MC-EVOBENCH: AUDITING EXPERIENCE-DRIVEN SELF-IMPROVEMENT IN MINECRAFT AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A central challenge in embodied recursive self-improvement is enabling agents to convert interaction history into capabilities that persist beyond one world and transfer to later tasks. A saved memory or successful final action does not by itself establish that a reusable capability was acquired; evaluation must separate capability change from carried-over resources, unequal source exposure, and early execution failures. We construct MC-EvoBench as a controlled Minecraft benchmark with a fixed catalog of 19 registered evolution chains, including 12 primary evaluation chains and seven calibration or probe chains, together with three graph-aligned paired probe families for transfer. The benchmark represents each task as a task-state graph, defines stage-level environment verifiers, assigns matched source and target roles, resets the target world, and permits only a declared agent artifact to cross the source–target boundary. For each probe, no-prior, related, unrelated, and shuffled source conditions are evaluated under matched environment-step budgets and five seeds; we report TSR, WGP, and VMP and retain provenance records. The benchmark reveals a consistent direct-capability gap between controller and planner-backed variants, while related experience yields no stable transfer: it does not improve either tool probe, its small wood-species gain hinges on one seed, and all intervals include zero. Stage audits further show that early wood-collection and crafting bottlenecks often leave later capabilities unobserved. MC-EvoBench thus offers a controlled testbed for durable, transferable embodied capabilities. Code and data are available at https://anonymous.4open.science/r/MC-EvoBench.

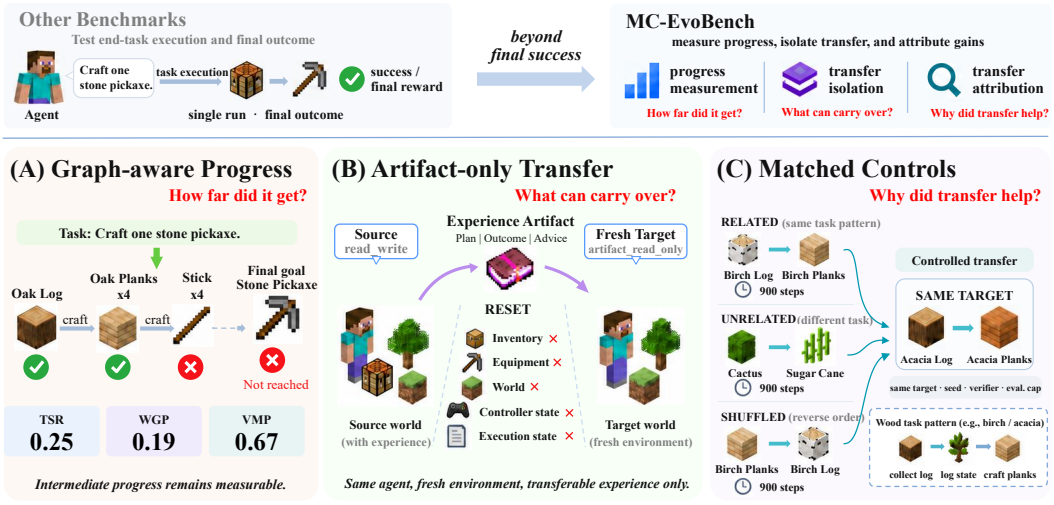

Figure 1: Evaluation mechanism of MC-EvoBench. A conventional benchmark records only whether an agent reaches the goal. MC-EvoBench also measures progress through the task graph, transfers only a declared experience artifact into a reset target world, and compares related, unrelated, and shuffled source conditions. We separate partial progress, physical carryover, and source-specific experience gains from improvements caused by favorable initial states or execution luck.

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.

1 INTRODUCTION

One of the central ambitions of autonomous intelligence is recursive self-improvement (RSI) (Chen et al., 2026): an agent should use experience from its own actions to become more capable over time. The goal is larger than learning one policy or storing one successful trajectory. An RSI system should turn experience into a capability, retain it, and apply it when the task, world, or required action changes. This loop is central to long-horizon autonomy: as tasks become harder, the agent must build on what it has already learned.

