radu Soricut, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *arXiv preprint arXiv:2307.15818*, 2023.

[46] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[47] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

[48] Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. *arXiv preprint arXiv:2502.19645*, 2025.

[49] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization, 2025. URL https://arxiv.org/abs/2504.16054.

[50] Xinghang Li, Minghuan Liu, Hanbo Zhang, Cunjun Yu, Jie Xu, Hongtao Wu, Chilam Cheang, Ya Jing, Weinan Zhang, Huaping Liu, et al. Vision-language foundation models as effective robot imitators. *arXiv preprint arXiv:2311.01378*, 2023.

[51] AI Figure. Helix: A vision-language-action model for generalist humanoid control. *Figure AI News*, 2024.

[52] Qixiu Li, Yaobo Liang, Zeyu Wang, Lin Luo, Xi Chen, Mozheng Liao, Fangyun Wei, Yu Deng, Sicheng Xu, Yizhong Zhang, et al. Cogact: A foundational vision-language-action model for synergizing cognition and action in robotic manipulation. *arXiv preprint arXiv:2411.19650*, 2024.

[53] Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. Rdt-1b: a diffusion foundation model for bimanual manipulation. *arXiv preprint arXiv:2410.07864*, 2024.

[54] Junjie Wen, Yichen Zhu, Jinming Li, Minjie Zhu, Kun Wu, Zhiyuan Xu, Ran Cheng, Chaomin Shen, Yaxin Peng, Feifei Feng, et al. Tinyvla: Towards fast, data-efficient vision-language-action models for robotic manipulation. *arXiv preprint arXiv:2409.12514*, 2024.

[55] Haoyu Zhen, Xiaowen Qiu, Peihao Chen, Jincheng Yang, Xin Yan, Yilun Du, Yining Hong, and Chuang Gan. 3d-vla: 3d vision-language-action generative world model. *arXiv preprint arXiv:2403.09631*, 2024.

[56] Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

[57] Moritz Reuss, Hongyi Zhou, Marcel Rühle, Ömer Erdinç Yağmurlu, Fabian Otto, and Rudolf Lioutikov. FLOWER: Democratizing generalist robot policies with efficient vision-language-flow models. In *9th Annual Conference on Robot Learning*, 2025. URL https://openreview.net/forum?id=JeppaebLRD.

[58] Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, et al. Gemini robotics: Bringing ai into the physical world. *arXiv preprint arXiv:2503.20020*, 2025.

[59] Junjie Wen, Yichen Zhu, Jinming Li, Zhibin Tang, Chaomin Shen, and Feifei Feng. Dexvla: Vision-language model with plug-in diffusion expert for general robot control. *arXiv preprint arXiv:2502.05855*, 2025.

[60] Zhongyi Zhou, Yichen Zhu, Minjie Zhu, Junjie Wen, Ning Liu, Zhiyuan Xu, Weibin Meng, Ran Cheng, Yaxin Peng, Chaomin Shen, and Feifei Feng. Chatvla: Unified multimodal understanding and robot control with vision-language-action model, 2025. URL https://arxiv.org/abs/2502.14420.

[61] Hengtao Li, Pengxiang Ding, Runze Suo, Yihao Wang, Zirui Ge, Dongyuan Zang, Kexian Yu, Mingyang Sun, Hongyin Zhang, Donglin Wang, et al. Vla-rft: Vision-language-action reinforcement fine-tuning with verified rewards in world simulators. *arXiv preprint arXiv:2510.00406*, 2025.

[62] Jijia Liu, Feng Gao, Bingwen Wei, Xinlei Chen, Qingmin Liao, Yi Wu, Chao Yu, and Yu Wang. What can RL bring to VLA generalization? an empirical study. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=qmBMPInbZC.

[63] Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, Suneel Indupuru, You Liang Tan, Chuning Zhu, Jiannan Xiang, Ayaan Malik, Kyungmin Lee, William Liang, Nadun Ranawaka, Jiasheng Gu, Yinzhen Xu, Guanzhi Wang, Fengyuan Hu, Avnish Narayan, Johan Bjorck, Jing Wang, Gwanghyun Kim, Dantong Niu, Ruijie Zheng, Yuqi Xie, Jimmy Wu, Qi Wang, Ryan Julian, Danfei Xu, Yilun Du, Yevgen Chebotar, Scott Reed, Jan Kautz, Yuke Zhu, Linxi "Jim" Fan, and Joel Jang. World action models are zero-shot policies, 2026. URL https://arxiv.org/abs/2602.15922.

[64] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[65] Philipp Wu, Alejandro Escontrela, Danijar Hafner, Pieter Abbeel, and Ken Goldberg. Daydreamer: World models for physical robot learning. In *Conference on robot learning*, pages 2226–2240. PMLR, 2023.

[66] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[67] Nicklas Hansen, Hao Su, and Xiaolong Wang. Td-mpc2: Scalable, robust world models for continuous control. In *International Conference on Learning Representations (ICLR)*, 2024.

[68] Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024.

[69] Jialong Wu, Shaofeng Yin, Ningya Feng, Xu He, Dong Li, Jianye Hao, and Mingsheng Long. ivideogpt: Interactive videogpts are scalable world models. In *Advances in Neural Information Processing Systems*, 2024.

[70] Danijar Hafner, Wilson Yan, and Timothy Lillicrap. Training agents inside of scalable world models. *arXiv preprint arXiv:2509.24527*, 2025.

[71] Shenyuan Gao, William Liang, Kaiyuan Zheng, Ayaan Malik, Seonghyeon Ye, Sihyun Yu, Wei-Cheng Tseng, Yuzhu Dong, Kaichun Mo, Chen-Hsuan Lin, Qianli Ma, Seungjun Nah, Loic Magne, Jiannan Xiang, Yuqi Xie, Ruijie Zheng, Dantong Niu, You Liang Tan, K.R. Zentner, George Kurian, Suneel Indupuru, Pooya Jannaty, Jinwei Gu, Jun Zhang, Jitendra Malik, Pieter Abbeel, Ming-Yu Liu, Yuke Zhu, Joel Jang, and Linxi "Jim" Fan. Dreamdojo: A generalist robot world model from large-scale human videos. *arXiv preprint arXiv:2602.06949*, 2026.

[72] Yanjiang Guo, Lucy Xiaoyang Shi, Jianyu Chen, and Chelsea Finn. Ctrl-world: A controllable generative world model for robot manipulation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=748bHL2BAv.

[73] Hongzhe Bi, Hengkai Tan, Shenghao Xie, Zeyuan Wang, Shuhe Huang, Haitian Liu, Ruowen Zhao, Yao Feng, Chendong Xiang, Yinze Rong, Hongyan Zhao, Hanyu Liu, Zhizhong Su, Lei Ma, Hang Su, and Jun Zhu. Motus: A unified latent action world model, 2025. URL https://arxiv.org/abs/2512.13030.

[74] Angen Ye, Boyuan Wang, Chaojun Ni, Guan Huang, Guosheng Zhao, Hao Li, Hengtao Li, Jie Li, Jindi Lv, Jingyu Liu, Min Cao, Peng Li, Qiuping Deng, Wenjun Mei, Xiaofeng Wang, Xinze Chen, Xinyu Zhou, Yang Wang, Yifan Chang, Yifan Li, Yukun Zhou, Yun Ye, Zhichao Liu, and Zheng Zhu. Gigaworld-policy: An efficient action-centered world-action model. *arXiv preprint arXiv:2603.17240*, 2026.

[75] Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation. *arXiv preprint arXiv:2311.01455*, 2023.

[76] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. In *7th Annual Conference on Robot Learning*, 2023. URL https://openreview.net/forum?id=9_8LF30mOC.

[77] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A simulated part-based interactive environment. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[78] Xiaogang Jia, Atalay Donat, Xi Huang, Xuan Zhao, Denis Blessing, Hongyi Zhou, Han A Wang, Hanyi Zhang, Qian Wang, Rudolf Lioutikov, et al. X-il: Exploring the design space of imitation learning policies. *arXiv preprint arXiv:2502.12330*, 2025.

[79] Ajay Mandlekar, Soroush Nasiriany, Bowen Wen, Iretiayo Akinola, Yashraj Narang, Linxi Fan, Yuke Zhu, and Dieter Fox. Mimicgen: A data generation system for scalable robot learning using human demonstrations. In *7th Annual Conference on Robot Learning*, 2023. URL https://openreview.net/forum?id=dk-2R1f_LR.

[80] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66): eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[81] Genesis Authors. Genesis: A generative and universal physics engine for robotics and beyond, December 2024. URL https://github.com/Genesis-Embodied-AI/Genesis.

[82] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A platform for embodied ai research, 2019. URL https://arxiv.org/abs/1904.01201.

[83] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 2020.

[84] Yuanpei Chen, Yaodong Yang, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuan Jiang, Zongqing Lu, Stephen Marcus McAleer, Hao Dong, and Song-Chun Zhu. Towards human-level bimanual dexterous manipulation with reinforcement learning. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=D29JbExncTP.

[85] Oier Mees, Lukas Hermann, Erick Rosete-Beas, and Wolfram Burgard. Calvin: A benchmark for language-conditioned policy learning for long-horizon robot manipulation tasks. *IEEE Robotics and Automation Letters (RA-L)*, 7(3):7327–7334, 2022.

[86] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Wensi Ai, Benjamin Martinez, Hang Yin, Michael Lingelbach, Minjune Hwang, Ayano Hiranaka, Sujay Garlanka, Arman Aydin, Sharon Lee, Jiankai Sun, Mona Anvari, Manasi Sharma, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Yunzhu Li, Silvio Savarese, Hyowon Gweon, C. Karen Liu, Jiajun Wu, and Li Fei-Fei. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. *arXiv preprint arXiv:2403.09227*, 2024.

[87] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, Peter David Fagan, Joey Hejna, Masha Itkina, Marion Lepert, Yecheng Jason Ma, Patrick Tree Miller, Jimmy Wu, Suneel Belkhale, Shivin Dass, Huy Ha, Arhan Jain, Abraham Lee, Youngwoon Lee, Marius Memmel, Sungjae Park, Ilija Radosavovic, Kaiyuan Wang, Albert Zhan, Kevin Black, Cheng Chi, Kyle Beltran Hatch, Shan Lin, Jingpei Lu, Jean Mercat, Abdul Rehman, Pannag R Sanketi, Archit Sharma, Cody Simpson, Quan Vuong, Homer Rich Walke, Blake Wulfe, Ted Xiao, Jonathan Heewon Yang, Arefeh Yavary, Tony Z. Zhao, Christopher Agia, Rohan Baijal, Mateo Guaman Castro, Daphne Chen, Qiuyu Chen, Trinity Chung, Jaimyn Drake, Ethan Paul Foster, Jensen Gao, Vitor Guizilini, David Antonio Herrera, Minho Heo, Kyle Hsu, Jiaheng Hu, Muhammad Zubair Irshad, Donovon Jackson, Charlotte Le, Yunshuang Li, Kevin Lin, Roy Lin, Zehan Ma, Abhiram Maddukuri, Suvir Mirchandani, Daniel Morton, Tony Nguyen, Abigail O'Neill, Rosario Scalise, Derick Seale, Victor Son, Stephen Tian, Emi Tran, Andrew E. Wang, Yilin Wu, Annie Xie, Jingyun Yang, Patrick Yin, Yunchu Zhang, Osbert Bastani, Glen Berseth, Jeannette Bohg, Ken Goldberg, Abhinav Gupta, Abhishek Gupta, Dinesh Jayaraman, Joseph J Lim, Jitendra Malik, Roberto Martín-Martín, Subramanian Ramamoorthy, Dorsa Sadigh, Shuran Song, Jiajun Wu, Michael C. Yip, Yuke Zhu, Thomas Kollar, Sergey Levine, and Chelsea Finn. Droid: A large-scale in-the-wild robot manipulation dataset, 2025. URL https://arxiv.org/abs/2403.12945.

