CODING AGENTS AS TASK COMPILERS: FROM SPECIFICATIONS TO STATE-CENTRIC VISUAL CONTROL SPECIALISTS

Anonymous authors
Paper under double-blind review

ABSTRACT

In visual control, a coding agent can build the controller rather than be the controller. Using a general-purpose coding agent as the online controller requires repeated model calls during each episode, incurring latency and inference cost. We instead concentrate coding-agent computation in development and formulate *task compilation*: compiling a task or task family into a reusable specialist that thereafter executes independently through a fixed runtime interface. We instantiate this idea with *State-Centric Task Compilation* (SCTC), which organizes the specialist around a task-state and memory module $\mathbf{Z}$, a visual encoder E, and a control policy P. The state interface separates the information estimated from images, retained across time, and used for control, providing a compatibility boundary for goal-conditioned reuse and localized recompilation. Across the evaluated manipulation tasks and games, coding agents construct effective frozen specialists. In Craftax, a specialist compiled by Astra achieves similar mean reward to Astra playing the game directly, with about $70\times$ shorter episode runtime. In object pushing, updating only the visual encoder restores success under visual perturbations, while adding a generalization requirement to the specification improves performance on unseen shapes. In Minecraft, frozen specialists use screenshots and keyboard/mouse actions to craft a stone pickaxe within five minutes in unseen worlds. These results show that coding agents can turn task specifications into reusable visual control specialists with low execution overhead.

1 INTRODUCTION

General-purpose coding agents can write and execute code, then revise it using feedback (Yang et al., 2024; Wang et al., 2025). Recent agents also support computer use, allowing them to inspect screenshots and interact with graphical interfaces through keyboard and mouse actions (OpenAI, 2026b). When used as online visual controllers, these agents select actions through repeated model calls, adding latency and inference cost (Elmaaroufi et al., 2026).

We instead move these capabilities into development, allowing the agent to construct a visual control specialist that later executes independently. We formulate this separation as *task compilation*: task specifications define the objective and runtime interface, while development interaction guides specialist construction. The specialist runs without its constructor, amortizing construction cost across executions. This perspective connects language-generated policies (Liang et al., 2022), reusable skills (Wang et al., 2023), executable world models (Tang et al., 2024), and feedback-guided controller revision (Elmaaroufi et al., 2026) through the separation of construction and execution.

This separation shifts attention to the construction problem: which information should connect visual observations to decisions? Consider making a stone pickaxe in Minecraft, our running example in Figure 1. Crafting recipes determine the required materials and intermediate tools, but the specialist must still locate a tree, retain its target during approach, and check whether mining added material to the inventory. Tracking the target, inventory validity, and task phase requires visual evidence, state-maintenance rules, and control procedures that use this information. The compiler must develop these components together.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

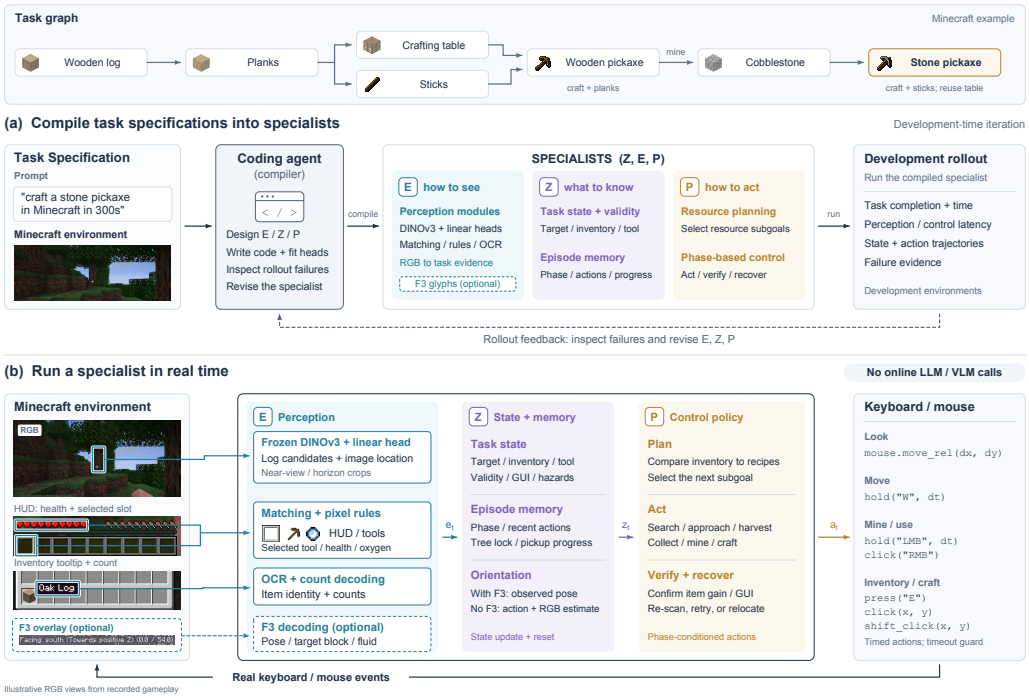


Figure 1: **A coding agent builds the controller rather than remaining in the control loop.** A state-centric task compilation process jointly develops task state, visual encoding, and control into a reusable specialist. At deployment, the implementation stays fixed while task state evolves, allowing the specialist to act independently of its constructor.

We introduce *State-Centric Task Compilation* (SCTC) to organize the compiled artifact around three coupled components (ZEP): a task-state and memory module $\mathbf{Z}$, a visual encoder E, and a control policy P. This design separates current visual evidence, maintained task state, and control through an explicit interface, allowing their implementations to vary while preserving the information expected by the retained components. State, encoding, and control may each be structured, learned, or hybrid.

This interface supports reuse across task instances and component revisions. A frozen pushing specialist can process new goal images through the same interface. An updated inventory reader can accommodate appearance changes while preserving state maintenance and crafting logic. These dependencies guide what to reuse and what to recompile.

Our contributions are threefold:

1. **Task compilation as a practical approach to visual control.** We find that coding agents can use task specifications and development interaction to construct executable visual control specialists that later act independently of their constructor. Across manipulation tasks and games, independently compiled artifacts achieve useful task performance after freezing.
2. **State-centric interfaces for reuse and localized recompilation.** We propose to organize the task-state and memory module ($\mathbf{Z}$), visual encoder (E), and control policy (P) around an explicit state interface. This interface provides a compatibility boundary: new runtime task inputs can reuse the frozen specialist, while perception can be replaced when the retained state and control components remain compatible.
3. **Reusable, inspectable, and efficient visual specialists.** Our experiments examine the practical benefits of task compilation: reuse across task instances and component revisions, inspectable implementations and development histories, and independent closed-loop execution with reduced deployment overhead. Compiled specialists execute locally, concen-

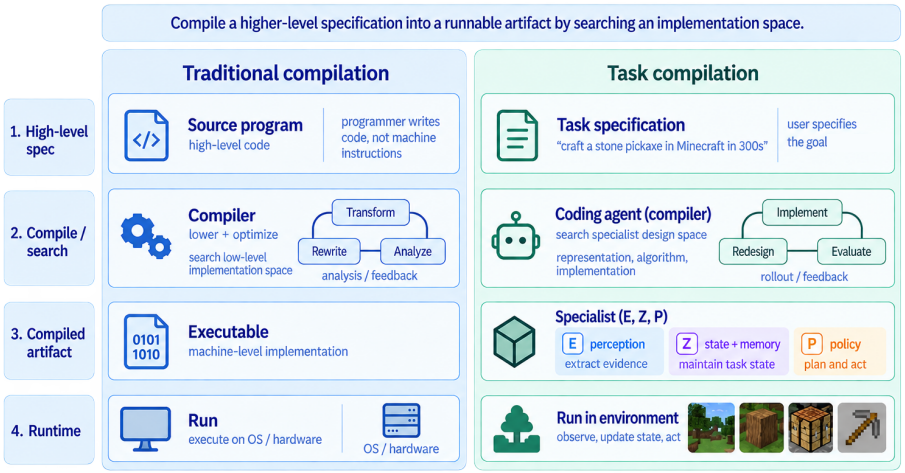


Figure 2: **Task compilation separates what to achieve from how to implement it.** The analogy is the abstraction boundary: users provide a program or task specification, while the compiler produces a reusable executable.

trating coding-agent computation in construction and subsequent revision and amortizing its cost across repeated deployments.

2 Task Compilation: From Specifications to Specialists

We define task compilation by separating construction from deployment. A compiler maps a task specification to a reusable specialist using one-shot synthesis, search, training, iterative interaction and revision, or combinations thereof.

2.1 A Task or Task Family Becomes a Reusable Specialist

Let σ specify a task or task family, including its objective and permitted runtime observation and action interface. A configured compiler C includes the construction-time resources, tools, and constraints under which it operates. Task compilation produces a frozen specialist,

$$\mathcal{S}^\star = C(\sigma). \tag{1}$$

After compilation, the specialist's code, weights, fixed assets, and update/reset rules are frozen; runtime inputs and episode state vary across executions.

A specification may describe either a fixed objective or a family of runtime goals. Let $\mathcal{G}_\sigma$ denote the admissible runtime goal space defined by σ, and let $g \in \mathcal{G}_\sigma$ be the goal supplied for a particular episode. Writing the history of image observations I_t and actions a_t as $h_t = (I_0, a_0, \ldots, I_t)$, the frozen specialist acts as a stateful controller

$$\mathcal{S}^\star : (h_t, g) \longmapsto a_t, \qquad g \in \mathcal{G}_\sigma. \tag{2}$$

For a fixed-objective task, $\mathcal{G}_\sigma$ can be a singleton and g may be omitted. For a goal-conditioned task family, different goals are supplied through the same runtime interface without changing the compiled implementation. For example, in PushObj, our object-pushing task family, the object shape and goal image vary across episodes, while the same frozen specialist controls each instance through the shared runtime interface without recompilation.