This view differs from evaluation settings in which training is separated from testing and improvement is summarized by a single score (Ren et al., 2026). For RSI, the important question is whether a gain remains useful after the original interaction has ended. A capability should survive a change in local conditions, help with a related goal, and combine with actions that were not part of the source task. RSI is therefore a systems problem: learning, memory, planning, and physical interaction must work together, while the resulting gain remains visible and attributable.

Embodied agents make this problem concrete and difficult. Their behavior depends on both internal state and physical state. A later success may reflect a tool left in the inventory, a changed world, a favorable location, or more source interaction rather than a reusable capability. A useful benchmark must therefore show that improvement survives a reset and appears in observable behavior.

When improved performance follows experience, does it reflect a reusable capability, or merely a changed physical state or execution condition?

We study this question through later behavior. A memory, skill file, or model update shows that an agent changed internally. It does not show that the agent gained a useful capability (Li et al., 2025b; Park et al., 2025; Li et al., 2026). The evidence must link internal change to later action. It must also distinguish capability from favorable execution. To test that claim, we compare target behavior after a world reset and check both task outcomes and stage-level records.

Existing benchmarks address parts of this problem. MCU (Zheng et al., 2023) evaluates diverse and combined Minecraft tasks with automatic verification. LifelongAgentBench (Zheng et al., 2025), EvoAgentBench (Gao et al., 2026), and PAST-Bench (Xue et al., 2026) study learning across tasks, transfer, and persistent state in other settings. Embodied evaluation also needs to separate world resources from agent state, match source budgets, and record where a task chain fails (Ni et al., 2025; Cheng et al., 2026; Luo et al., 2025). Otherwise, a score difference may reflect an easier world, more source interaction, or an early failure that hides later behavior.

Minecraft makes these controls practical. Collecting resources, crafting tools, and using them create visible task dependencies. A task-state graph represents these relations, and environment evidence verifies each stage. Resetting the world and inventory removes physical carryover, while stage records show progress before a chain stops (Guo et al., 2025; Huang et al., 2025).

We introduce MC-EvoBench to evaluate embodied RSI in Minecraft. Figure 1 shows its mechanism: measure progress, transfer only a declared agent artifact, and use matched controls to assess experience gains. Table 1 compares this design with representative Minecraft benchmarks (Guss et al., 2019; Fan et al., 2022; Hill et al., 2023; Wang et al., 2025). MC-EvoBench provides a task catalog, verifiers, and three short probe families for wood-species crafting, wooden-tool transfer, and stone-tool transfer. The framework can study capability formation, transfer, retention, and composition. This paper focuses on forward transfer between graph-related tasks.

Our main contributions are as follows:

1. **Embodied RSI as attributable capability change.** We formalize an agent as a policy, a persistent state, and an update rule, and separate this persistent state from the world state. A gain counts as experience-driven only when it appears after the target world is reset, only a declared artifact crosses the source–target boundary, and matched unrelated and shuffled sources rule out alternative explanations.
2. **MC-EvoBench, an executable audit of experience-driven improvement.** It instantiates this definition in Minecraft with a task-state graph of 80 tasks, 111 states, and 369

Table 1: Minecraft benchmark comparison for embodied RSI.

Dimension	MineRL	MineDojo	MCU	MinePlanner	OpenHA	Voyager	MC-EvoBench
Capability evolution	✗	✗	✗	✗	✗	✓	✓
Source isolation	✗	✗	✗	✗	✗	✗	✓
Experience persistence	✗	✗	✗	✗	✗	✓	✓
Capability transfer	✗	✗	✗	✗	✗	✓	✓
Retention and composition	✗	✗	✗	✗	✗	✗	✓
Gain attribution	✗	✗	✗	✗	✗	✗	✓

✓ Explicitly modeled or reported; ✗ not a primary design or evaluation dimension.

typed edges organized into 19 chains; stage-level verifiers with three complementary metrics, Task Success Rate (TSR), Weighted Graph Progress (WGP), and Verified Milestone Progress (VMP); and a four-condition paired protocol with matched environment-step budgets and provenance records.

3. **Controlled evidence on embodied transfer.** Controller-based agents clearly outperform planner-backed variants in direct capability. Under matched controls, the reference method shows no stable transfer gain, even though its related sources earn more verified progress and, on the longer probes, more computation, and early wood-collection and crafting bottlenecks leave later capabilities unobserved. Future RSI methods must therefore first clear these execution bottlenecks, and their gains should be claimed only after a controlled audit.