[88] Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, et al. Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6892–6903. IEEE, 2024.

[89] Qingwen Bu, Jisong Cai, Li Chen, Xiuqi Cui, Yan Ding, Siyuan Feng, Xindong He, Xu Huang, et al. Agibot world colosseo: A large-scale manipulation platform for scalable and intelligent embodied systems. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2025.

[90] Pranav Atreya, Karl Pertsch, Tony Lee, Moo Jin Kim, Arhan Jain, Artur Kuramshin, Clemens Eppner, Cyrus Neary, Edward Hu, Fabio Ramos, et al. Roboarena: Distributed real-world evaluation of generalist robot policies. In *Proceedings of the Conference on Robot Learning (CoRL 2025)*, 2025.

[91] Yejin Kim, Wilbert Pumacay, Omar Rayyan, Max Argus, Winson Han, Eli VanderBilt, Jordi Salvador, Abhay Deshpande, Rose Hendrix, Snehal Jauhri, et al. Molmospaces: A large-scale open ecosystem for robot navigation and manipulation. *arXiv preprint arXiv:2602.11337*, 2026.

[92] Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. Smolvla: A vision-language-action model for affordable and efficient robotics. *arXiv preprint arXiv:2506.01844*, 2025.

[93] Moo Jin Kim, Yihuai Gao, Tsung-Yi Lin, Yen-Chen Lin, Yunhao Ge, Grace Lam, Percy Liang, Shuran Song, Ming-Yu Liu, Chelsea Finn, et al. Cosmos policy: Fine-tuning video models for visuomotor control and planning. *arXiv preprint arXiv:2601.16163*, 2026.

[94] Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. Swe-skills-bench: Do agent skills actually help in real-world software engineering?, 2026. URL https://arxiv.org/abs/2603.15401.

[95] Yixin Lin, Austin S. Wang, Eric Undersander, and Akshara Rai. Polymetis: a real-time PyTorch controller manager for robotics. https://facebookresearch.github.io/fairo/polymetis/, 2021.

[96] Sabela Ramos, Sertan Girgin, Léonard Hussenot, Manu Orsini, Piotr Stanczyk, Olivier Pietquin, Matthieu Geist, and Olivier Bachem. RLDS: an ecosystem to generate, share and use datasets in reinforcement learning. In *NeurIPS Offline RL Workshop*, 2021. URL https://arxiv.org/abs/2111.02767.

[97] Stereolabs. ZED SDK. https://www.stereolabs.com/developers/release, 2024.

[98] Firas Al-Hafez, Guoping Zhao, Jan Peters, and Davide Tateo. Locomujoco: A comprehensive imitation learning benchmark for locomotion. In *6th Robot Learning Workshop, NeurIPS*, 2023.

[99] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

Algorithm 1 Three typed contracts used as the module boundary. Policy implementations hide model-specific loading and preprocessing; Benchmark/Environment implementations hide simulator-specific task logic and success criteria; Robot implementations hide drivers, controllers, and safety logic.

```python
# === Policy ===
class Policy:
    def reset(self) -> None:
        ...

    def infer(
        self,
        obs: Observation,
    ) -> Action:
        ...
```

```python
# === Benchmark / Env ===
class Benchmark:
    def reset(
        self, seed: int,
    ) -> Observation:
        ...

    def step(
        self, action: Action,
    ) -> Transition:
        ...
```

```python
# === Robot ===
class Robot:
    def reset(
        self,
    ) -> Observation:
        ...

    def get_observation(
        self,
    ) -> Observation:
        ...

    def apply_action(
        self, action: Action,
    ) -> None:
        ...

    def safe_stop(self) -> None:
        ...
```

A Module interfaces and the policy WebSocket

This appendix supplements §3.2 and §3.2: the three typed interfaces are engineered so that any policy can communicate with any benchmark or real-robot endpoint through a *single* Tier-2 WebSocket protocol. Algorithm 2 shows the two minimal endpoints—policy-side on the left, environment-side on the right—and the subsection that follows documents the `policy_websocket` package they both depend on. The two endpoints are condensed from the reference implementations released alongside the platform (`openvla/vla-scripts/policy_server.py` and `LIBERO/scripts/run_eval.py`); substituting either side leaves the other unchanged.

A.1 Typed contract overview

The Method section uses *contract* to mean the small typed surface that an agent must satisfy before a module can enter the registry. Each contract hides implementation-specific machinery and exposes only the methods needed for composition. Algorithm 1 summarizes the three surfaces used throughout the paper; the later appendix subsections expand them into the reference WebSocket, benchmark, and hardware implementations.

The registry stores what the agent needs to compose these modules: observation keys, action shape, timing, reset semantics, and verification status. Algorithm 2 and Table 4 give the Policy WebSocket implementation; Algorithm 3 gives the Benchmark/Environment surface; and Algorithm 4 gives the hardware-facing Robot surface.

A.2 The `policy_websocket` package

NAUTILUS ships the Tier-2 protocol of §3.2 as a stand-alone library, `policy_websocket`, with no dependency on the rest of the platform. The protocol surface is exhausted by the four classes in Table 4 and the session state machine described below.

Session state machine. A session has three phases. *(i) Connect*: the client opens a WebSocket; the server immediately pushes its `metadata` blob (`action_dim`, `policy_name`, `checkpoint`, ...) so the client can validate compatibility before the first step. *(ii) Loop*: at each step the client sends one msgpack-encoded observation dict and receives one action dict; the response carries a `server_timing` field (`infer_ms`, `prev_total_ms`) so the client can profile inference latency without instrumenting the server. *(iii) Termination*: the client closes the connection at episode end. If `infer` raises on the server side, the server sends the formatted traceback as a string and closes the connection with code `INTERNAL_ERROR`; the client surfaces this as a `RuntimeError`, so policy-side failures propagate to the rollout loop instead of being silently swallowed.

Algorithm 2 Two minimal endpoints of the policy WebSocket protocol. The policy server (left) wraps any inference-capable model behind `BasePolicy` (optionally through `ActionChunkBroker` for chunk-prediction models). The policy client (right) is itself a `BasePolicy`, so a benchmark or real-robot rollout loop drives a local stub and a remote checkpoint with identical code. Substituting either side leaves the other unchanged.

```python
# === Policy server ===
from policy_websocket import (
    BasePolicy, WebsocketPolicyServer,
    ActionChunkBroker,
)

class MyPolicy(BasePolicy):    # wrap any model
    def __init__(self, ckpt):
        self.model = load_model(ckpt)
    def infer(self, obs):      # core contract
        x = preprocess(obs)
        a = self.model.predict(x)  # (H, act_dim)
        return {"actions": a}

# optional: chunk policy -> per-step server
policy = ActionChunkBroker(
    MyPolicy(ckpt), action_horizon=H,
)
WebsocketPolicyServer(         # serve over ws://
    policy=policy,
    host="0.0.0.0", port=PORT,
    metadata={"action_dim": D,
              "execute_steps": H},
).serve_forever()
```

```python
# === Policy client (benchmark / robot) ===
from policy_websocket import (
    WebsocketClientPolicy,
)

# remote policy behaves like a local one
policy = WebsocketClientPolicy(
    host=HOST, port=PORT,
)
meta = policy.get_server_metadata()
assert meta["action_dim"] == D

env = make_env()      # sim env or real robot
for ep in range(N_episodes):
    obs = env.reset(); policy.reset()
    for t in range(T_max):
        act = policy.infer(obs)["actions"]
        obs, _, done, _ = env.step(act)
        if done: break
```

Table 4: The four classes that make up the `policy_websocket` public API. Together they specify the entire surface that the two endpoints in Algorithm 2 depend on.

Class	Role	Key methods / behaviour
`BasePolicy`	abstract contract every policy implements	`infer(obs)` $\rightarrow$ `{"actions": ...}`; optional `reset()`
`Websocket-PolicyServer`	wraps any `BasePolicy` and serves it over WebSocket	`serve_forever()`; pushes a `metadata` blob on connect; injects `server_timing` into every response
`Websocket-ClientPolicy`	itself a `BasePolicy`; forwards `infer` to a remote server	`infer(obs)`, `get_server_metadata()`, `close()` — drop-in substitute for an in-process policy
`Action-ChunkBroker`	adapts *predict-N, execute-M* chunked policies to one-step clients	wraps any `BasePolicy`; queries the inner policy once per M `infer` calls (ACT, Diffusion Policy, OpenVLA-OFT)

Wire format. Payloads are serialised with msgpack plus a NumPy extension, which carries multi-camera RGB-D observations and action arrays without Base64 inflation; the connection is persistent and on localhost the round-trip is approximately 0.3 ms. A `GET /healthz` endpoint on the same port serves liveness probes for container orchestration.

What the symmetry buys. Both endpoints in Algorithm 2 depend only on `policy_websocket`; neither imports a symbol from the other. The consequence is operational: a new policy is added by writing the left column without touching any benchmark or robot code, and a new benchmark or real-robot stack is added by writing the right column without touching any policy. Any (policy, benchmark) or (policy, robot) pair therefore composes without a custom adapter — a random-action stub on the left can drive any client for L3-IL smoke testing (§E.2), and the same simulator client can drive any registered checkpoint on the right. This is also what permits NAUTILUS's `policy-generator` and

Algorithm 3 The two-method Benchmark/Environment contract every supported task suite satisfies above its simulator backend (Isaac Sim, MuJoCo, or SAPIEN). `reset` and `step` mirror the Gym API; task definition, success criterion, and deterministic seeding are stored as benchmark metadata.

```
class Benchmark:
    def reset(self, seed: int) -> Observation: ...
    def step(self, action: Action) -> Transition: ...
```

`benchmark-generator` skills (§3.3) to scaffold the two sides independently: they target the same wire format, share no state, and the wire format is the entirety of the contract.

Deployed instances. Concretely, the policy server in Algorithm 2 has been exercised in tree with OpenVLA-OFT, OpenPI-π_0, and Diffusion Policy checkpoints; the policy client with the LIBERO, RoboCasa, and ManiSkill simulators and with real-robot rollouts driven through `nautilus-collect` (§C.5). None of these combinations required modifications to either column of Algorithm 2; the per-checkpoint specifics (observation remapping, action chunk size, gripper sign convention) live entirely *inside* the policy-server class, behind the `infer` method.