2.2 Compilation, Specialization, and Reuse

The compilation view emphasizes staging: general-purpose construction-time computation produces an artifact that is reused at deployment. Conventional ahead-of-time compilation transforms a source program into target code while preserving source-defined behavior (Leroy, 2009; LLVM Project, 2026a;b). Related mechanisms include partial evaluation, which specializes computation using information available before runtime (Futamura, 1999), and program synthesis, which constructs implementations from behavioral requirements (Gulwani et al., 2017). Task compilation applies this

separation to task specifications. The compiler chooses a task-specific representation, algorithm, and implementation, while later execution is handled by the frozen specialist rather than by the constructor.

Figure 2 highlights this analogy: conventional compilation lets programmers work with a higher-level source abstraction while repeatedly executing the resulting target program. Task compilation similarly moves general-purpose reasoning and iterative development out of the runtime loop, producing a reusable specialist that executes independently.

3 State-Centric Task Compilation for Visual Control

We introduce *State-Centric Task Compilation* (SCTC), a task compilation method that organizes a visual control specialist and its runtime computation around an explicit task-state interface.

3.1 A stateful compiled specialist

Visual observations often contain task-irrelevant detail, while control depends on a smaller set of task-relevant distinctions that may need to persist over time. We organize the compiled specialist internally as

$$\mathcal{S}^\star = (\mathbf{Z}, E, P). \tag{3}$$

Visual encoder E. The encoder maps the permitted image observation I_t and optional runtime goal g to visual evidence e_t. It may combine pretrained features from models such as DINOv3 (Siméoni et al., 2025), image-processing routines, optical character recognition (OCR), matching, or task-specific learned predictors. In Minecraft, for example, visual routines can read inventory evidence or propose a tree target.

Task-state and memory module Z. To maintain task information across steps, we define $\mathbf{Z} = (\mathcal{Z}, U_Z, z_{\text{init}})$. The state space $\mathcal{Z}$ defines the representation carried across time, U_Z updates it recursively, and z_{init} initializes episode memory. A runtime state is $z_t \in \mathcal{Z}$. All episode-specific information retained across steps and available to control belongs to $\mathbf{Z}$, wherever it is implemented. The representation may be structured, learned/latent, or hybrid. In Minecraft, one structured realization contains a retained tree target, inventory validity, and task phase.

Planning and control P. Using this maintained state and the runtime goal, the policy plans and selects actions. It may use explicit procedures, search, or learned models, including learned dynamics for planning.

At episode start, these components interact as follows:

$$e_0 = E(I_0; g), \qquad z_0 = z_{\text{init}}(e_0; g), \qquad a_0 = P(z_0; g).$$

For subsequent steps $t \geq 1$,

$$e_t = E(I_t; g), \qquad z_t = U_Z(z_{t-1}, e_t, a_{t-1}; g), \qquad a_t = P(z_t; g). \tag{4}$$

Thus, one runtime step computes

$$(z_{t-1}, I_t, a_{t-1}; g) \longmapsto (z_t, a_t)$$

through E, U_Z, and P. The compiled code, parameters, fixed assets, and update/reset rules remain fixed during deployment; the episode state evolves through this local loop.

Compilation process. Following the compilation–execution separation in Figure 2, Figure 3 illustrates the SCTC development loop. In this Craftax example, the specification asks for a controller that maximizes game return using RGB observations and native actions; the development setting supplies permitted resources, evaluation tasks, and a construction budget. The coding agent first builds a runnable specialist $(\mathbf{Z}, E, P)$ (v0), jointly designing visual extraction, state maintenance, and control. It then executes the specialist on development tasks and inspects episode returns, images, actions, and maintained state to identify failures. These findings guide revisions to the relevant

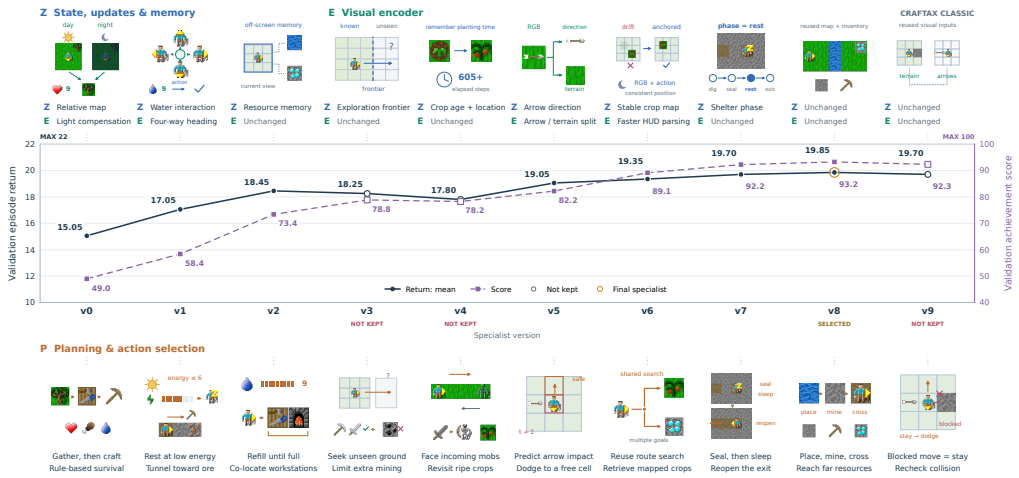


Figure 3: **Compiled Astra-2 improves through coordinated changes to perception, memory, and control.** Development failures guide revisions across these components. Mean return on 20 validation worlds determines which complete candidates are retained, so the deployed specialist accumulates validated improvements rather than every attempted change.

components and the information passed between them. Each candidate is compared with the accepted version; development improvements trigger validation, which determines whether to retain the revision or restore its parent. The successive versions in Figure 3 follow this implementation–evaluation–revision cycle within the construction budget. Finally, the validation-selected specialist, v8, is frozen and deployed independently of the coding agent, processing new observations and updating episode state through the fixed ZEP loop.

3.2 The State Interface as a Compatibility Boundary

The key interface carries information from visual encoding into state maintenance and from maintained state into control. Compatibility is functional: a replacement is compatible when it supplies the information expected by retained components through their existing interfaces. The interface may be explicit, learned, or hybrid. For example, in structured implementations, field meanings, update/reset rules, and producer/consumer links provide concrete diagnostics of this compatibility.

This compatibility boundary determines which components can remain frozen when runtime goals or perception change.

3.3 Runtime Goal Generalization and Local Recompilation

If σ defines a task family with goal space $\mathcal{G}_\sigma$, changing g within that space changes a runtime input rather than the implementation and therefore does not require recompilation. The held-out-shape pushing experiments evaluate this form of reuse through goal images.

A change to the implementation instead starts a new compilation phase from the frozen artifact. *Local recompilation* restricts this revision to selected components while retaining compatible ones. For an appearance change, a replacement encoder whose outputs remain compatible with the retained state module gives

$$(\mathbf{Z}, E, P) \longrightarrow (\mathbf{Z}, E', P). \tag{5}$$

The state updater, reset rules, and control policy remain unchanged. By contrast, a new objective, a goal outside the compiled task family, or a newly required decision-relevant distinction may require changing $\mathbf{Z}$ and the control components that depend on it as well.

In our experiments, coding agents realize the compiler through execution-guided development; the corresponding rollout, revision, and validation procedures are detailed in Section 4 and the appendix.

Table 1: **The choice of coding agent matters for specialist quality.** All three Astra constructions exceed both visual RL baselines; independent starts also reveal variability within each compiler family. Results are on 100 test worlds; Reward is mean native return ± across-world sample SD, with Median return and benchmark Score (0–100) reported alongside.

(a) Specialists compiled by coding agents

Compiler	Version	Reward	Median	Score	Compile (h)	API ($)
Luna-1	v3	9.49 ± 2.03	10.10	15.75	1.29	0.74
Luna-2	v2	6.05 ± 2.12	6.10	7.02	1.66	1.24
Luna-3	v3	8.52 ± 2.47	9.10	15.63	1.14	0.74
Sol-1	v9	9.65 ± 2.96	10.10	22.10	1.17	6.43
Sol-2	v8	11.53 ± 3.05	12.10	24.78	1.91	14.20
Sol-3	v9	9.71 ± 2.22	10.10	18.25	1.46	13.30
Astra-1	v8	17.03 ± 2.68	18.10	68.42	0.87	18.93
Astra-2	v8	$\mathbf{19.45 \pm 1.92}$	**20.10**	**91.03**	1.58	31.06
Astra-3	v8	18.45 ± 2.64	19.10	84.71	0.98	23.87

(b) Visual reinforcement-learning baselines

Policy	Update	Reward	Median	Score	Train (GPU-h)	API ($)
CNN-PPO	1997	16.08 ± 2.97	16.10	55.21	≈ 48	—
Qwen3-VL-8B PPO	368	10.08 ± 2.02	10.10	17.30	≈ 144	—

4 EXPERIMENTS

We study task compilation through three research questions covering specialist construction, localized recompilation and generalization, and real-time deployment.

RQ1 (Craftax). *How do coding-agent choice and iterative construction affect specialist performance, and how efficiently do frozen specialists execute after compilation?*

RQ2 (Pushing). *Can state-centric interfaces support localized recompilation, and can changing only the task specification improve generalization to unseen object geometries?*

RQ3 (Minecraft). *Can compiled specialists complete tasks across diverse worlds in a real-time, open-world game under a fixed wall-clock budget?*

4.1 CRAFTAX: COMPILER CHOICE, ITERATIVE IMPROVEMENT, AND EFFICIENT EXECUTION

Craftax Classic (Matthews et al., 2024) is a procedurally generated survival and crafting game with 22 achievements spanning resource gathering, tool crafting, combat, and survival. Specialists receive 144×144 RGB images and choose among 17 native actions, with episode memory and a 10,000-step cap. *Reward* is native episode return: newly unlocked achievements plus 0.1 times the change in health, with a maximum of 22. *Benchmark Score* measures achievement coverage through a shifted geometric mean of success rates, ranging from 0 to 100. Reward governs version selection (Appendix B).

Compiler choice and specialist performance. Table 1 compares specialists compiled by GPT-5.6-Luna, GPT-5.6-Sol (OpenAI, 2026c), and GPT-6-Astra (OpenAI, 2026a), all at xhigh effort. Luna-1, Luna-2, and Luna-3 denote three independent compilation runs of GPT-5.6-Luna, with the same naming convention used for Sol and Astra. Each compilation starts from an empty workspace and the installed game, following a common prompt and prescribed development schedule. We select the final artifact on 20 validation worlds and evaluate it on 100 test worlds. The visual RL baselines are a CNN trained from scratch with PPO and Qwen3-VL-8B adapted with LoRA-PPO; both use single-frame observations, the same seed splits, and the same reward objective. We report compile time in wall-clock hours and training time in aggregate GPU-hours across eight H800 GPUs. Appendices B and B.3 detail construction and training.

All three Astra compilations exceed both visual RL baselines, while CNN-PPO outperforms the Luna and Sol artifacts. Across these independent runs, specialist performance varies substantially with the choice of coding agent. Astra's stronger specialists come with higher construction API costs. We next examine the development of compiled Astra-2, which achieves the highest validation Reward.

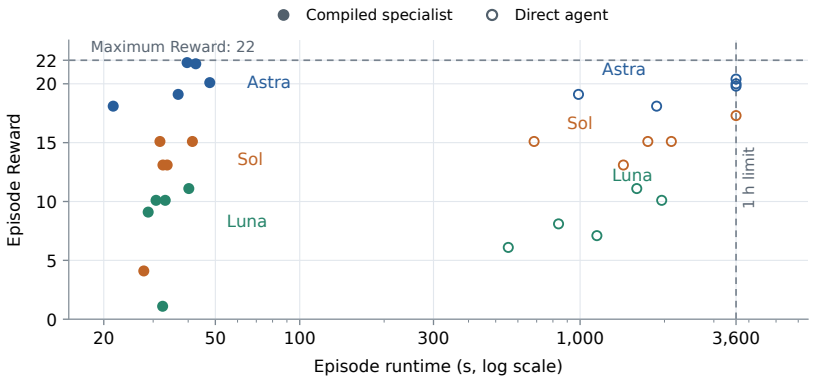


Figure 4: **Compiled execution is faster; retained capability varies.** Each point is one episode on one of five shared test worlds; compiled representatives are selected by validation Reward. The compiled Astra specialist attains similar return to direct control at much shorter runtime, while Sol trades return for speed; runtime covers the full execution loop on a log scale.

Iterative improvement. Figure 3 traces the development of compiled Astra-2. The initial v0 uses the game's public rules and artwork to combine texture matching and lighting compensation with a relative map and rule-based gathering, crafting, and survival. This first executable specialist already reaches 15.05 validation Reward. The largest early gains occur in v1 and v2, raising validation Reward to 18.45. Version v1 confirms drinking from the previous action and visible water even when the HUD value cannot increase, while v2 retrieves off-screen resources and colocates workstations. Later revisions address specific hazards through changes to perception, memory, and control. Version v5 separates arrow pixels from terrain and records their direction to predict impacts and dodge. Version v7 responds to deaths during sleep by maintaining a shelter phase that coordinates sealing an entrance, resting, and reopening it. Validation ultimately selects v8 at 19.85 and rejects v3, v4, and v9 (Appendix B.2). Together, these revisions show that compiled specialists can be iteratively improved through inspectable changes linking observed failures to visual evidence, maintained memory, and control decisions.

Execution efficiency after compilation. Direct agents are coding agents that play Craftax by inspecting each RGB observation and selecting the next action online. We compare direct play at xhigh effort with frozen specialists on five shared test seeds (3000–3004) under a one-hour evaluation horizon. For each compiler family, we select one of three independent compilations by highest mean Reward on the 20 validation worlds. Figure 4 shows substantially shorter episode runtimes across families. The compiled Astra specialist runs episodes approximately 70× faster on average, with a slightly higher mean Reward on these worlds. Runtime covers the complete execution loop (Appendix B.4). This comparison illustrates how task compilation can preserve strong control performance while enabling efficient local execution: the specialist reuses its code across worlds with zero online API use.

4.2 Pushing: Localized Recompilation and Generalization from Specifications

PushT (Chi et al., 2023) requires a pusher to translate and rotate a T-shaped object to match a goal image. Using current and goal RGB images, we test localized repair under appearance changes and generalization to unseen shapes (RQ2). We report SR95@300, the fraction of episodes covering at least 95% of the target area within 300 control steps, on 300 test seeds per condition or shape. Appendix C gives detailed protocols and all-run results.

Localized recompilation under visual shifts. Table 2a compares three branches from a common clean T specialist. Frozen applies it unchanged, E-only revises the visual encoder while retaining the state module, controller, and learned dynamics, and Full permits all components to change. E-only and Full share development data and validation opportunities within each perturbation family. Although severe noise, blur, dimming, and recoloring cause near-total failure for Frozen, E-only restores near-perfect success and closely matches Full. The state interface thus supports local repair of visual extraction while preserving the compiled control solution.

Table 2: **The state interface supports both local repair and frozen reuse.** Both panels report SR95@300 (%; 300 seeds per condition or shape). (a) E-only preserves state and control and nearly matches Full under visual shifts; the macro-average equally weights the four displayed perturbation families after averaging severities, excluding clean. (b) A generalization requirement improves T-only transfer but adds no gain in the multishape setting; each row is the validation-selected run among three independent compilations.

(a) PushT: visual shifts and localized recompilation

Method	Clean	Noise		Blur		Brightness	Object	Macro-average
		$\sigma = 20$	$\sigma = 30$	$\sigma = 1.5$	$\sigma = 3$	$\times 0.25$	color	(four families)
Frozen	99.67	0.33	0.00	87.33	2.00	0.00	0.00	11.21
E-only	—	**100.00**	**100.00**	**100.00**	99.67	**99.67**	**100.00**	99.88
Full	—	**100.00**	**100.00**	**100.00**	**100.00**	**99.67**	**100.00**	**99.92**

(b) PushObj: specification-guided shape generalization

Development	Specification	T	Held-out shapes S	V	C	A	Held-out macro SR
T	Standard	**100.00**	0.00	0.00	0.00	0.00	0.00
T	+ Generalization	99.33	83.67	86.00	22.00	83.00	68.67
T/L/Z/+	Standard	98.00	**89.00**	91.00	54.00	88.00	**80.50**
T/L/Z/+	+ Generalization	96.00	25.00	**97.67**	**95.00**	**96.67**	78.58

Generalization specified at construction. Table 2b crosses T-only versus T/L/Z/+ development with a standard specification versus one requiring generalization to unseen geometry. For each setting, validation SR on development shapes selects one of three independent compilations for frozen evaluation on S/V/C/A. Under T-only development, adding this requirement raises macro-averaged SR on held-out shapes from zero to 68.67% while retaining high T success. Multishape development improves transfer under either specification, with no further macro-average gain from the requirement. The across-run summaries show the same pattern. A generalization requirement can therefore improve transfer without adding development geometries, with the largest benefit under T-only development.

4.3 MINECRAFT: DEADLINE-CONSTRAINED DEPLOYMENT AND REUSE

Minecraft is a real-time, three-dimensional survival game. We test whether frozen specialists can craft a stone pickaxe through RGB screenshots and keyboard/mouse actions within five minutes across 30 worlds held out from development (RQ3). Table 3 compares specialists with and without the visible F3 pose/target overlay against direct Astra at xhigh and medium effort using ordinary RGB. SR measures completion within the deadline at each crafting milestone, and t averages completion time among successful episodes. Time per gesture covers the full interaction loop. Appendix D details the setup and evaluation.

Execution cost and timing. The F3-assisted specialist achieves the highest final-task success, while the F3-free variant matches direct Astra at medium effort in final-task success. Both execute at about 0.38 s per gesture, compared with 3.7–4.4 s for direct Astra across the full interaction loop. The same code and assets serve all 30 worlds with zero online API use, concentrating coding-agent computation in construction. Task compilation therefore combines task completion in unseen worlds with frequent local interaction under the deadline and reuse across deployments.

5 RELATED WORK

Coding agents as task compilers. Code as Policies, Voyager, and WorldCoder construct reusable policy programs, skills, and executable world models (Liang et al., 2022; Wang et al., 2023; Tang et al., 2024). This area is developing rapidly, with several concurrent efforts exploring related construction-to-execution patterns. RHO uses execution feedback to construct a policy repository that is frozen for CaP-Bench deployment (Elmaaroufi et al., 2026). GaP produces a typed graph policy with locally revisable subgraphs, agentic Push-T produces a state-based visual controller, and CaP-X studies how supplied primitives and interfaces affect construction (Chen et al., 2026; Xie

Table 3: **Local execution supports closed-loop control within a fixed time budget.** On 30 shared held-out worlds, frozen specialists interact much faster than direct agents, while different completion rates separate execution speed from task competence. F3 supplies visible pose/target text; SR is success within 300 s, t is mean visual-confirmation time among successes, and time per gesture covers the full interaction loop.