A.3 Benchmark/Environment contract

The benchmark side of every Tier-2 WebSocket session is a Gym-compatible environment with an associated task definition and success criterion (Algorithm 3). Any benchmark wrapper that satisfies this two-method surface — LIBERO, RoboCasa, ManiSkill, RoboTwin, and ALOHA in this submission — is mountable through `policy_websocket`'s `WebsocketClientPolicy` (right column of Algorithm 2) without further modification. The success criterion is benchmark metadata used to aggregate rollouts on the standard task split, the same surface that §4.1 validates against published references.

B Robot hardware platform

Design principle. Hardware is where the modularity claim of §3.2 is most aggressively stress-tested: each robot family carries its own driver ecosystem, control-loop frequency, joint limits, and safety logic, and these concerns do not generalise across embodiments. Where DROID [87] standardises the *box*—one Franka, one camera rig, one teleoperation stack—so that data-collection sites can replicate the same setup, NAUTILUS takes the deliberate inverse position: we standardise the *interface* and let the same agent stack drive heterogeneous embodiments without modifying policy or benchmark code. Replicability of any single embodiment is delegated to whoever ships its driver; portability across embodiments is delegated to the contract.

Two-container isolation. NAUTILUS factors every supported platform into two ROS 2-based containers whose only coupling is a fixed upward message schema. The *hardware container* hosts whatever talks to the manufacturer driver and re-publishes everything upward as the same standardised ROS 2 hardware interface—a fixed set of state topics, command topics, services, and actions whose schema does not depend on the robot underneath. The container is deliberately permissive about *how* that driver is implemented: a native ROS 2 package, a Python SDK wrapper around a vendor library, or a C++ binary all sit equally well inside, provided whatever sits inside speaks the standard schema upward. The *platform container* hosts the robot learning stack—perception, policy inference, and the Tier-2 WebSocket endpoint of §3.2—and consumes only that uniform interface; it never imports a manufacturer symbol or links against a vendor library. The two communicate over DDS plus host-shared-memory IPC, which insulates the deep-learning stack's dependency closure from the driver's—these are otherwise incompatible at the system-library level—and confines cross-version friction to a one-time build-time problem. ROS 2 Humble on Ubuntu 22.04 is the standard base for both containers; the only exception we currently ship is Franka with Polymetis, where the hardware container is pinned to Ubuntu 20.04 / ROS 2 Foxy purely for Polymetis compatibility—the platform container above it stays on the standard base. Three loop frequencies coexist on this fabric: the manufacturer driver runs at 1 kHz, ROS 2 state aggregation at 50 Hz, and the policy/benchmark application loop at 15 Hz, which is sufficient headroom for diffusion- and VLA-class policies whose inference dominates the application-loop budget.

Algorithm 4 The four-method `Robot` contract that every supported embodiment implements above the standardised ROS 2 hardware interface.

```
class Robot:
    def reset(self) -> Observation: ...
    def get_observation(self) -> Observation: ...
    def apply_action(self, action: Action) -> None: ...
    def safe_stop(self) -> None: ...
```

Table 5: Robot families currently integrated under the four-method `Robot` contract of §3.2. The driver column illustrates the deliberate heterogeneity allowed inside the hardware container (ROS 2 package, Python SDK, or C++ SDK); the upward ROS 2 schema is identical regardless. From the agent's WebSocket-side interface (§3.2), a single-arm xArm rollout and a humanoid H1 rollout are indistinguishable apart from their observation and action shapes.

Platform	**Embodiment class**	**Driver inside hardware container**	**Container base**
Franka Panda 7-DoF	single-arm and bimanual manipulation	Polymetis [95] (Python over libfranka), 1 kHz	Ubuntu 20.04 / ROS 2 Foxy — pinned for Polymetis compatibility
UFactory xArm 7	single-arm manipulation	xArm Python SDK	Ubuntu 22.04 / ROS 2 Humble
Allegro Hand (16-DoF)	dexterous manipulation (mounted on a host arm)	Allegro ROS 2 driver (native ROS 2 package)	Ubuntu 22.04 / ROS 2 Humble
Unitree H1	humanoid locomotion	Unitree SDK (C++)	Ubuntu 22.04 / ROS 2 Humble

The `Robot` contract. Above this uniform hardware interface, the platform-container side of every supported robot exposes the same four-method `Robot` contract introduced in §3.2 (Algorithm 4).

Each method is a thin reader or writer of the standardised hardware interface: `get_observation` consumes the standardised state topics, `apply_action` publishes the standardised command topics or invokes the standardised services, `reset` composes them into a known initial state, and `safe_stop` triggers the standardised emergency-stop service. Everything below the contract—manufacturer driver, real-time inverse kinematics, joint-limit enforcement, fault recovery—is reused from the existing ROS 2 driver ecosystem rather than reimplemented. This deliberate non-invention is what keeps the integration cost of a new embodiment bounded, and it is what allows the agent's robot-skill family (§3.3) to target a single schema regardless of which robot is on the other end.

Currently supported platforms. The platform integrates four robot families that together span the four embodiment classes introduced in §3.2; the same `Robot` contract serves all four (Table 5).

Adding a new embodiment. Bringing a new robot under NAUTILUS requires four artefacts: a `Robot` subclass implementing the four contract methods, a URDF (or equivalent kinematic description), a ROS 2 node that wraps the manufacturer driver and exchanges the standard message schema, and a launch file that wires the two containers together. The robot-skill family of §3.3 is best understood as *scaffolding* for this process rather than full automation: it ships a triggerable code base in which the four artefacts appear as templates with the contract methods, ROS 2 schema, and launch wiring already laid out. The embodiment-specific judgement—which SDK call corresponds to which contract method, what the joint limits and safe default impedance gains should be, how the manufacturer driver reports faults—still has to be supplied by a human (or by a human-in-the-loop agent run) on a per-platform basis. With the scaffolding in place we have found the integration scope to fit within a few hundred lines per platform, configuration included; the structural simplicity of the target is what bounds that scope, not the skill family producing finished code on its own.

C Robot learning platform: `nautilus-collect` implementation and data-collection stack

This appendix is the concrete realisation of the on-robot Tier-1 transport of §3.2 and of the typed `Robot` contract of §3.2. Where §B described the hardware-side platform list, the four-method contract, and the cost of onboarding a new embodiment, this appendix describes what runs *inside* those containers and the data-collection stack that ships on top. The argument back to §3.2 is that `nautilus-collect` is empirical evidence of modularity at the right grain — and, as a direct consequence, that the same code path serves both teleoperated data collection and closed-loop policy rollout, with only the action source swapped.

C.1 Modular node architecture

The runtime decomposes into five ROS 2 nodes, each owning exactly one device — arm, gripper, state aggregator, teleoperation reader, and hand-eye TF publisher. Each node prefixes its topics and services with a configurable namespace, which is the mechanism behind the bimanual variant referenced in §B: a bimanual rig is two single-arm node groups running under different namespaces with no code duplication, and the aggregator is configured by namespace list rather than by arm count. From the agent's perspective the namespace prefix is the only coupling between nodes.

C.2 ROS 2 interface schema

The wire-level instantiation of the typed contract of §3.2 comprises four entry families: a canonical `RobotState` message that, after the WebSocket layer of §3.2 strips the ROS 2 envelope, becomes exactly the `Observation` the agent sees; control-mode services that switch impedance modes in one call rather than as a code edit; kinematics services that expose IK and FK so agent-generated code never has to ship a solver of its own; and a long-running action with progress feedback, demonstrating the action-vs-service split. The schema is deliberately symmetric across embodiments — the bimanual driver, and the xArm, Allegro Hand, and Unitree H1 wrappers all speak it — which is why a single agent skill family can target every embodiment without per-platform branching.

C.3 Python module layer

The Python layer is the *narrow waist* of the platform: everything below it is ROS 2, everything above it is policy or agent code. It declares the four-method `Robot` contract of §3.2 (reproduced in §B) above the schema below, ships two Franka implementations in tree (single-arm and bimanual, each of order a few hundred lines, consistent with the integration scope claimed in §B), and exposes a Gym-compatible adapter that lets the same `Robot` face either a human teleoperator or a policy server. The only difference between the two clients is who calls `apply_action`, which is the structural reason that the data-collection stack of §C.4 and the closed-loop rollout share their entire infrastructure below the action source.

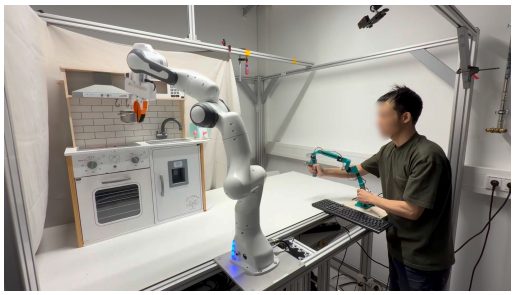
C.4 Data-collection stack

Data collection is the most exercised loop in `nautilus-collect`: the same stack has been used to gather manipulation trajectories on the Franka single-arm and bimanual lines that constitute the reference implementation in tree, and is reused at inference time with only the action source swapped from the human teleoperator to a policy server. We document the stack here because, in addition to being a useful artefact in its own right, it is the most operationally exercised piece of evidence for the modularity claim of §3.2.

Teleoperation. Two teleoperation modalities sit behind the same controller abstraction, so that the data-collection loop is agnostic to which device is on the operator end. **VR** drives Meta Quest 2/3 controllers in relative-pose mode with in-band episode boundaries and success/failure labels that the operator triggers mid-recording without leaving the rig — so the final flag lands at the same moment the episode ends, removing the post-hoc relabelling pass other open data-collection pipelines defer to. **GELLO** is a kinematically-similar leader arm whose joint stream is mapped one-to-one onto the follower; we use it for contact-rich tasks where the free-hand impedance of a held-in-hand controller



(a) VR teleoperation via Oculus Quest controllers.



(b) GELLO leader-arm teleoperation.

Figure 4: **Two teleoperation modalities currently supported by `nautilus-collect`.** Both devices sit behind the same controller abstraction (§C.4); the data-collection loop is unchanged across modalities. **(a)** Oculus Quest 2/3 driving the Franka reference platform in relative-pose mode with in-band success/failure labels. **(b)** A GELLO kinematically-similar leader arm: the operator's joint stream is mapped one-to-one onto the follower, which is preferable for contact-rich tasks where the free-hand impedance of a VR controller is a poor match.

is a poor match. Both controllers emit the same `Action`, so adding a third modality is a one-class extension. Figure 4 shows the two rigs side by side.

Synchronised observation. Per timestep the recorder snapshots joint state, end-effector pose, gripper state, multi-camera RGB-D, and a shared ROS-namespace timestamp. A short approximate-time-synchronisation window plus a small state cache absorb the jitter between the 50 Hz state stream and the 15–30 Hz camera stream without dropping samples.

Companion toolbox. Around the recording loop, `nautilus-collect` absorbs the side concerns of robot learning data collection so they do not leak into policy code. The argument is operational: every utility we found ourselves wanting in our own runs — TFDS/RLDS [96] export under the field names of the public DROID release [87] so DROID-trained models load our trajectories without a custom loader, in-band success/failure labelling so episodes do not need a post-hoc relabelling pass, replay against the original `Robot` object so the controller's behaviour is its own regression test, and side-by-side visualisers for the raw and converted forms — ships with the platform rather than being rebuilt by each policy author. Hand-eye calibration must be repeated whenever a camera is moved; its Charuco-board result is published as static TF and consumed transparently by the camera reader, so downstream code never sees calibration as a parameter — it is part of the platform, not the policy.

C.5 Integration boundary with the agent

From above, the Tier-2 WebSocket "real environment" of §3.2 wraps the Gym-compatible adapter; the agent never imports a ROS 2 symbol, and sees only the `Observation` and `Action` schema converted from the messages of §C.2. The two-tier transport boundary is therefore exactly the boundary between what `nautilus-collect` ships and what the agent generates. This is what permits the robot-skill family of §3.3 to onboard a new embodiment by writing the four artefacts of the recipe in §B — a `Robot` subclass, a URDF, a ROS 2 driver wrapper, and a launch file — without any of those artefacts touching the WebSocket layer, the schema, or the agent itself. Modularity in `nautilus-collect` is operationally the property that the new-embodiment patch and the agent are disjoint.

C.6 Build, dependencies, and reproducibility

The control side depends symbol-for-symbol on Polymetis [95] (vendored as a git submodule because the upstream repository has been archived) plus libfranka; the perception side depends on the ZED SDK [97]. All three are version-pinned at the container layer described in §B, so the perception stack and the real-time stack rebuild independently. The Charuco calibration of §C.4 has to be redone whenever a camera is physically moved; the full list of hardware-side human work is given in Table 10.

Table 6: Six-layer implementation stack of the NAUTILUS plugin. **L1** is the commodity agent loop; **L2**–**L6** route requests, execute subagents, validate artefacts, read State, encode domain priors, and leave reproducibility records over the substrate surface.

Layer	Role in harness	Implements	Contents
L1	Commodity agent loop	External	Claude Code agent loop
L2	User-facing routing	Guides	Slash commands; skills
L3	Task-specialised execution	Guides	Isolated subagents (App. E)
L4	Validation and State boundary	Sensors / State / substrate	Runtime Sensors; MCP registry; contract & WebSocket utilities
L5	Domain reference library	Guides	Schemas, classification protocols, validation contracts
L6	Run record and hand-off	Workspace artefacts	Append-only logs; reproduction receipts

D Implementation details supporting the Method

This section collects implementation-level details that support §3 but are not necessary for the main conceptual argument. The main text presents NAUTILUS as a substrate (typed contracts, chambered execution, uniform transport) plus a content layer (Guides, Sensors, State, and workspace artefacts). The details below document how that anatomy is realised in the plugin, how runs are recorded, how registry entries are verified, and how hardware entries are pinned for reproducibility.

D.1 Content-layer implementation stack

The content layer is implemented as six layers around a commodity Claude Code loop. **L1** is external to NAUTILUS: it provides the base code-agent loop, file access, shell access, and conversational state. NAUTILUS contributes **L2**–**L6**. **L2** exposes user-facing slash commands and description-driven skills that route a request either to a deterministic script or to a subagent workflow. **L3** contains task-specialised subagents for environment probing, benchmark wrapping, policy serving, training, finetuning, and robot integration; each subagent runs in an isolated context and the main thread orchestrates dispatch. **L4** is the tool, validation, and registry boundary: lifecycle hooks, read-only MCP registry tools, contract utilities, and WebSocket utilities live here. **L5** is the domain knowledge library, including the InstallationPlan schema, the IL-vs-RL protocol, capability-matching rules, smoke contracts, and observation/action mapping rules. **L6** is the run-record layer that leaves process logs and user-facing receipts in the target repository.

These layers map onto the Method concepts as follows. **L2**, **L3**, and **L5** primarily implement Guides; **L4** implements runtime Sensors and State access; **L6** implements workspace artefacts. The important design choice is that robotics-specific knowledge is not hidden inside model weights or pair-specific glue code. It is exposed as inspectable schemas, protocols, contracts, and receipts that can be reviewed and extended independently.

D.2 Workspace artefacts

Every NAUTILUS run leaves two classes of artefacts in the target repository. *Process logs* (`<repo>/.nautilus/run-log/NN-<task>.md`) are append-only engineering records. Each names the tool, subagent, command, and one-line result, so an auditor or re-runner can reconstruct the run in chronological order. *Receipts* are end-of-run summaries written to the repository root, such as `install.md`, `history.md`, and `benchmark.md`. They are regenerated on every run and serve as the hand-off document for a human user: installation recipe, probe-and-decision narrative, and benchmark task/protocol guide.

The split is deliberate. Process logs are low-level engineering evidence, while receipts are concise reproduction instructions. Mixing the two produces records that satisfy neither audience: too noisy for a user, too lossy for an auditor.

D.3 Registry sharing and benchmark verification

Registry mutation is intentionally excluded from the runtime MCP server. A user can package a policy, benchmark, or robot as an `unverified` registry entry with pinned source and environment artefacts. These entries are already useful for sharing: another user can reproduce the same setup from the recorded commit, image ID, contract spec, and run receipt.

NAUTILUS maintains `verified` benchmark entries because benchmark protocols grow more slowly than policy configurations. Verification is based on *cross-policy evidence*: for a fixed benchmark wrapper, multiple released policies—each replayed with its official checkpoint—should reproduce their published reference results, with deviations recorded transparently rather than converted into a universal pass/fail threshold. The verification command fills reproducibility metadata such as Docker image ID and image size, while entries that build but lack sufficient cross-policy evidence remain `unverified`.

For a verified entry, registry pre-flight can return a short `quick_start` recipe (clone, pull the pinned image, compose up) instead of re-running the full probe–render–build–classify path. The registry therefore acts as a shareable cache of reproducible contracts, not as mutable agent memory.

D.4 Hardware artefact pinning

Hardware entries follow the same verified-registry pattern as policies and benchmarks. Each hardware entry pins three artefacts: the `nautilus-collect` source commit implementing the Robot contract, the corresponding `nautilus-collect` Docker image, and a platform-specific driver-stack image. The driver-stack image owns the vendor-side dependency closure, such as Franka's `libfranka` plus Polymetis, UFactory xArm's Python SDK, the Allegro Hand ROS 2 driver package, or Unitree H1's `unitree_sdk2` plus `unitree_ros2`. The platform container above the driver stack consumes only the standard Robot contract and the Tier-2 policy endpoint, so policy-side code is not rewritten when the hardware container changes.

This is the hardware analogue of the benchmark and policy registry: the agent looks up a verified contract and pinned artefacts rather than hand-wiring a driver stack from scratch. Adding a new embodiment is therefore framed as implementing and verifying a Robot contract entry, not as modifying policy or benchmark code.

D.5 Surgical-pruning

For each released tuple (P_i, B_j, c_{ij}), where P_i is a public policy repository, B_j is a target benchmark, and c_{ij} is the released checkpoint used in the reference result, we construct a pruned repository $P_i^{\setminus B_j}$. We developed an additional agentic plugin to automatically prune the target-benchmark-specific support from P_i: example directories, benchmark adapters, task configs, dispatch entries, benchmark-specific scripts, documentation references, and other code paths whose purpose is to run P_i on B_j. The pruning is surgical rather than destructive: supports of other non-target benchmarks in the same repository $P_i^{\setminus B_j}$ must remain usable after pruning, so the integration test starts from a healthy policy repository with only the target integration missing. To make checkpoint loading possible without leaking B_j integration code, we leave a short checkpoint note in $P_i^{\setminus B_j}$ containing only high-level natural language descriptions of the non-trivial policy-specific facts when loading the checkpoint c_{ij}.

D.6 Trial isolation protocol

Pruning is only meaningful if the agent cannot recover the removed code by other means. We distinguish three leakage channels: runtime network access, persisted agent memory and state, and memorization in model weights. The first two are excluded by mechanism; the third cannot be excluded by any infrastructure, so our claim is scoped to an agent that reconstructs the integration under controlled access, not to a model that has never seen the public code.

Runtime network isolation. Each trial runs on an internal Docker network whose only egress is a deny-by-default CONNECT proxy. GitHub, code-search and web-search hosts, and telemetry endpoints are denied before any allowlist is applied. Per trial, the allowlist contains only the model-provider endpoint required by the agent and the package sources needed for environment setup;

Table 7: L2 slash commands exposed by the NAUTILUS plugin.

Command	Purpose
`> /nautilus:help`	print a user-facing overview of the plugin
`> /nautilus:benchmark`	browse, submit, or verify benchmark registry entries
`> /nautilus:policy`	browse, submit, or verify policy registry entries
`> /nautilus:robot`	browse robot registry entries
`> /nautilus:eval`	end-to-end cross-evaluation: clone both repos, build containers in parallel, render an obs/action adapter, run a one-episode smoke
`> /nautilus:train`	train a registered policy on a benchmark-native dataset (Mode A) or a LeRobot Hugging Face dataset (Mode B); 1-epoch smoke + detached full run

Table 8: L3 subagents. Each runs in an isolated context; the main thread orchestrates dispatch.

Subagent	Role
`env-generator`	probe a target repo, render base `docker/`, classify
`benchmark-generator`	extend an existing image for a sim benchmark; IL/RL classification, scaffolding, smoke test
`policy-generator`	wrap a checkpoint as a WebSocket server with multi-benchmark dispatch
`train-generator`	cross-pollinate a policy with a benchmark-native or LeRobot dataset; render mapper, training script, 1-epoch smoke
`model-finetune`	LoRA, freeze-backbone, or full-finetune scripts in PyTorch or JAX/Flax
`robot-generator`	interface-defined stub; full implementation deferred

repository access, when needed, is served by a local git-mirror sidecar that exposes exactly one repository. A pre-flight script verifies proxy denial, DNS posture, and mirror identity from inside the container, archives its JSON report, and aborts the trial on failure.

Agent memory reset. All three memory levels are managed explicitly per trial. *User level:* the CLI configuration directory (`CLAUDE_CONFIG_DIR`) is redirected to a fresh directory inside the trial's report tree; the host user profile, including accumulated memory, session history, and cached project state, is never mounted. *Session level:* every trial receives a fresh session identifier, and the per-trial configuration directory, including the session transcript, is archived for audit, so what the agent saw can be reconstructed; archives are never mounted into later trials. *Project level:* automatic project memory resolves inside the per-trial configuration directory and starts empty. In addition, an integrity gate scans every mounted tree before the first trial and fails the whole batch if any agent-instruction file is present (`CLAUDE.md`, `AGENTS.md`, `AGENTS.override.md`, `GEMINI.md`, `.cursorrules`, `.claude/`, `.codex/`, `.agents/`), since such files are read silently by coding CLIs and would act as untraced prompts. A content signature of each mounted tree is archived per trial. *Across trials:* workspace volumes are destroyed after each trial, so a later trial cannot inherit an earlier trial's solved workspace. Registry content that a configuration prescribes, such as the verified benchmark images of the full NAUTILUS setting, is an experimental input that differs between configurations by design; it is not accumulated agent memory.