			Crafting table		Wooden pickaxe		Stone pickaxe		
Policy	n	Reuse[‡]	SR↑	t ↓	SR↑	t ↓	SR↑	t ↓	s/gesture↓
SCTC (with F3)	30	$O(1)$	<u>83.3</u>	81.6	**70.0**	96.3	**60.0**	174.1	**0.374**
SCTC (w/o F3)	30	$O(1)$	66.7	118.3	<u>66.7</u>	135.5	<u>30.0</u>	207.1	0.382
Astra (xhigh)	30	$O(n)$	66.7	187.4	36.7	203.5	13.3	244.2	4.368
Astra (medium)	30	$O(n)$	**96.7**	171.8	**70.0**	194.0	<u>30.0</u>	269.3	3.655

[‡] Reuse counts coding-agent computation across n worlds under a fixed specification and per-episode budget: $O(1)$ one-time compilation or $O(n)$ repeated online reasoning.

et al., 2026; Fu et al., 2026). By contrast, RACaP illustrates another runtime boundary: policy APIs are frozen, but language-model decisions remain online (Li et al., 2026). Practitioner commentary likewise frames coding agents as compiler-like systems, reflecting broader interest in moving general-purpose construction out of the repeated execution loop (Dash, 2026).

State interfaces, reuse, and localized recompilation. VoxPoser and ReKep introduce structured representations between perception and control, while self-supervised policy adaptation changes visual features during deployment while retaining the action head (Huang et al., 2023; 2024; Hansen et al., 2021). SCTC instead uses the task-state and memory module $\mathbf{Z}$ to define the compatibility boundary between visual evidence and control. Concretely, if a replacement encoder E' supplies evidence expected by retained $\mathbf{Z}$ and P, only E needs to be recompiled; newly required decision-relevant information may require revising $\mathbf{Z}$ and the dependent control components. Our pushing studies test this pre-deployment repair boundary and, separately, frozen reuse on unseen geometry (Section 4.2).

Construction and execution mechanisms. FunSearch and AlphaEvolve exemplify search procedures for constructing executable programs (Romera-Paredes et al., 2024; Novikov et al., 2025); Eureka and Text2Reward instead generate rewards used to train learned policies (Ma et al., 2024; Xie et al., 2024). These approaches offer different construction procedures; at deployment, direct visuomotor policies and predictive models are possible specialist implementations (Chi et al., 2023; Kim et al., 2024; Black et al., 2024; Zhou et al., 2025), while asynchronous action-chunk execution changes how a finished policy runs (Black et al., 2025).

Taken together, prior work and emerging practice point to a common construction–artifact–execution pattern. We formalize this pattern as *task compilation* (Section 2) and develop SCTC to make state-mediated reuse and localized recompilation boundaries explicit.

6 Limitations and Conclusion

Compilation trades generality for a cheaper runtime. A frozen specialist is limited to the visual distinctions, maintained state, control procedures, and learned components encoded in its artifact; it does not automatically inherit the builder's broader visual and reasoning capabilities. Local recompilation also depends on retained interfaces continuing to supply the information needed by unchanged components. Qualitatively new objectives, observations, or decision-relevant distinctions may therefore require joint revision of multiple components rather than a local replacement. Our evaluation is currently limited to simulated manipulation and game environments. Extending task compilation to physical systems, where perception, dynamics, and execution are less controlled, is an important direction for future work.

In summary, task compilation enables coding agents to build visual control specialists that execute independently. SCTC organizes their construction around task state, visual evidence extraction, and control, with the state interface supporting reuse and localized recompilation. General-purpose computation can thus be concentrated in development and its cost amortized across repeated task-specific executions.

AI USE STATEMENT

The authors developed the conceptual framework and experimental design. Coding agents generated the experimental code, including the specialist implementations and evaluation and analysis scripts. The models and inference settings used in the experiments are reported in Section 4 and Appendices B–D. Generative AI also assisted data organization, numerical analysis, literature search, figure preparation, and manuscript development. The authors and AI tools jointly produced the manuscript, with the authors participating in result interpretation and the formulation of conclusions. The authors reviewed the AI-assisted content and take responsibility for the final text, claims, code, and experimental results.

REPRODUCIBILITY STATEMENT

Appendix A presents the construction-prompt template. Appendices B, C, and D document task settings, evaluation splits, metrics, selection procedures, baseline configurations, and detailed results. The supplementary archive `iclr2027_supplementary_code.zip` provides selected frozen specialists for Craftax, PushT/PushObj, and Minecraft, together with task-specific evaluation code, dependency specifications, reference results, and reproduction instructions. Its task READMEs specify artifact coverage and required external assets. The accompanying paper source package includes numerical summaries and scripts for regenerating the tables and figures.

ETHICS STATEMENT

This study evaluates visual control in simulated manipulation tasks and game environments. All evaluation observations and trajectories are generated within these environments, without human-subject experiments or personal-data collection. Third-party software, pretrained models, and game assets remain subject to their respective licenses. Applications beyond the evaluated settings require assessment of task-specific failure modes and appropriate human oversight.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, et al. Qwen3-VL technical report, 2025. URL https://arxiv.org/abs/2511.21631.

Kevin Black, Noah Brown, Danny Driess, et al. π_0: A vision-language-action flow model for general robot control, 2024. URL https://arxiv.org/abs/2410.24164.

Kevin Black, Manuel Y. Galliker, and Sergey Levine. Real-time execution of action chunking flow policies, 2025. URL https://arxiv.org/abs/2506.07339.

Kaiyuan Chen, Shuangyu Xie, Letian Fu, et al. GaP: A graph-as-policy multi-agent self-learning harness for variational automation tasks, 2026. URL https://arxiv.org/abs/2607.05369.

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, 2023. URL https://arxiv.org/abs/2303.04137.

Anil Dash. Coding agents as the new compilers. https://www.anildash.com/2026/02/11/coding-agents-as-the-new-compilers/, February 2026. Blog post; published February 12, 2026; accessed September 25, 2026.

Karim Elmaaroufi, Justin Svegliato, Sarunas Kalade, Graham Schelle, Sanjit A. Seshia, and Matei Zaharia. RHO: Your coding agent is secretly a roboticist, 2026. URL https://arxiv.org/abs/2606.16458.

Letian Fu, Justin Yu, Karim El-Refai, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation, 2026. URL https://arxiv.org/abs/2603.22435.

Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999. doi: 10.1023/A:1010095604496.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2):1–119, 2017. doi: 10.1561/2500000010.

Nicklas Hansen, Rishabh Jangir, Yu Sun, Guillem Alenyà, Pieter Abbeel, Alexei A. Efros, Lerrel Pinto, and Xiaolong Wang. Self-supervised policy adaptation during deployment. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2007.04309.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

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, 2023. URL https://arxiv.org/abs/2307.05973.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. ReKep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation, 2024. URL https://arxiv.org/abs/2409.01652.

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, 2024. URL https://arxiv.org/abs/2406.09246.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814.

Zexi Li, Yehang Zhang, Haojian Huang, Bohan Zhou, Wenqian Li, Chenxu Wang, Yifan Chang, Yangkai Wei, Tianyi Zhang, Ying-Cong Chen, Kaiwen Zhou, Yinchuan Li, and James Cheng. RACaP: Agentic reasoning, acting, and coding as policies for evolvable robot learning, 2026. URL https://arxiv.org/abs/2609.29394v1. Version 1, September 24, 2026.

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, 2022. URL https://arxiv.org/abs/2209.07753.

LLVM Project. *LLVM Language Reference Manual*, 2026a. URL https://llvm.org/docs/LangRef.html. Online documentation, accessed September 25, 2026.

LLVM Project. *Using the New Pass Manager*, 2026b. URL https://llvm.org/docs/NewPassManager.html. Online documentation, accessed September 25, 2026.

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 *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.12931v2.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning, 2024. URL https://arxiv.org/abs/2402.16801.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131v1. White paper; version 1.

OpenAI. GPT-6 Astra System Card, September 2026a. URL https://deploymentsafety.openai.com/gpt-6-astra. Published September 3, 2026.

OpenAI. Computer use, 2026b. URL https://developers.openai.com/api/docs/guides/tools-computer-use. API documentation. Accessed September 26, 2026.

OpenAI. GPT-5.6 System Card, July 2026c. URL https://deploymentsafety.openai.com/gpt-5-6. Published July 9, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015. URL https://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Oriane Siméoni, Huy V. Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, et al. DINOv3, 2025. URL https://arxiv.org/abs/2508.10104.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment, 2024. URL https://arxiv.org/abs/2402.12275.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

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 *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Shuangyu Xie, Kaiyuan Chen, and Ken Goldberg. Revisiting the "Push-T" robot manipulation task with agentic robotics, 2026. URL https://arxiv.org/abs/2608.18227.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2Reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2309.11489v3.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Gaoyue Zhou, Hengkai Pan, Yann LeCun, and Lerrel Pinto. DINO-WM: World models on pre-trained visual features enable zero-shot planning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 79115–79135. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhou25t.html.

A CONSTRUCTION SETTINGS AND PROMPT

Coding agents receive a task specification, permitted resources and runtime interfaces, development and validation tasks, and selection rules. The Craftax comparison uses three independent starts each of GPT-5.6-Luna, GPT-5.6-Sol, and GPT-6-Astra, all at xhigh effort; the pushing study uses GPT-6-Astra (xhigh). Construction produces frozen code and assets, with episode memory initialized anew at deployment. Task-specific protocols appear in Appendices B, C, and D.