E Subagent walkthroughs

This appendix supports §3.3 and §4.2. It is organised around Table 8: each L3 subagent gets its own subsection, followed by the cross-evaluation *orchestrator* that composes them. We flag the distinction up front: cross-evaluation (§E.5) is not itself a subagent — it is the workflow that wires `policy-generator`, `benchmark-generator`, and the State registry together via the `> /nautilus:eval` slash command. All subagents otherwise share the same recipe — *probe* the target repository, *render* a small set of templated files that wire it into the substrate of §3.2, and *smoke* the result against a fail-fast contract — and differ only in which substrate invariant they target and which artefacts they emit. Table 7 lists every slash command; Table 8 lists every subagent.

E.1 `env-generator`: base scaffolding and idempotent extension

The claim of `env-generator` is that *every other subagent can be written without knowing how the target repository was built*. It establishes this by reducing a heterogeneous source repository to a small file-system contract (the `docker/` directory plus a `.classification` label) that downstream subagents read but never re-derive. The substrate it pins is the one most likely to bite a generic harness: target repositories in this domain mix system libraries (Vulkan, MuJoCo, ROS), C++ extensions compiled against a specific CUDA toolchain, and Python packages with sharp version constraints, and no single package manager covers all three. We pair Docker with `uv` — Docker for the system and toolchain layer, `uv` for a single-pass Python resolution with reproducible lock files — and reject `conda`/`poetry`/`pipenv` on entry; the result is one version-controlled artefact that owns all three layers, which is what makes downstream composition safe.

Two design choices keep the subagents structurally decoupled. *First*, downstream subagents are coupled to `env-generator` only through three file-system contracts — the `docker/` layout, the `.classification` label ($\in$ {`plain`, `benchmark`, `policy`, `robot`}), and four named anchor comments embedded in the base templates — never through shared in-memory state. Every intermediate result is therefore independently inspectable and the pipeline can resume from any checkpoint, which matters when an agent run is interrupted at the boundary between two subagents. *Second*, every injection is wrapped in sentinel-guarded blocks of the form

```
# --- NAUTILUS OPEN: needs-vulkan ---
...
# --- NAUTILUS CLOSE: needs-vulkan ---
```

and the helper that performs an injection skips it whenever the opening sentinel already exists. This is what makes re-running any downstream subagent idempotent — a property we rely on whenever a smoke test fails partway through and the user edits and retries.

E.2 `benchmark-generator`: extending the base for sim benchmarks

Sim benchmarks fail in three structurally distinct layers — the container build, the reward function, and the policy–environment interface — and a generic end-to-end smoke conflates them, so the agent that has to retry on failure cannot tell which scaffold is broken. `benchmark-generator` is organised around catching each layer separately, which in turn forces an IL/RL split because the L3 probes for the two regimes share nothing.

The IL/RL classifier uses a single primary criterion — whether `step()` returns a meaningful reward signal — and falls back to secondary cues (policy architecture, PPO/SAC/TD3 imports, presence of offline datasets) only on borderline cases such as benchmarks that expose a reward function for compatibility but are conventionally evaluated under IL. RL targets are further split into a *dispatcher* mode (the repository's own training script is wrapped) and an *sb3* mode (a Hydra-driven `stable-baselines3` scaffold). The split is load-bearing because IL and RL targets demand different evaluation entry points and different smoke criteria; collapsing them would force every target into a least-common-denominator rollout that fails to catch either family's typical errors.

The smoke contract runs four checks in sequence and halts at the first failure, surfacing a typed error to the Sensor layer of §3.3. The argument for layering, rather than a flat end-to-end rollout, is diagnostic: an L1 failure implicates the build, L2 the reward function, and L3 the policy–environment interface, so the level at which the contract breaks tells the agent which scaffold to retry. L3-IL and L3-RL are mutually exclusive; the classification picks one.

Level	Test	Pass criterion
L1	`env.reset()` + 10 random steps	no exception; observation shape matches spec
L2	100-step rollout with reward logging	reward finite and non-constant
L3-IL	WebSocket handshake with a random-action policy server	client–server round-trip completes
L3-RL	100-step training run	loss finite and decreasing

E.3 `policy-generator`: wrapping a checkpoint as a WebSocket server

Wrapping an inference-only repository as a *policy container* (§3.2) has two problems that the WebSocket layer of §A.2 does not solve: locating an inference entry point in a repository that may not document one, and providing a smoke contract for VLA-class checkpoints too large to host on the scaffolding machine. `policy-generator`'s design lives entirely in those two corners. For the first, a scanner pairs candidate classes (those defining `forward`/`predict`/`infer`) with checkpoint loaders and cross-references the README; when heuristics disagree, it falls back to an interactive prompt rather than guessing, on the principle that a wrong entry point is worse than a paused agent. Once located, chunked outputs of shape $(H, \text{act_dim})$ with $H > 1$ are auto-wrapped behind an `ActionChunkBroker` so chunk-prediction policies serve the same per-step protocol as one-step policies and the benchmark side stays untouched.

For the second, the smoke contract admits three modes — `with_ckpt`, `mock_forward`, and `skipped` — and the middle one is load-bearing. If smoke required a real checkpoint, the contract would silently become opt-out for exactly the policies that are tens of gigabytes and most likely to expose integration failures. `mock_forward` monkey-patches `infer` to return a random action tensor of the declared shape and still exercises the full container lifecycle (start, healthz, accept obs, return action), so the WebSocket plumbing is verified even when the weights are absent.

E.4 `train-generator`: cross-pollinating policies with datasets

Cross-pollinating a wrapped policy with someone else's dataset would normally require hand-tuning three things at once: the dataloader to the policy's training-time constructor, the dataset's gripper sign convention to the benchmark's, and the gap between the inference-only and training-time constructors of the policy class. `train-generator` avoids all three by running introspection inside the *running* policy container — the chambered execution surface of §3.2 — rather than in the agent's main thread, and reusing the cross-evaluation mapping rules at data-loading time. Three probes there read the dataloader's obs/action schema, the training-side hyperparameters from the policy repository (with README defaults as fallback), and the training-time constructor signature. The dataset and policy specs are then fed through the same five mapping rules that cross-evaluation uses (§E.5), but applied at data-loading time, so the inference-time adapter and the training-time mapper share their generator.

Two dataset modes cover the empirical access patterns we have seen. *Mode A* bind-mounts the benchmark's native dataset into the container and reuses the benchmark's own dataloader, which preserves benchmark-specific conventions such as gripper sign. *Mode B* installs `lerobot` inside the container and pulls a Hugging Face dataset, unlocking the LeRobot ecosystem at the cost of a Python ≥ 3.12 requirement that the spec-capture script pre-checks before any rebuild. The choice is made by the `benchmark=/lerobot=` flag on `> /nautilus:train`. The training entry point is a thin wrapper that imports the policy with probed constructor arguments, injects the mapper as a data transform, and calls the policy's own `train` method rather than reinventing a training loop. The 1-epoch smoke is capped at $\sim$8 steps and 30 minutes, on the same diagnostic principle as the benchmark smoke ladder: a short run that verifies loss is finite and decreasing, that a checkpoint was written, and that GPU memory stayed in budget is enough to catch the integration failures that would otherwise surface 12 hours into a full run.

E.5 Cross-evaluation: orchestrator across subagents

Cross-evaluation is the workflow that operationalises the $\Theta(N+M)$ collapse claim of §3.3: it takes any two registry entries and produces a running rollout without writing pair-specific glue. The defining choice is that compatibility is decided *before* any container starts, by comparing the two registry specs (obs keys, action dim, gripper convention) and bucketing the pair as *native* (specs match, run directly), *compatible-zero-shot* (structurally compatible after a few well-defined transformations, render an adapter), or *incompatible-action* (fundamentally mismatched, abort with a typed error pointing to retraining). The argument for this pre-render gate is operational: a runtime mismatch between policy and benchmark surfaces silently as a `ConnectionRefused` or a shape error well after expensive containers have been brought up, whereas the spec-level check fails in seconds and tells the agent which side to fix. Around this gate, `> /nautilus:eval` resolves names through MCP `lookup_policy`/`lookup_benchmark` (§F), short-circuits already-verified entries to their published

images, otherwise dispatches `env-generator` plus the matching downstream subagent in parallel on each side, and finishes with a one-episode smoke that emits `cross-eval-report.md`.

For the *compatible-zero-shot* bucket, the adapter is auto-generated from the two spec JSONs using five rules in a fixed order: *key rename* (e.g. `agentview_rgb` → `image`), *chunk split*, *dim slice*, *dim pad* (zero-pad trailing dims when the policy expects a wider input than the benchmark exposes), and *image preprocess* (`uint8` HWC → `float32` CHW with normalisation). The adapter is injected as a preprocessing step inside the policy server's `infer`, leaving the benchmark side untouched, which preserves the property that the benchmark image we cache is the same image any other policy will be evaluated against. As a worked example, `> /nautilus:eval policy=openpi-pi0 benchmark=libero` classifies as *compatible-zero-shot* (LIBERO exposes `agentview_rgb` where OpenPI-π_0 expects `image`; action dims match), renders an adapter that combines a key rename with image preprocess, and runs one LIBERO episode — the only artefact the user sees is a Markdown report recording the bucket, the applied rules, the rollout outcome, and per-step inference latency from the WebSocket protocol's `server_timing` field.

F Typed registry and State layer

This appendix expands §3.3 into the schema, tools, curation protocol, and matching logic that together realise the State component of harness engineering at the robotics-research scale. The argument back to §3.3 is that the registry is a *contract carrier*: it stores enough machine-introspectable metadata that any policy and any benchmark can be composed without hand-written adapters, but it stores no code and no agent state, so its mutation surface is small and human-auditable.

F.1 Schema and MCP tools

The registry is split into an *index* layer that lists what is registered and a *spec* layer that records the per-entry contract metadata an agent needs to compose two entries (obs/action shapes and dtypes, reward structure for benchmarks, released checkpoint URLs for policies, optional training block for datasets). The split is what makes the registry small enough to audit during review: the index changes whenever someone adds an entry, but the spec for a given entry is rewritten only when its on-disk contract genuinely changes. JSON-schema validation runs at write time so a malformed entry never lands in the index.

Agents read the registry through a FastMCP server rather than the filesystem, and the server exposes only read-only tools. Two properties follow. First, the on-disk format can evolve without breaking agent code, since stability is guaranteed at the tool return shape, not at the YAML/JSON layout. Second — and this is why the registry is suited to be a contract carrier rather than a mutable store — there is no write API at runtime; the only way to mutate the verified registry is through a reviewed registry update.

F.2 Curation and fuzzy matching

The registry flow of §3.4 operationalises the boundary between the registry as machine-introspected data and the registry as human-attested record. A user's *submission* runs local pre-flights (clean git tree, Docker login, no name or URL collision in the index) and appends an `unverified` entry, deferring to the user to share the generated evidence package. A maintainer's *verification* checks reproducibility pins, re-runs the relevant smoke contracts inside the proposed container, and marks benchmark entries as `verified` only when independent evidence supports the same conclusion. The MCP server is deliberately read-only, so runtime agents cannot modify verified entries.

Lookup is tolerant by design, because users refer to repositories by names that the registry should not have to canonicalise on their behalf. A four-tier policy is applied in order — exact-name, exact-URL, URL-basename, and an aggressively normalised key match (lowercased, trailing digits stripped, non-alphanumerics removed) — and the matched tier is returned alongside the entry so downstream code can prefer high-confidence matches when disambiguation is needed. In practice `lookup_benchmark("maniskill")`, `lookup_benchmark("ManiSkill")`, and a fork URL all reach the same entry; what we surface to the agent is the tier at which each match landed.

F.3 Hardware registry parallel

Hardware platforms enter the registry through the same submission–verification flow as benchmarks and policies: the four currently registered platforms — Franka Panda (single-arm and bimanual), UFactory xArm, Allegro Hand, and Unitree H1 of Table 5 — each is represented by pinned source and container artefacts, with verification confirming that the hardware container builds and that the upward ROS 2 schema of §C.2 matches. The argument back to §3.4 is straightforward: from the agent's perspective the four-method `Robot` contract is what makes hardware addressable, and from the registry's perspective hardware carries no special-case logic, so a fifth platform enters NAUTILUS along the same path as a fifth benchmark or a fifth policy.

G Runtime sensor reference

This appendix is the long-form reference for the six runtime Sensors of §3.3, ordered earliest (before any tool runs) to latest (after deployment). A separate curation check (human attestation at registry promotion) lives at the curation boundary and is described in §3.4. Each Sensor is a narrow guardrail around one boundary where agent mistakes otherwise become expensive: shell execution, rendered configuration, captured interfaces, policy–benchmark composition, deployment smoke tests, and post-run cross-run verification. The common contract is that a Sensor does not add new task logic; it checks a machine-readable artefact or run result, then either lets the workflow continue, asks the agent to repair the artefact locally, or blocks the workflow until a human or registry update resolves the mismatch.

Table 9: The six runtime Sensors fired by NAUTILUS, ordered earliest to latest in the workflow. Registry verification is described in §3.4.

Sensor	Fires when	Checks	Response
Pre-action filter	before any tool call, via Claude Code's PreToolUse hook	Shell commands for destructive patterns such as `rm -rf /`, fork bombs, `mkfs` on a device, or `docker volume prune -force`.	Block the tool call and require a safer command.
Render-time auditor	before writing generated `docker-compose` files	Whether private registry URLs or verified images leak into the from-scratch reproduction path, where the default must remain rebuildable by the user.	Rewrite the rendered file or stop before the file is committed to the workspace.
Interface verify	during benchmark or robot capture	Whether observation, proprioception, and action specs are task-invariant and match the typed contract expected by §3.2.	Reject the captured spec and ask the agent to normalise the wrapper.
Spec-comparison gate	before any heavyweight run; registry-only, no docker	The policy spec against the benchmark or robot spec, including observation keys, dtypes, action shape, and control mode. The gate reports the native, compatible-zero-shot, or incompatible-action bucket.	Proceed, request an adapter, or block the run before container construction.
Tiered smoke ladder	at deployment, before scoring	Increasingly realistic execution checks: endpoint liveness, `reset/step`, finite observations and rewards, and IL/RL-specific gating such as a 1-epoch training smoke.	Keep repair local to the failing tier; only promote to scoring after the tier passes.
Cross-run verification	post-run, before verified benchmark status	Whether independent runs of the same benchmark wrapper support the same conclusion despite variation from seeds, checkpoints, and hardware.	Keep insufficiently supported entries `unverified` and record the evidence for later reproduction.

For the π_0–LIBERO trace in Figure 2, the spec-comparison gate is the first Sensor that uses both sides of the registry: it reads the generated π_0 Policy contract and the LIBERO Benchmark/Environment contract, then decides whether the pair can be run natively, needs an observation/action adapter, or should be blocked as incompatible. The later smoke ladder tests the same decision against a live WebSocket endpoint, and cross-run verification asks whether the benchmark wrapper has enough independent evidence to become a verified registry entry.

H Real-robot deployment details

This appendix expands the compact evidence block of §4.3. The main paper uses the hardware examples as a systems demonstration: unchanged policy code, unchanged WebSocket envelope, unchanged platform-side container, and a hardware-container swap. Here we spell out the two deployments: a VLA manipulation checkpoint on Franka and an RL-trained locomotion policy on H1, plus the collect–fine-tune–redeploy loop available when real-world adaptation is needed.

Franka manipulation setup. We take the $\pi_{0.5}$ checkpoint reproduced in §4.2 (whose wrapper score against LIBERO is reported in Table 2) and deploy it on a Franka Panda single-arm manipulator. The platform-side container is unchanged from the simulation rollout; the only swap is the hardware container of §B, released alongside the platform.

From benchmark wrapper to real-robot rollout. After the $\pi_{0.5}$ checkpoint is wrapped for benchmark evaluation, the same policy artefact is launched against the Franka hardware container. The launch file and policy artefact are unchanged across the simulation-to-real boundary, and no policy-code edits are required.

Fine-tuning recovery. The released $\pi_{0.5}$ checkpoint does not zero-shot complete pick-and-place in our real setup. When adaptation is needed, performance recovery uses the same NAUTILUS workflow: collect demonstrations through §C.4, fine-tune the checkpoint, and redeploy through the same launch path. The important systems point is that data collection and deployment share the same code path below the action source: the human teleoperator and policy server both call into the same `Robot` interface, with only the actor producing `apply_action` changing.

Unitree H1 locomotion setup. We deploy an RL-trained locomotion policy on a Unitree H1 humanoid. Unlike the Franka case, this is not a VLA manipulation checkpoint; it is a different policy family and a different control regime. The deployment nevertheless uses the same policy-facing WebSocket endpoint and the same platform-side container that drove the Franka deployment; only the H1 hardware container and launch namespace change.

H1 deployment. The H1 example is not intended as a state-of-the-art locomotion benchmark. Its purpose is to show that the same deployment substrate that drove a single-arm VLA manipulation rollout also supports an RL-trained legged-locomotion rollout without changes to the platform container, WebSocket envelope, or policy-side integration code.

Human/agent division of labor. Table 10 states who does what in the two deployments, and what a robot that is not yet registered would require. In these demonstrations, the agent's integration work ended at the simulation pass; operators prepared and launched hardware execution. On Franka, the time from simulation pass to the first real-arm rollout was 1.5 h, mostly spent on Docker image build and pull, with zero changes to policy code; the H1 deployment follows the same division. We do not claim real-robot automation or a reduction of hardware-side human labor.

I Full Trust Verification Experiment

Table 11 summarizes our full trust verification study across five trust-validated robot learning benchmarks. Each entry reports our reproduced score followed by the reference score reported by the corresponding official paper, repository, or benchmark leaderboard. Unless otherwise stated, all scores are success rates in percentage points. We use the officially released checkpoints whenever available; otherwise, we follow the official training and evaluation protocol and report the result from the corresponding reproduced checkpoint.

Table 10: **Human/agent division of labor in real-robot deployment.**

Stage	Agent	Human
Policy wrapping and benchmark evaluation	Wraps the checkpoint behind the Policy interface and runs the Sensors and benchmark evaluation up to the simulation pass	Issues the request and reviews the receipts
Hardware setup (registered robot)	—	Connects the hardware, calibrates it (hand-eye calibration on Franka), and starts the hardware container
Real-robot rollout	—	Launches the rollout through the unchanged launch path and supervises safety
Adaptation	—	Collects and labels demonstrations and runs fine-tuning
Onboarding a new robot	Scaffolds the `Robot` subclass, URDF, ROS 2 driver node, and launch file (§B)	Maps SDK calls, sets limits and gains, handles faults, and tests on hardware (typically a few hundred lines); registration makes this work reusable

LIBERO. For LIBERO, we evaluate on the four standard suites: LIBERO-Spatial, LIBERO-Object, LIBERO-Goal, and LIBERO-Long. The reference numbers for Diffusion Policy and OpenVLA are taken from the OpenVLA LIBERO simulation benchmark. OpenVLA reports results averaged over three random seeds, with 500 rollouts per seed, corresponding to 10 tasks and 50 rollouts per task. The OpenVLA checkpoints are the official LoRA-finetuned checkpoints for the four LIBERO suites, with LoRA rank 32; the evaluation uses center cropping because random crop augmentation was used during fine-tuning. For $\pi_{0.5}$, we use the official OpenPI LIBERO fine-tuning setup and checkpoint `pi05_libero`; the official OpenPI repository provides the LIBERO fine-tuning workflow by converting LIBERO data to LeRobot format, computing normalization statistics, and launching the `pi05_libero` training configuration. For SmolVLA, we follow the official SmolVLA simulation fine-tuning setup, where only the action expert is trained while the VLM backbone is frozen; the paper reports 100k fine-tuning steps with batch size 64 for simulation benchmarks and uses 10 flow-matching inference steps.

RoboCasa. For RoboCasa, we follow the RoboCasa365 multi-task learning benchmark. The benchmark trains language-conditioned policies on the human pretraining split, which contains 300 tasks, including 65 atomic tasks and 235 composite tasks, with 100 demonstrations per task. Evaluation is performed on the `atomic_seen`, `composite_seen`, and `composite_unseen` splits in pretraining scenes. The official Diffusion Policy baseline uses batch size 192 and is trained for 250k steps. The official OpenPI π_0 and $\pi_{0.5}$ baselines use full fine-tuning with batch size 64 and are trained for 75k steps. The RoboCasa365 simulator runs at 20 Hz and uses an operational-space controller with a 12-dimensional action space: seven dimensions for the arm, i.e., 3D translation, 3D rotation, and gripper control, plus five dimensions for mobile base translation, rotation, torso height, and mobile-base control gating. We note that the Diffusion Policy reference value of 15.7 corresponds to the Atomic-Seen split, whereas the official overall average for Diffusion Policy is 6.1. In contrast, the π_0 and $\pi_{0.5}$ values in Table 11 use the official average across the three RoboCasa365 evaluation splits.

ManiSkill. For ManiSkill, we follow the simulation benchmark released with the RDT repository. The benchmark evaluates five ManiSkill tasks: `PegInsertionSide`, `PickCube`, `StackCube`, `PlugCharger`, and `PushCube`. The official protocol generates 5,000 demonstration trajectories using motion planning. For OpenVLA and Octo, the original absolute joint-position-control trajectories are converted to delta end-effector pose control to match their action spaces; for RDT and Diffusion Policy, joint-position-control data are used. Diffusion Policy is trained from scratch for 1000 epochs, and the epoch-700 checkpoint is selected based on the lowest validation sample loss. OpenVLA is fine-tuned from the official pretrained checkpoint using LoRA rank 32. Each method is evaluated over 250 trials, consisting of 10 random seeds and 25 trials per seed.