A.1 CONSTRUCTION-PROMPT TEMPLATE

The following template summarizes the construction requests, with braces marking task-specific values. It uses the paper's notation: **Z** maintains state and memory, E extracts current visual evidence, and P plans and acts. Archived prompts that place state updating inside P assign the same responsibility to **Z** here. The instantiated prompts and subsequent interventions belong to the construction record.

```
You are {model}, with reasoning effort {effort}.
Construct a visual control specialist for {task} using
{initial_workspace_and_permitted_resources}.

Build E: visual evidence extraction; Z: state, memory, recursive
updates and reset; P: planning and control. Use {runtime_inputs}
and {action_interface}. Reset episode memory between episodes.
Deploy frozen code and assets independently of the coding agent.
Keep evaluator-only information outside the runtime interface.

Fix {environment}, {dev/validation/test_splits}, {metrics},
{episode_limit}, {construction_budget}, and {selection_rule}.
Build a runnable initial specialist. On each development batch,
inspect RGB, actions and maintained state; identify a failure,
revise the relevant components, and compare parent and candidate.
Record the actual parent, component changes, evidence, results,
and resource use. Check compatibility of retained components.
Submit candidates under {validation_protocol}; retain rejected
versions and restore the complete incumbent after rejection.

At {stopping_rule}, freeze the validation-selected artifact and
run {test_protocol}. Save code/assets, hashes, exact prompts,
human interventions, per-episode results, costs, and commands.
```

Implementation records. Each revision records the representation, its update/reset behavior, and the components that produce or consume it. Visual assets and any construction-only supervision are recorded separately from runtime inputs. The supplied Astra-2 archive contains candidate code, source diffs, and evaluation records; the other experiments currently supply different subsets of these records.

B CRAFTAX: CONSTRUCTION AND EXECUTION

B.1 ENVIRONMENT, METRICS, AND COMPILATION PROTOCOL

The nine runs in Table 1 comprise three independent starts each of GPT-5.6-Luna, GPT-5.6-Sol, and GPT-6-Astra at xhigh effort. Each starts from an empty workspace and the installed game. Craftax Classic 1.6.1 uses 64×64 maps, 17 native actions, and `make_craftax_pixel_renderer(16)`, producing 144×144 RGB including the HUD. Episodes end at death or 10,000 steps, with auto-reset disabled. Specialists receive RGB, previous actions, and their own episode state; rewards, seed identities, and native state remain evaluator-side.

Native Reward is $G = \sum_t [(C_{t+1} - C_t) + 0.1(H_{t+1} - H_t)]$, where C_t counts achievements and H_t is health. With initial and maximum health nine, Reward is bounded above by 22. Benchmark Score aggregates the success rates of all 22 achievements, including those never attained:

$$p_j = \frac{100}{n} \sum_{i=1}^{n} \mathbf{1}[\text{episode } i \text{ attains } j], \qquad S = \exp\left(\frac{1}{22} \sum_{j=1}^{22} \log(1 + p_j) \right) - 1. \tag{6}$$

Here p_j is a percentage. A terminal episode score instead applies $\exp(k \log 101/22) - 1$ to that episode's achievement count k. All reported $\pm$ values use sample SD across worlds.

Development and selection. Development uses seeds 1000–1099, validation 2000–2019, and test 3000–3099. Ten five-seed development batches are fixed before construction:

```
rng = np.random.Generator(np.random.PCG64(42))
dev_schedule = [rng.choice(np.arange(1000, 1100), 5,
                replace=False).tolist() for _ in range(10)]
```

Batches may overlap. A runnable v0 establishes the validation incumbent. Each subsequent version compares its accepted parent and candidate on the current batch, inspecting RGB trajectories, actions, and maintained state. Failed trials retry that batch. The first strict development-return improvement triggers one 20-world validation submission; only a strict improvement in unrounded validation return replaces the complete incumbent. Validation exposes aggregate return and acceptance. Construction allows up to ten submissions, continuing through plateaus within the resource budget. The selected artifact is frozen before the 100-world test. Table 1 reports test Reward, median, Score, and construction time and API cost.

Recorded protocol differences. The supplied cross-run audit records different random-key derivations: Astra-2, Luna-2, and Sol-1 use `PRNGKey(seed)` with one `fold_in` per step; Astra-1 uses nested folding; Luna-1/3 and Sol-2 split repeatedly; Sol-3 splits before reset and Astra-3 folds before reset. Thus some realized worlds or transitions differ despite shared seed lists. Luna-1 inspects per-seed validation trajectories, completes some gates after testing, and evaluates only 18 worlds for rejected v2. Luna-3 uses an extra development seed and repeats v3 validation; reported statistics use 20 unique validation seeds. Luna-2 restores an accidentally modified selected artifact, reproducing its original 100 test returns, lengths, and achievements. These implementation histories accompany the artifact comparison.

B.2 ASTRA-2: ITERATIVE IMPROVEMENT

Astra-2 uses GPT-6-Astra (xhigh), `codex-cli` 0.155.1, and the game's public rules and artwork. Its evaluator sets `day_length=300` and `always_diamond=true`; reset uses `PRNGKey(seed)`, and step t uses `fold_in(PRNGKey(seed),t+1)`. CPU evaluation uses five-episode batches and eight cores per worker, with four workers for the final test. Table 4 gives all ten submissions; Figure 3 illustrates their implemented changes.

Initial state and early gains (v0–v2). V0 matches sampled tile pixels to public textures, compensates for lighting, reads HUD digits, and maintains relative position, terrain, visits, and interaction flags. Shortest paths and crafting priorities yield 15.05 validation Reward. V1 addresses repeated drinking at a saturated HUD: the previous action and visible water confirm completion even when

Table 4: Astra-2 submitted versions. Each development comparison uses the current five-seed batch; validation always uses seeds 2000–2019. The parent is the retained artifact, so v4 and v5 both branch from v2. Return selects versions; Score is reported separately.

Version	Parent	Parent dev	Candidate dev	Val return	Val Score	Decision
v0	—	—	13.30	15.05 ± 3.03	48.96	Accept
v1	v0	11.70	14.70	17.05 ± 2.61	58.35	Accept
v2	v1	17.30	17.70	18.45 ± 1.50	73.35	Accept
v3	v2	18.30	19.10	18.25 ± 2.32	78.83	Reject
v4	v2	17.50	17.50	17.80 ± 2.58	78.24	Reject
v5	v2	18.70	18.90	19.05 ± 1.15	82.18	Accept
v6	v5	16.70	16.90	19.35 ± 1.48	89.11	Accept
v7	v6	19.10	20.30	19.70 ± 1.31	92.21	Accept
v8	v7	18.70	19.10	19.85 ± 1.33	93.23	Accept; selected
v9	v8	17.50	18.70	19.70 ± 1.50	92.31	Reject

the displayed value cannot increase. Heading extraction, mining access, and rest timing also change, raising validation Reward to 17.05. V2 retrieves remembered off-screen water and minerals, retains a refill intention, and colocates the furnace with a usable crafting table; validation reaches 18.45.

Visual evidence and recursive memory (v5–v6). V5 detects directional arrows using opaque sprite pixels and masks them during terrain recognition. The resulting projectile grid supports next-cell prediction and dodging; revised weapon preparation accompanies the gain to 19.05. In v6, plant locations and ages alone fail to support reliable harvests: nighttime visual registration has shifted the map, making a remembered table appear as water and invalidating a crop after 85 steps. Replacing whole-window registration with displacement inferred from the previous neighboring cells, collision evidence, and action stabilizes these coordinates. The same crop memory can then guide revisits, and validation reaches 19.35.

Coordinated recovery and reuse (v7–v8). V7 responds to deaths during sleep by maintaining shelter coordinates, preparation/rest/exit phase, timestamps, and timeouts. Control excavates a dead end, seals its entrance, rests, and reopens it, with combat or projectile avoidance able to interrupt preparation. Validation rises to 19.70. V8 addresses minerals blocked by water: when inventory contains stone and a wooden pickaxe, planning admits water edges at cost four and control executes place–mine–cross. It reuses the map, inventory, and encoder, reaching 19.85 and selecting v8.

Rejected revisions and retained evidence. V3's exploration changes, v4's crop and combat changes, and v9's blockage-aware dodging pass development but reduce validation return, so each restores its parent. V4's submission is triggered by an unrounded development gain of 1.49×10^{-9} despite a rounded tie. Four additional trials stop before validation: two v2 candidates return 17.10/15.70 against parent 17.30, and two v6 candidates return 16.70/13.70 against parent 16.70. The complete archive contains 125 development, 200 validation, and 100 test episodes, with five pre-reset worker failures recorded separately. A lane-order check verifies an evaluator optimization; affected v7 debug traces are reconstructed from saved RGB and code because the originals retained mutable dictionaries by reference. The frozen v8 obtains 19.45 ± 1.92 test Reward and 91.03 Score. These are outcomes of complete revisions that can change multiple components; the records expose how evidence, memory maintenance, and decisions evolve together.

B.3 Visual reinforcement-learning baselines

CNN-PPO and Qwen3-VL-8B PPO share the compilation study's task, seed splits, native reward, and 10,000-step cap. Each trains once on eight H800 GPUs using a single current frame and categorical action sampling; stateful compiled specialists additionally retain episode history. Training cycles through all 100 development worlds. Validation occurs approximately hourly, selects by strictly higher mean Reward, and stops after two consecutive non-improvements. The initial model is eligible. Evaluation uses one episode per world with action RNG `numpy.default_rng(world_seed+3000000)`.

CNN-PPO. The randomly initialized encoder normalizes 144×144 RGB to $[0, 1]$ and applies three ReLU convolutions: channels 32/64/64, kernels 8/4/3, strides 4/2/1. Flattened features feed a 512-unit ReLU layer and separate 17-action and value heads. All 6,508,210 parameters train. Orthogonal initialization uses gains 0.01 for the actor, 1 for the critic, and $\sqrt{2}$ elsewhere. Update 1997 is selected at 16.65 validation Reward; subsequent values 16.55 and 15.70 stop the run.

Qwen3-VL-8B PPO. Qwen3-VL-8B-Instruct (Bai et al., 2025) starts from checkpoint revision `0c351dd01ed87e9c1b53cbc748cba10e6187ff3b`. Its backbone, visual encoder, and action projection are frozen; new rank-16 LoRA adapters train the query/value projections in all 36 language-attention layers (Hu et al., 2021). A zero-initialized value head brings trainable parameters to 7,671,809. The input is the current RGB resized to 288×288 by nearest neighbor plus a fixed task/action prompt. Frozen language-head rows A–Q project the final hidden vector into 17 action logits, sampled at temperature 3. Update 368 is selected at 10.80 validation Reward; updates 391/414 yield 10.00/10.40 and stop training.

Table 5: Visual PPO configurations. Both use eight H800s, 32 environments per GPU, 16,384 transitions per update, constant learning rates, and optimizer $\beta = (0.9, 0.999)$, $\epsilon = 10^{-5}$. PyTorch 2.6.0 and CPU JAX 0.4.38 provide training and simulation; Qwen uses Transformers 4.57.6 and PEFT 0.18.1.

Setting	CNN-PPO	Qwen3-VL-8B PPO
Input / sampling temperature	144×144 RGB / 1	288×288 RGB + prompt / 3
Trainable parameters	6,508,210	7,671,809
Environments / steps per rollout	256 / 64	256 / 64
PPO epochs / global minibatch	up to 4 / 1024	2 / 1024
Microbatch per GPU / accumulation	128 / 1	16 / 8
Optimizer / weight decay	Adam / 0	AdamW / 0.01
Learning rate	2×10^{-4}	2×10^{-5} LoRA; 10^{-4} value
Discount / GAE parameter	0.99 / 0.8	0.99 / 0.95
Policy / value clipping	0.2 / 0.2	0.2 / 0.2
Entropy / value coefficient	0.01 / 0.5	0.02 / 0.5
Precision	FP32; TF32 enabled	BF16; FP32 value head
LoRA rank / alpha / dropout	—	16 / 32 / 0
Selected / final update	1997 / 3028	368 / 414
Selected / final transitions	32,718,848 / 49,610,752	6,029,312 / 6,782,976
Selected / final active GPU-hours	32.05 / 48.06	129.06 / 145.17

Optimization and accounting. Both use PPO with GAE (Schulman et al., 2017; 2015), global advantage normalization, clipped policy/value losses, gradient norm 1, and a 0.06 approximate-KL cutoff within each rollout. The value loss has internal factor $1/2$ and external weight 0.5. Death and the time limit use zero bootstrap; other rollout boundaries bootstrap from the critic. Program seed 20260924 and rank-specific sampling streams separate world selection, action sampling, and minibatch shuffling. CNN's KL cutoff triggers in 2843 of 3028 updates; Qwen's never triggers. Active time includes rollout, optimization, CPU simulation, and synchronization, with validation, testing, loading, and saving separate. Full runs take 6.0075 and 18.1459 hours on eight H800s, or 48.06 and 145.17 GPU-hours; Table 1 uses rounded 48 and 144. Table 6 consolidates test results and all achievement rates. Exact configurations and validation trajectories remain in the archived baseline reports.

B.4 Execution efficiency after compilation

Three direct xhigh agents and all nine validation-selected specialists each run once on seeds 3000–3004, yielding 60 episodes. Ranking the three independent compilations in each family by their recorded mean Reward on validation seeds 2000–2019 selects Astra-2 (19.85), Sol-2 (12.15), and Luna-1 (10.45) for Figure 4; Table 7 retains every artifact. This cross-run ranking uses archived validation summaries; versions within runs were selected before the five-seed evaluation. SD measures variation across five worlds; each direct model contributes one five-episode set.

Execution and stopping. The environment and images match the compilation study. Reset uses `PRNGKey(seed)` and transition t uses `fold_in(PRNGKey(seed),t+1)`. CPU execution uses an x86-64 host with two Intel Xeon Platinum 8458P processors (44 physical cores per socket,

Table 6: Visual RL results on the 100 shared test worlds. CNN initialization is random; Qwen initialization is pretrained with LoRA disabled. The trained checkpoints are CNN update 1997 and Qwen update 368. Reward reports mean ± sample SD; Score aggregates all 22 achievement rates.

Metric	CNN init.	CNN-PPO	Qwen init.	Qwen-PPO
Reward	1.40 ± 1.31	16.08 ± 2.97	0.76 ± 1.07	10.08 ± 2.03
Benchmark Score	1.7485	55.2098	1.3606	17.3002
Mean episode length	140.80	390.91	159.88	234.40
Achievement SR (%)				
collect wood	46	100	53	100
place table	9	100	1	96
eat cow	1	87	4	70
collect sapling	37	74	69	97
collect drink	10	82	18	42
make wood pickaxe	1	100	0	93
make wood sword	1	97	0	67
place plant	29	72	1	92
defeat zombie	0	96	7	83
collect stone	0	96	0	79
place stone	0	92	0	67
eat plant	0	0	0	0
defeat skeleton	0	73	0	21
make stone pickaxe	0	93	0	0
make stone sword	0	94	0	1
wake up	96	89	13	92
place furnace	0	85	0	75
collect coal	0	90	0	23
collect iron	0	77	0	0
collect diamond	0	0	0	0
make iron pickaxe	0	36	0	0
make iron sword	0	65	0	0

176 logical CPUs in total). The environment is allocated eight cores, with OMP/OpenBLAS restricted to one thread each. Direct agents play in five concurrent independent sessions, receiving RGB, their action history, observation index, and termination flag, and choosing one native action per image. Compiled artifacts run sequentially in separate processes, each with five synchronous episodes and fresh episode memory. The game advances only when an action is submitted.

All episodes stop at death or 10,000 steps; none reaches the step cap. Compiled runs additionally set one-hour and all-22-achievement limits in advance. Direct Astra receives the one-hour deadline before play; achievement stopping is added before any full completion. Sol's timeout is assigned during post-run scoring of its recorded trajectory. Direct Luna's five episodes all die within one hour under its original stopping protocol. Ten earlier Astra-2/Luna-2 repeats, which use different stopping rules, remain separate in the source records.

Timing and API cost. Episode runtime is elapsed wall-clock time from initial RGB to termination or the one-hour evaluation boundary and includes reasoning, service/tool latency, environment stepping, and waiting. The first environment-step JIT is included; initialization is separate. These durations measure the complete execution arrangements and depend on episode outcome. Compiled initialization takes 6.28–13.04 seconds; mean cumulative policy-call time is 1.25–6.51 seconds per episode across artifacts. The report retains batch and process timings separately.

API estimates use the experiment's saved rates per million uncached-input/cached-input/output tokens: Astra \$10/\$1/\$50, Sol \$4/\$0.40/\$20, Luna \$0.20/\$0.02/\$1.20. Images count as input; output includes reasoning, and context compaction is included. Sol's \$33.910178 estimate includes requests logged within the one-hour evaluation window. Luna's pre-game tool-startup failure adds \$0.040402 outside the evaluation; valid-session errors remain included. Compiled execution uses zero online API tokens. Construction costs in Table 1 and local compute are separate from these deployment charges.

Table 7: All current five-seed evaluations. Reward and episode runtime report mean ± sample SD; Score is the aggregate achievement score. D/T/F denotes death, one-hour timeout, and all-achievement completion. API cost is the five-episode total (USD); † marks Sol's charge for requests logged within the evaluation window.

Method	Reward	Score	Runtime (s)	D/T/F	API USD
Astra (direct)	19.48 ± 0.90	76.66	2732.38 ± 1228.82	2/3/0	80.8039
Sol (direct)	15.14 ± 1.49	43.02	1914.54 ± 1079.21	4/1/0	33.9102†
Luna (direct)	8.50 ± 2.07	16.23	1217.61 ± 564.18	5/0/0	0.9829
Astra-1 (v8)	17.30 ± 1.64	57.46	28.29 ± 5.94	5/0/0	0
Astra-2 (v8)	20.16 ± 1.61	93.56	37.69 ± 9.86	3/0/2	0
Astra-3 (v8)	16.90 ± 3.96	74.81	34.51 ± 15.87	5/0/0	0
Sol-1 (v9)	9.90 ± 2.28	16.08	27.91 ± 1.81	5/0/0	0
Sol-2 (v8)	12.10 ± 4.58	26.38	33.41 ± 4.99	5/0/0	0
Sol-3 (v9)	10.30 ± 1.92	16.40	40.43 ± 10.92	5/0/0	0
Luna-1 (v3)	8.30 ± 4.09	11.14	33.06 ± 4.34	5/0/0	0
Luna-2 (v2)	6.50 ± 1.14	5.01	20.55 ± 1.64	5/0/0	0
Luna-3 (v3)	8.10 ± 2.00	10.67	22.81 ± 1.42	5/0/0	0

C PUSHING: LOCALIZED RECOMPILATION AND SHAPE GENERALIZATION

C.1 ENVIRONMENT AND CONSTRUCTION PROTOCOL

Task and runtime. We use `gym-pusht` 0.1.6, Gymnasium 1.3.0, Pymunk 6.11.1, Pygame 2.6.1, OpenCV 4.10.0.84, NumPy 1.26.4, and Python 3.10.12. Observations are 224×224 RGB; actions specify an absolute pusher target in $[0, 512]^2$ at 10 Hz, with ten 0.01 s physics substeps per control step. The goal image renders the object at $[256, 256, \pi/4]$ with the pusher hidden. Specialists receive current/goal RGB and retain their own action history and state; code, parameters, and configuration are frozen at test, with memory reset between episodes. Ground-truth state, rewards, seeds, shape assets, and simulator interfaces remain evaluator-side.