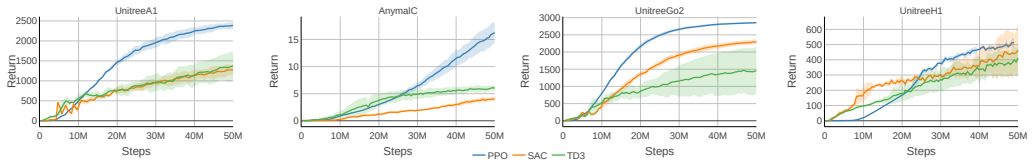


Figure 5: The performance benchmark on Loco-Mujoco with three classical RL algorithms, including PPO, SAC, and TD3.

RoboTwin. For RoboTwin, we report Easy and Hard success rates separately rather than mean and standard deviation. RoboTwin 2.0 evaluates 50 dual-arm manipulation tasks. All models are trained on 50 clean, non-randomized demonstrations per task. The Easy setting evaluates policies in clean environments, while the Hard setting evaluates the same policies under domain-randomized environments with clutter, lighting variation, table-height variation, background textures, and unseen language instructions. ACT is trained with chunk size 50, batch size 8, single-GPU training for 6,000 epochs, and temporal aggregation during deployment. Diffusion Policy is trained for 600 epochs with batch size 128 and planning horizon 8. π_0 is fine-tuned for 30,000 steps with batch size 32. We therefore report RoboTwin as "Easy success / Hard success" rather than using a $\pm$ notation.

ALOHA. For ALOHA, the reference scores are taken from the OpenVLA-OFT ALOHA experiments. These experiments use a bimanual ALOHA platform with two ViperX 300 S arms, three camera views, 14-dimensional joint-angle robot state, and target absolute joint-angle actions at 25 Hz. The ALOHA suite contains four real-world tasks: `fold shorts`, `fold shirt`, `scoop X into bowl`, and `put X into pot`. The demonstrations per task are 20, 30, 45, and 300 respectively, and the evaluation uses 10, 10, 12, and 24 trials respectively. The ALOHA metric is an aggregate task performance score based on a predefined partial-completion rubric rather than a single binary success indicator for all tasks. ACT and Diffusion Policy are trained from scratch on each task. ACT uses action chunks of 25 steps and is trained for 10k–70k epochs depending on the task. Diffusion Policy uses a 2-step observation history, predicts 24-step action chunks, and is trained for 30k–120k steps depending on the task. The π_0 reference is obtained by fine-tuning the official model with the default full fine-tuning recipe and training until convergence.

Loco-Mujoco (RL) We conduct additional experiments on Loco-Mujoco [98], an RL locomotion benchmark implemented in JAX that includes both humanoid and quadrupedal robots. We evaluate three classical algorithms: PPO [37], SAC [99], and TD3 [39]. Since Loco-Mujoco only provides a native implementation of PPO, we further incorporate SAC and TD3 from Stable-Baselines3 and reimplement them in JAX to demonstrate the adaptability of our framework. As shown in Fig. 5, all three algorithms successfully acquire locomotion skills. Moreover, for H1, we successfully transfer the learned policy to the real world.

Comparability and limitations. The goal of this experiment is to verify whether our reproduced results match the official reference numbers under the closest available protocol, rather than to claim a perfectly controlled cross-benchmark comparison. Several benchmarks differ in evaluation budget, action representation, robot embodiment, and metric definition. In particular, RoboTwin reports Easy and Hard success rates, RoboCasa reports both split-level and average success rates, and ALOHA reports rubric-based aggregate task performance. We therefore treat each benchmark independently and compare our reproduction only against the official protocol used for that benchmark.

J Full agentic-ablation tables

This appendix gives the per-pair breakdown of the ablation summarised in Table 3. Each of the three pruned policy–benchmark pairs forms a separate row group, and within each group the four settings (vanilla coding agent, full NAUTILUS, w/o template, w/o verified image) are evaluated under the same column schema as the main table: **wrap success** and **task success** (the Pass rate of our metrics), **agent inference time**, **interaction budget** (total agent turns), **token/context budget** (total token

Table 11: **Benchmark Comparison.** Each cell reports our reproduced success rate / the reference success rate reported in the original paper. For RoboTwin, values are written as *easy (hard)*, where the number outside the parentheses is the easy-task success rate and the number in parentheses is the hard-task success rate.

	Benchmark				
Method	LIBERO	RoboCasa	ManiSkill	RoboTwin	ALOHA
Diffusion Policy	70.2 / 72.4	15 / 15.7	32.4 / 30.2	26.4 (0.42) / 28.04 (0.64)	75.8 / 77.5
ACT	– / –	– / –	– / –	30.0 (1.64) / 29.74 (1.74)	72.8 / 72.3
π_0	93.6 / 94.2	15.2 / 14.8	– / –	42.64 (18.62) / 46.42 (16.34)	– / 83.9
$\pi_{0.5}$	97.0 / 96.85	18.6 / 16.9	– / –	– / –	– / –
OpenVLA	78.2 / 76.5	– / –	4.0 / 4.8	– / –	– / –
SmolVLA	88.2 / 87.3	– / –	– / –	– / –	– / –

Table 12: **Per-pair ablation breakdown (Claude Opus 4.7 / Claude Sonnet 4.6).** Same column schema as Table 3. Each pair is denoted $\mathrm{Policy}^{\setminus \mathrm{Benchmark}}$, where the superscript $\setminus \mathrm{Bench}$ marks the benchmark whose support has been surgically removed from the policy repository: $\mathrm{OpenPI}^{\setminus \mathrm{LIBERO}}$, $\mathrm{OpenPI}^{\setminus \mathrm{ALOHA}}$, and $\mathrm{OpenVLA}^{\setminus \mathrm{LIBERO}}$. The row groups correspond to these pruned pairs. Every cell reports paired values Claude Opus 4.7 / Claude Sonnet 4.6. The *w/o template* row is omitted for both OpenPI repositories under both models because the upstream openpi codebase ships its own Tier-2 WebSocket implementation between the policy container and the environment endpoint (§3.2), which leaks the inter-module-transport prior that the *w/o template* ablation is designed to remove from the agent's context. Sonnet 4.6 was not run under the *w/o template* ablation for any pair (Sonnet side reported as "–").

	Agent success (↑)		**Agent inference**	**Interaction budget** (↓)	**Token/context budget** (↓)		**Cost**
Setting	**Wrap**	**Task**	**Time**	**Agent turns**	**Tokens**	**Peak ctx**	**USD**
$\mathrm{OpenPI}^{\setminus \mathrm{LIBERO}}$							
Vanilla agent	100.0% / 100.0%	95.0% / 90.0%	21m29s / 13m47s	199.1 / 137.0	15.1M / 12.4M	136.2k / 140.7k	$6.17 / $5.37
NAUTILUS	100.0% / 100.0%	100.0% / 100.0%	26m35s / 13m27s	185.5 / 93.0	17.8M / 7.6M	166.7k / 134.8k	$7.75 / $3.89
w/o template	– / –	– / –	– / –	– / –	– / –	– / –	– / –
w/o verified image	100.0% / 100.0%	100.0% / 75.0%	29m14s / 18m53s	416.3 / 363.0	16.1M / 12.8M	145.9k / 141.3k	$12.83 / $11.76
$\mathrm{OpenPI}^{\setminus \mathrm{ALOHA}}$							
Vanilla agent	100.0% / 100.0%	100.0% / 96.0%	12m53s / 8m36s	134.5 / 118.0	9.7M / 5.9M	111.1k / 97.8k	$4.07 / $2.57
NAUTILUS	100.0% / 100.0%	100.0% / 100.0%	18m31s / 8m41s	127.0 / 57.5	10.9M / 4.2M	135.9k / 108.2k	$5.02 / $2.49
w/o template	– / –	– / –	– / –	– / –	– / –	– / –	– / –
w/o verified image	– / 100.0%	– / 100.0%	– / 12m27s	– / 212.0	– / 6.7M	– / 106.5k	– / $7.74
$\mathrm{OpenVLA}^{\setminus \mathrm{LIBERO}}$							
Vanilla agent	100.0% / 96.4%	91.2% / 93.6%	15m08s / 20m41s	152.0 / 219.0	11.9M / 19.0M	122.1k / 151.6k	$5.05 / $7.62
NAUTILUS	100.0% / 100.0%	96.1% / 94.0%	26m26s / 28m52s	144.8 / 196.0	12.6M / 16.7M	153.6k / 164.9k	$6.32 / $7.91
w/o template	55.3% / –	55.1% / –	26m00s / –	607.5 / –	41.7M / –	219.2k / –	$74.94 / –
w/o verified image	100.0% / 100.0%	75.2% / 62.6%	21m22s / 49m08s	446.0 / 615.5	16.4M / 27.2M	125.0k / 150.5k	$16.77 / $37.64

footprint and peak context length), and **cost** in metered USD. Each cell aggregates 10 independent trials per setting; time, interaction, token, context, and cost columns are averaged over accepted trials.

K Failure analysis of integration trials

A trial fails when it does not reach task success within its 2 h wall-clock budget. Full transcripts are retained for every trial, and the two cases below come from our archived per-trial failure analyses. The failures fall into three categories, which appear both with and without NAUTILUS:

(i) *Contract mismatches*, e.g., a run that passed the WebSocket handshake but then failed on JSON/msgpack array encoding.

(ii) *Silent semantic errors*, such as mismatched observation preprocessing or action definitions, which raise no error message.

(iii) *Environment/dependency failures* while building or running the containers.

Most failures arise during task execution. Repairing them consists mainly of re-running evaluations, inspecting evaluation logs, and adjusting choices such as image preprocessing and action normalization, which is why failed runs are expensive. Two representative cases follow.

Case 1: silent preprocessing mismatch (category ii). The agent brought up both sides cleanly: the handshake succeeded, dtype/shape/range checks passed, episodes ran to their step limits, and success was uniformly 0.00. The cause was a duplicated 180° image rotation: the policy's preprocessing already un-rotates the simulator's frames, and the agent's wrapper rotated them a second time, shifting the visual distribution without tripping any interface check. Every observable proxy is green, and repair requires reconciling preprocessing semantics across two repositories rather than fixing an explicit error. The run consumed its full 2 h budget with a benchmark success rate of zero.

Case 2: episode bookkeeping (category ii). Uniform zero scores that looked like a policy failure were caused by trial accounting: the environment's registered 300-step horizon silently overrode the intended 600-step cap, and in a variant, an extra `env.step()` consumed two simulator steps per logical control step. Episodes therefore ended before the task could be completed. Diagnosis requires outcome-aware reasoning ("the score is implausible for this policy") rather than reading an error message, and agents lacking it spent the remaining budget re-verifying healthy components.