Splits and metrics. Development uses seeds 1000–1199, validation 2000–2099, and testing 3000–3299. Methods share saved initial states for the same shape, with separate physics, perturbation, and policy RNG streams. Coverage is object–target intersection area divided by target area. SR95@300 counts episodes reaching coverage ≥ 0.95 at initialization or within 300 native control steps, including action-chunk steps. Episodes stop at first success; policy errors remain failures. Mean first-success steps charge all failures 300 steps. The study includes 300 clean, 9900 perturbed, and 28,800 shape episodes.

Construction and selection. GPT-6-Astra (xhigh) builds each specialist in a fresh context and directory with prompt-level resource restrictions. The encoder uses segmentation, connected components, moments, and contour fitting or registration. Control uses dynamics fitted from self-collected RGB/action development interactions, stored as JSON/NPZ assets. Visual repairs can fit fixed calibration parameters from clean-encoder pseudo-labels. This protocol supplies development interaction rather than expert demonstrations or pretrained visual backbones; prediction uses the fitted dynamics rather than the physics engine. Validation ranks candidates by SR, then lower failure-charged steps, then earlier submission, equally weighting severities or development shapes. The operative cap is three submissions; A1 had completed five before the cap changed from ten. No common numerical compute ceiling is imposed.

C.2 VISUAL PERTURBATIONS AND LOCAL RECOMPILATION

The common clean specialist passes validation at 100/100 and frozen testing at 299/300 (72.08 mean steps, zero policy errors). Its test result also serves as the specified 98% admission gate for subsequent construction. Each perturbation family independently branches from this artifact.

Perturbations. Each condition modifies both current and goal RGB. Gaussian noise uses $\sigma = 10, 20, 30$ in 0–255 units, independently per pixel/channel and between current/goal images. Current noise changes each step; goal noise is sampled once per episode. Gaussian blur uses $\sigma = 0.5, 1.5, 3$ pixels and kernel width $2\lceil 3\sigma \rceil + 1$. Brightness multiplies RGB by 0.25, 0.5, or 1.5 before clipping

and quantization. Background recoloring maps $(255, 255, 255)$ to $(48, 48, 48)$; object recoloring maps $(119, 136, 153)$ to $(220, 60, 60)$. Table 2a displays six conditions from four families, with brightness 0.25; its macro SR equally weights displayed families after averaging their severities, excluding clean. Table 8 includes all eleven conditions.

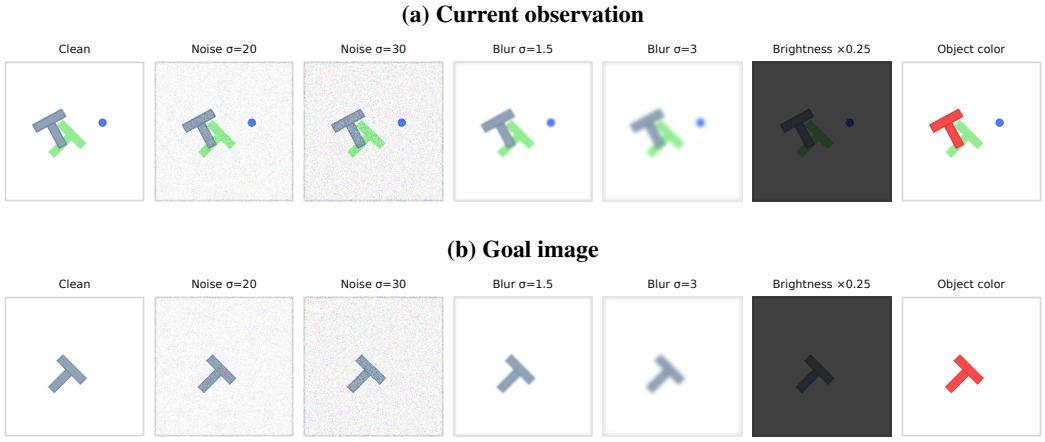


Figure 5: **PushT visual perturbations.** Current and goal RGB under clean inputs and representative perturbations from Table 2a. Noise, blur, brightness, and object recoloring affect both inputs; noise is sampled independently. The goal image omits the pusher.

Revision boundary. E-only changes the encoder and its exclusive assets, retaining the complete state module, controller, dynamics weights, initialization, downstream configuration, and shared dependencies. Its current-image outputs preserve meanings, units, coordinates, and missing-value conventions; cross-step memory remains in $\mathbf{Z}$. Full permits all components to change. Each pair shares its starting artifact, development tasks, synchronized RGB/action data, feedback, and validation opportunities. The report records frozen-file, interface, repeat-call, and call-order checks plus shared-data hashes. Each original family/scope branch yields one artifact covering all its severities. Table 9 records the realized Full edits and validation results for the original ten conditions.

Table 8: PushT success counts and mean steps. A failed episode is charged 300 steps. Each condition uses 300 seeds. Brightness $\times 0.25$ counts are reconstructed from the reported SR and this denominator; its mean-step statistics were not supplied (—).

	Successes / 300			Mean steps, failures = 300		
Condition	Frozen	E-only	Full	Frozen	E-only	Full
Clean starting specialist	299/300	—	—	72.08	—	—
Noise $\sigma = 10$	300/300	300/300	300/300	72.12	70.30	66.84
Noise $\sigma = 20$	1/300	300/300	300/300	299.90	70.72	66.35
Noise $\sigma = 30$	0/300	300/300	300/300	300.00	70.59	66.28
Blur $\sigma = 0.5$	299/300	300/300	298/300	71.51	70.80	72.12
Blur $\sigma = 1.5$	262/300	300/300	300/300	103.61	70.45	69.87
Blur $\sigma = 3$	6/300	299/300	300/300	296.83	71.65	69.76
Brightness $\times 0.25$	0/300	299/300	299/300	—	—	—
Brightness $\times 0.5$	298/300	299/300	299/300	72.55	72.11	72.05
Brightness $\times 1.5$	300/300	300/300	300/300	69.69	69.72	69.54
Background color	299/300	299/300	299/300	71.90	72.13	71.90
Object color	0/300	300/300	300/300	300.00	71.72	70.87

Across all eleven conditions, the five-family macro SR is 52.51% for Frozen, 99.87% for E-only, and 99.84% for Full. Brightness 0.25 collapses Frozen to zero while both repairs reach 99.67%; milder brightness and background recoloring largely preserve the original specialist. Full's noise branch also changes control and reduces mean steps to 66.28–66.84 versus E-only's 70.30–70.72. The results locate the repair in current-image extraction for these shifts, while recording when Full also changes downstream behavior. The brightness-0.25 update supplies SR but not step statistics.

Table 9: Recorded validation history and actual Full edits for the original ten-condition evaluation. Each branch makes one formal validation submission on its 100-seed pool. Brightness here uses multipliers 0.5 and 1.5. Full denotes permitted scope; the realized edit can be smaller.

Family	Episodes	E-only SR	Full SR	Actual Full edit
Noise	300	100.00	100.00	E, P control settings, associated assets/documentation
Blur	300	99.67	100.00	E only
Brightness	200	100.00	100.00	E, fixed visual parameters, documentation
Background color	100	100.00	100.00	Original specialist retained
Object color	100	100.00	100.00	E and associated documentation

C.3 Specification and development-shape diversity

The four groups cross development shapes with one added specification clause: *"At deployment, the specialist should handle object geometries unseen during development."* A/B use T-only development, with standard/generalization specifications; C/D use T/L/Z/+ with the same two specifications. Each group has three independent constructions and the same 200 development/100 validation base tasks. C/D divide these equally across four shapes; A/B generate only T development samples. Validation uses development shapes. Every artifact is tested on T/L/Z/+/S/V/C/A, 300 seeds per shape; S/V/C/A are held out for all groups.

T retains its native geometry. Other glyphs use DejaVu Sans contours at height 140 pixels, simplified with 2-pixel tolerance and centered at the area centroid, preserving holes and concavities. The report checks that collision, rendering, and scoring use consistent geometry. Goal images supply the target contour to the frozen runtime.

Table 10: Shape results: mean $\pm$ sample SD over three independent constructions. SR is in percent; unsuccessful episodes count as 300 steps. T is seen by all groups; L/Z/+ only by C/D. S/V/C/A are held out for every group. Macro SR weights the four held-out shapes equally.

Shape / metric	A	B	C	D
T	99.33 ± 0.88	98.78 ± 0.96	99.22 ± 1.07	98.00 ± 2.00
L	0.33 ± 0.58	91.33 ± 10.58	99.33 ± 0.67	98.56 ± 2.22
Z	0.00 ± 0.00	64.33 ± 27.82	95.22 ± 2.22	96.56 ± 2.46
+	0.44 ± 0.77	6.33 ± 5.90	97.56 ± 2.52	99.67 ± 0.00
S	0.22 ± 0.38	49.67 ± 29.87	79.78 ± 25.45	56.67 ± 35.47
V	0.11 ± 0.19	78.22 ± 10.72	94.78 ± 3.34	92.33 ± 7.84
C	0.00 ± 0.00	15.33 ± 6.11	48.78 ± 14.87	52.56 ± 45.02
A	0.00 ± 0.00	86.11 ± 9.71	93.11 ± 4.54	94.22 ± 2.34
Held-out macro SR	0.08 ± 0.08	57.33 ± 11.09	79.11 ± 10.07	73.94 ± 14.52
Worst-shape SR	0.00 ± 0.00	15.33 ± 6.11	48.78 ± 14.87	29.22 ± 26.26
Held-out mean steps	299.87 ± 0.13	198.06 ± 40.43	186.07 ± 24.70	188.89 ± 33.02

Table 10 reports across-construction mean and sample SD, rather than variability over the 300 evaluation episodes. Held-out macro SR first averages S/V/C/A equally within a run; worst-shape SR takes their minimum. The specification adds 57.25 percentage points to T-only mean held-out SR, with the largest remaining gap on C. Multishape development yields higher mean transfer under either specification, with no additional gain from the clause in these runs.

For the main table, we rank each group's three frozen artifacts by recorded validation SR, breaking ties by lower validation mean first-success steps with failures charged 300. Applying this rule to the archived validation summaries selects A1/B2/C2/D3, with validation SRs of 100/100/99/99% and mean steps of 59.89/72.60/171.59/103.37, respectively. Validation contains only each group's development shapes. Each selected artifact supplies all test-shape entries in its row; Table 11 retains success rates for all 12 runs.

The source package retains `push.md`, the brightness update, and machine-readable result tables; the generator reproduces aggregates from these records.

Table 11: All 12 frozen shape specialists: SR (%). Every cell evaluates 300 seeds. Run numbers identify independent constructions, not successive revisions.

Run	T	L	Z	+	S	V	C	A
A1	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
A2	99.67	0.00	0.00	0.00	0.00	0.33	0.00	0.00
A3	98.33	1.00	0.00	1.33	0.67	0.00	0.00	0.00
B1	99.33	99.33	40.00	5.33	37.67	82.67	10.00	97.00
B2	99.33	95.33	94.67	12.67	83.67	86.00	22.00	83.00
B3	97.67	79.33	58.33	1.00	27.67	66.00	14.00	78.33
C1	100.00	100.00	97.67	98.67	51.00	96.00	32.00	94.67
C2	98.00	99.33	94.67	94.67	89.00	91.00	54.00	88.00
C3	99.67	98.67	93.33	99.33	99.33	97.33	60.33	96.67
D1	100.00	96.00	95.67	99.67	50.00	83.33	5.33	92.00
D2	98.00	99.67	94.67	99.67	95.00	96.00	57.33	94.00
D3	96.00	100.00	99.33	99.67	25.00	97.67	95.00	96.67

D MINECRAFT: HELD-OUT WORLDS AND DESKTOP CONTROL

D.1 TASK, SPLITS, AND MEASUREMENTS

All four methods use a Mac mini M4, macOS, and Minecraft: Java Edition 26.3 in Survival mode, Normal difficulty, with Auto-Jump enabled and commands disabled. The 30 shared test worlds are disjoint from 25 development seeds. The source package retains both seed manifests and their overlap check. Each trial restores the same archived initial save; world 15 includes one brown mushroom, and no target item is initially present.

The 300 s budget starts after loading/setup. Each milestone requires an item visible in a pre-deadline screenshot, corroborated by terminal-save evidence; screenshot acquisition defines first-confirmation time, including for later annotations. SR uses all 30 worlds; completion times use successes. Seconds per gesture is $\sum_i \min(T_i, 300) / \sum_i A_i$, for wall time T_i and executed keyboard/mouse gestures A_i, including reasoning, service/tool waits, capture, and actions. Reuse counts coding-agent computation across worlds; specialist execution recurs locally in each episode.

D.2 OBSERVATION AND CONTROL INTERFACES

All methods use 854×508 RGB including a 28-pixel title bar. A native macOS library captures the Java game window; Python dispatches keyboard/mouse events through `CGEventPost`, while Accessibility handles focus. Requested duration segments are capped at one second and each look axis at 180 mouse units. Foreground checks and input release handle timeout, termination, and focus loss. World loading and scoring remain evaluator-side.

Compiled specialists. Both combine frozen DINOv3 ViT-S/16 features (Siméoni et al., 2025) and linear region classifiers with OCR, digit/HUD templates, and pixel rules. F3-assisted reads visible pose and targeted-block text; F3-free maintains relative heading and calibrates pitch from actions and image motion. Inventory and crafting use task-specific GUI routines, fixed slot coordinates, and encoded recipes. The specialist detects occupied slots, hovers for Tesseract tooltip OCR, reads counts, allocates ingredients, clicks, and rescans. An increased product count confirms crafting; up to three checks precede failure and replanning. Coordinates, templates, and OCR assume the recorded GUI scale, resource pack, and language. The archived GUI record retains exact coordinates and source locations.

Direct Astra. Xhigh and medium receive ordinary RGB with F3 disabled. Astra interprets screenshots and chooses mouse/key commands through the shared transport; specialist OCR and GUI routines are exclusive to compiled execution. Xhigh contributes 24 episodes from one session and six from another; medium runs 30 in a separate session. Each retains its own interaction history; seeds, biome labels, specialist code, other results, and hidden scoring remain evaluator-side.

D.3 RESULTS BY BIOME

Table 12 retains all four policies across eight biomes; overall rates weight this observed mixture. Table 13 lists the corresponding test seeds. Each method runs once per world, so small biome groups describe the sampled terrain outcomes.

Termination and records. F3-assisted has 18 goal stops, 11 timeouts, and one death; F3-free has nine goal stops, ten timeouts, two deaths, and nine exceptions. Xhigh has three goal stops, 25 timeouts, and two deaths; medium has five goal stops, 24 timeouts, and one death. All remain in the SR denominator. A milestone can be confirmed before the deadline even when the terminal label is timeout. F3-assisted world 3 includes a scored restart following two infrastructure interruptions. Medium includes 15 rejected post-termination requests and seven delayed annotations of pre-deadline screenshots; these preserve the original cutoff. The report and 120 episode rows retain these records and the per-world measurements.

Table 12: Minecraft milestone results by spawn biome on the test set. Each policy runs once per shared initial world. SR is completion within 300 s (%); t is mean first visual-confirmation time among successes (s). A dash denotes no success.

			Crafting table		Wooden pickaxe		Stone pickaxe	
Spawn biome	n	Policy	SR	t	SR	t	SR	t
Forest	9	F3-assisted	88.9	63.0	66.7	76.7	66.7	156.5
		F3-free	55.6	115.6	55.6	133.4	33.3	203.8
		Astra (xhigh)	44.4	165.5	22.2	202.2	11.1	233.0
		Astra (medium)	100.0	164.3	66.7	179.6	44.4	279.1
Plains	6	F3-assisted	33.3	133.5	33.3	145.7	33.3	226.3
		F3-free	50.0	166.2	50.0	186.7	16.7	271.3
		Astra (xhigh)	50.0	187.3	33.3	249.8	16.7	271.9
		Astra (medium)	83.3	205.1	50.0	246.0	0.0	—
Taiga	3	F3-assisted	100.0	43.6	100.0	52.7	66.7	153.1
		F3-free	100.0	127.3	100.0	151.4	33.3	183.0
		Astra (xhigh)	100.0	135.9	100.0	205.4	33.3	270.7
		Astra (medium)	100.0	154.8	66.7	165.4	33.3	213.2
Savanna	2	F3-assisted	100.0	70.9	100.0	80.4	100.0	159.4
		F3-free	100.0	67.3	100.0	81.3	0.0	—
		Astra (xhigh)	50.0	136.6	50.0	192.8	0.0	—
		Astra (medium)	100.0	135.2	100.0	183.5	50.0	287.4
Jungle	3	F3-assisted	100.0	109.3	100.0	126.2	100.0	204.6
		F3-free	100.0	109.5	100.0	125.1	66.7	197.5
		Astra (xhigh)	100.0	180.1	66.7	178.3	33.3	201.2
		Astra (medium)	100.0	195.2	66.7	198.5	33.3	232.6
Snowy	4	F3-assisted	100.0	122.7	75.0	150.5	50.0	182.4
		F3-free	50.0	138.6	50.0	150.3	25.0	281.0
		Astra (xhigh)	100.0	260.8	0.0	—	0.0	—
		Astra (medium)	100.0	182.6	75.0	196.8	25.0	294.9
Swamp	1	F3-assisted	100.0	47.0	100.0	56.7	100.0	137.8
		F3-free	100.0	45.3	100.0	56.0	100.0	121.7
		Astra (xhigh)	0.0	—	0.0	—	0.0	—
		Astra (medium)	100.0	155.6	100.0	189.0	100.0	278.8
Others	2	F3-assisted	100.0	65.5	50.0	65.4	0.0	—
		F3-free	50.0	122.6	50.0	134.2	0.0	—
		Astra (xhigh)	100.0	198.0	50.0	168.7	0.0	—
		Astra (medium)	100.0	134.8	100.0	191.8	0.0	—

Overall rates weight the observed biome mixture. Desert has no sampled worlds.

Table 13: The 30 shared Minecraft test seeds, grouped by spawn biome. All seeds are disjoint from specialist development.

Spawn biome	n	Test seeds
Forest	9	4834296712910080402, 7291408623162765033, 1382187940642021303, 98725203191101045, 5788054948697208719, −5598672410188875473, −5428637977870121785, −7634404828800179876, 515825757778021849
Plains	6	−3262223252979801651, 5317898608262583281, 1607951438177922389, 7990985454709331423, 5323574760380294518, 5201429215874031013
Taiga	3	738531013070951866, −3789465003865386793, 5201669397561643650
Savanna	2	−2110043219135487861, −6753477820431330407
Jungle	3	7693427745428915804, −7504421814536559417, −1689077412957444709
Snowy	4	8959897510035123532, −7855385437179703084, −625918394183225546, −1393046942603549186
Swamp	1	−184617843142904718
Others	2	8295121936602387515, 512576065903896814