What the harness changes. These are common integration errors, not failures specific to either configuration. NAUTILUS does not prevent them; its typed contracts and Sensors move contract-level errors to early, cheap checks, whereas silent semantic errors that pass every interface check are exposed only by the benchmark score.

L Broader Impacts

NAUTILUS is open-source research infrastructure that sits between coding agents and the policy/benchmark/robot ecosystem. We discuss its broader impacts under two headings: positive and negative.

Positive impacts. NAUTILUS directly lowers the engineering tax of robot learning research. Cross-family reproduction, fair evaluation, and deployment to real hardware no longer require a fresh integration project for every (P, B, R) triple, which reduces duplicated effort across labs and makes rigorous benchmarking more accessible to small groups that cannot afford a full infrastructure team. Because every registry entry is pinned to a commit hash and a container-image digest, third parties can verify published results without trusting authors' local configurations, which we expect to improve the reliability of the empirical literature over time. Beyond engineering, the same substrate—typed contracts, chambered execution, and the State registry—is a starting point for *robotic auto-research*: agents that not only run pre-specified workflows but also propose experiments, execute ablations, interpret results, and refine hypotheses end-to-end. We see this autonomous-research direction as the most consequential positive externality of the substrate, and we describe it as the top-priority extension in Section 5.

Negative impacts and mitigations. The same automation that lowers reproduction cost also lowers the marginal cost of deploying immature or unsafe policies on physical hardware. An agent that can spin up a hardware container and run a checkpoint end-to-end is, by construction, an agent that can do so for a checkpoint that has not been adequately validated. Robot-learning workflows further sit in a domain with non-trivial physical-safety implications. We address these risks at the substrate level through two mechanisms documented in the main paper. First, every workflow stage runs inside a chambered execution container with explicit input and output contracts, so policy-side faults cannot silently reach hardware drivers; the simulation-pass-then-deploy ordering of §4.3 is part of this discipline. Second, verification Sensors (§3.3, Appendix G) gate every transition between stages on machine-checkable evidence rather than agent self-reports. We also flag two mitigations as planned rather than complete: human-in-the-loop modes for high-stakes deployment, and broader hardware-side safeguards beyond the two embodiment classes validated here. Both are discussed in Section 5 as priority follow-on work.

Export control. We see no direct export-control exposure from NAUTILUS itself. The released harness consists of skills, typed contracts, registry entries, and container recipes; it contains no model weights, controlled algorithms, or hardware designs, and these artefacts are not tied to a particular LLM provider. Third-party policies, checkpoints, and benchmarks stay at their original sources,

which the registry references by commit hash and image digest, and remain subject to their own licenses and terms.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 state four contributions: a modular substrate that reduces $(P \times B \times R)$ integration cost, reusable agent skills distilled from robotics priors, an open release spanning four policy families, six benchmark suites, and four embodiment classes, and an extensible registry. Each is supported in Sections 3–4.3, including the trust-validation table (Table 2), the ablation testbed, and the Franka and Unitree H1 deployments.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5 contains an explicit limitations paragraph that names the boundaries of the current system: (1) workflow coverage is limited to reproduction, evaluation, deployment, and initial fine-tuning support, with broader research workflows depending on extending the skills library; (2) cross-simulator unification is at the environment-API level rather than the physics layer, so simulator-specific effects must still be reported per simulator; (3) Nautilus is implemented as a plugin on a commodity coding-agent harness and inherits some upstream platform churn; (4) silent semantic errors that pass every interface check are exposed only by the task-level score; (5) the isolation protocol (Appendix D.6) cannot exclude memorization of public code in model weights, and pruning tests the reconstruction of a removed integration rather than onboarding a genuinely unseen repository; (6) on real hardware, setup, calibration, and safety remain human work (Table 10).

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.

- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not present formal theorems, lemmas, or proofs. The complexity statement that a shared harness reduces integration cost from $\Theta(N{\cdot}M{\cdot}K)$ to $\Theta(N{+}M{+}K)$ in Section 1 is an informal engineering argument used as motivation, not a theoretical result.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides the information needed to reproduce its main results. Section 3 fully specifies the typed contracts, registry schema, and skill structure. Section 4.1 lists every (policy, benchmark, checkpoint) tuple used for trust validation along with the comparison protocol against published references (Table 2). Section 4.2 defines the leave-one-benchmark-out testbed, the three pruned policy–benchmark pairs, the agent configurations being ablated, and the 10-trial-per-pair protocol. Hardware setup, data-collection pipeline, and deployment containers for the Franka and Unitree H1 results are documented in Appendix B. Code, registry, pruned testbeds, and pinned container images are publicly released (project website: https://yufengjin.github.io/nautilus/).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed

instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Both NAUTILUS and `nautilus-collect` are released in full as open source, linked from the project website (https://yufengjin.github.io/nautilus/). The release includes the platform, skills library, registry schema, the three pruned policy–benchmark testbeds, and run scripts that reproduce the trust-validation and ablation tables; the README documents the exact commands and container images required for each experiment. Every registry entry is pinned to a GitHub commit hash and a container-image digest so that downstream users can reproduce results bit-for-bit. Existing third-party benchmarks and policies used in the paper remain at their upstream sources, and we provide download/setup scripts rather than redistributing them.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Section 4.1 specifies, for every entry in Table 2, the policy, the released checkpoint, the benchmark and task split, the protocol for comparing against the published reference, and the deviation Δ. Section 4.2 specifies the three pruned policy–benchmark pairs, the agent configurations under ablation, the 10 independent trials per pair, the isolated sandbox in which each trial runs, and the metrics reported (wrap success, task success, agent inference time, interaction budget, token/context budget, USD cost). Hyperparameters and per-pair breakdowns of the ablation, together with the hardware deployment configuration for Section 4.3, are deferred to Appendix B and the per-task ablation tables.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: For the ablation in Section 4.2, every (policy–benchmark, agent-configuration) cell is run for 10 independent trials and the reported numbers are means over those trials, with the variability source being agent-side stochasticity in the coding agent's generations under fixed temperature. For the benchmark trust-validation in Section 4.1, we report the per-row deviation Δ between our wrapped success rate and the value in the original publication rather than imposing a fixed pass/fail threshold, because the source publications do not report a common across-seed variance that would let us define a uniform significance test; this is stated explicitly in the protocol.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Compute resources are documented for each class of experiment. All trust-validation and ablation runs (Sections 4.1 and 4.2) were executed on two single-GPU workstations: one with an NVIDIA RTX 4090 and one with an NVIDIA RTX 5090. Per-trial wall-clock times per (policy, benchmark) pair are listed in Appendix B. Agent-side cost is reported directly in the main text: Table 3 aggregates agent inference time, assistant turns, action turns, total tokens, peak context length, and metered USD cost per configuration. Model fine-tuning experiments were instead run on a multi-GPU internal cluster; the cluster configuration is described in Appendix B. For the real-robot experiments, the Franka and Unitree H1 control PCs and timings are described in Appendix H, including the 1.5-hour simulation-pass-to-first-rollout figure reported in Section 4.3.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

Answer: [Yes]

Justification: The work conforms with the NeurIPS Code of Ethics. The contribution is open-source research infrastructure for robot learning; it does not involve human subjects, personally identifiable data, or scraped content. All third-party benchmarks, policies, and code we wrap are used in accordance with their licenses (see item 12). Real-robot demonstrations were performed on lab-owned hardware (Franka Panda and Unitree H1) under standard laboratory safety procedures, with no participants other than the authors operating the equipment.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Broader impacts are discussed in Appendix L. On the positive side, NAUTILUS lowers the engineering barrier for cross-family reproduction and fair evaluation in robot learning, which we expect to accelerate community-driven validation, reduce duplicated integration work, and make rigorous benchmarking accessible to smaller labs; more broadly, the same substrate is a starting point for autonomous robot-learning research, where agents propose, run, and analyse experiments end-to-end, an enabling step toward robotic auto-research. On the negative side, the harness can in principle accelerate the deployment of immature policies onto physical hardware, and it lowers the marginal cost of automating robot-learning workflows that have downstream safety implications. We mitigate these risks through chambered execution and explicit verification gates in the substrate, and discuss human-in-the-loop deployment modes for high-stakes hardware as a planned extension in Section 5.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper releases research infrastructure (the NAUTILUS harness, skills library, registry, and pruned testbeds), not pre-trained language models, image generators, or scraped datasets. The harness wraps existing third-party policies and benchmarks but does not redistribute new high-misuse-risk artefacts; downstream policies remain under their original authors' release terms. Operational safeguards relevant to robot deployment (chambered execution, verification Sensors) are described in the broader-impacts discussion (Appendix L).

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Both NAUTILUS and `nautilus-collect` are released as fully open source under the MIT License. Every third-party asset used in the paper is open-source and is cited at first mention in the main text, including the policy implementations (Diffusion Policy,

ACT, π_0, $\pi_{0.5}$, OpenVLA, OpenPI, SmolVLA, cosmos-policy, Motus), the benchmark suites (LIBERO, ManiSkill, RoboCasa, RoboTwin, ALOHA), and the upstream platforms and agents (LeRobot, Isaac Lab, robosuite, Claude Code, OpenHands, SWE-agent). Each asset is used within the scope of its upstream license, and we credit each at the citation it appears under. We do not redistribute third-party weights or datasets; setup scripts pull each asset from its original source.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The paper introduces NAUTILUS and `nautilus-collect` as new assets. Both are released as open source under the MIT License, with the documentation packaged alongside the code: a README that lists installation and reproduction commands, schema documentation for the typed contracts (Appendix A), per-skill `SKILL.md` files for the agent-side guides (Appendix E), and the registry schema and curation protocol (Appendix F). The three pruned policy–benchmark testbeds used for the ablation are released together with their construction scripts and the released checkpoints they target, so that users can reproduce the leave-one-benchmark-out study. The release pins a commit hash and a container-image digest for every registry entry.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects. All evaluations use public benchmark task suites with their released task definitions, and the real-robot demonstrations on Franka and Unitree H1 were operated by the authors on lab-owned hardware with no external participants.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve research with human subjects, so IRB review is not applicable. Real-robot demonstrations on Franka and Unitree H1 were performed by the authors in a controlled laboratory setting under standard hardware-safety procedures.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a non-standard, central component of NAUTILUS: the system is an agentic harness whose runtime loop is driven by a coding-agent LLM that reasons over the typed contracts, invokes the skills library, and produces the integration code measured in our experiments. Section 3 describes how the agent is exposed to Guides, Sensors, and the State registry, and explicitly treats the underlying LLM as a replaceable backbone of the harness. Section 4.2 reports per-configuration LLM cost (assistant turns, action-taking turns, total tokens, peak context length, and metered USD), and the registry, sandbox, and verification gates that constrain the LLM's behaviour are described in Appendices F and G. The reference implementation runs on Claude Code [16]; LLM use beyond this methodological role (e.g., light-touch writing or formatting assistance) does not bear on the scientific contribution.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.