Section 2 reviews related benchmarks and agents. Section 3 defines the framework and benchmark. Section 4 reports the reference results. Appendices A–G give the task, environment, protocol, and reproduction details.

2 Related Work

Minecraft and embodied-agent benchmarks. Minecraft combines visual control, persistent resources, and tasks with many prerequisites. MineDojo provides a large open-world task collection and a knowledge base that supports broad embodied-agent research (Fan et al., 2022). BEDD extends MineRL (Guss et al., 2019) with demonstrations and evaluation for tasks whose success can be difficult to describe with a single exact goal. MinePlanner focuses on long-horizon planning in large Minecraft worlds, while OpenHA studies open-source hierarchical agents and their execution in the same environment (Hill et al., 2023; Wang et al., 2025). Voyager uses a curriculum, a skill library, and environment feedback for continued exploration and skill reuse, including in a new world (Wang et al., 2023). MCU evaluates a broad set of tasks, their combinations, and automatic assessment (Zheng et al., 2023). Together, these benchmarks cover open-world interaction, planning, task diversity, and automatic verification. They do not, however, isolate whether a controlled source interaction causes a later improvement after the target world is reset.

Lifelong learning and self-evolving agents. Several recent benchmarks study improvement across tasks and the use of stored experience. LifelongAgentBench tests skill gain, retention, and transfer across databases, operating systems, and knowledge graphs (Zheng et al., 2025). EvoAgentBench uses agent traces to identify abilities and groups tasks by shared procedures, making transfer between related tasks explicit (Gao et al., 2026). PAST-Bench compares runs with and without persistent state and tracks how experience is saved and reused in personal-agent settings (Xue et al., 2026). These studies establish useful ideas for measuring persistence and transfer, but their tasks do not expose the same physical-state confounds as embodied environments. In Minecraft, an apparent gain can instead come from a carried tool, a modified world, or a different starting location. MC-EvoBench addresses this boundary by resetting inventory and world changes while allowing only a declared agent artifact to transfer. Its task-state graph makes prerequisites and progress visible.

Memory-mediated improvement and benchmark position. Memory systems provide a practical mechanism for experience-driven improvement. Reflexion (Shinn et al., 2023) stores language feedback from failed attempts and uses it to guide later decisions. Agent Workflow Memory extracts

reusable procedures from interaction traces for later tasks (Wang et al., 2026). Skill libraries and model updates offer related ways to preserve information beyond a single episode. These methods show how an agent can store and retrieve experience, but reuse alone does not establish that a capability transferred. MC-EvoBench treats experience-driven improvement as an attribution problem. It separates source and target episodes, resets the physical state, and transfers only a declared artifact. Matched source conditions, verifier predicates, and stage records distinguish capability transfer from carried-over resources and partial execution.

3 MC-EvoBench Framework and Benchmark Design

MC-EvoBench tests whether experience changes later behavior. To support that claim, it resets the target world, checks outcomes against environment evidence, and compares matched source conditions. Figure 2 shows how these parts work together. Panel (a) follows the agent from source interaction to target evaluation. Panel (b) shows the tasks, states, relations, chains, and environment profiles that define each test.

3.1 Evaluation Target and State Boundary

We ask whether controlled source interaction improves performance on a later target task in a reset world. A stored memory or model update cannot answer this question by itself. We represent an agent as $A = (\pi, z, U)$: π selects actions from observations and goals, z is the state the method can retain, and U updates that state from experience. Source experience E_S produces $z_S = U(z_0, E_S)$. The evaluator records and freezes z_S before the target run. No source resources, positions, world edits, route information, or success markers enter the target.

World state includes inventory, tools, resources, position, terrain, and world changes. Persistent agent state may include memories, prompts, replay data, checkpoints, or model updates. Every target run starts with the same registered world and inventory settings. Thus, source tools and world changes cannot explain a target score difference. A stored artifact provides evidence of a gain only when the target verifier also records a change in behavior.

3.2 Task-State Graph

MC-EvoBench represents task dependencies and observable resource flow with

$$G = (\mathcal{T}, \mathcal{S}, \mathcal{E}),$$

Here, $\mathcal{T}$ contains tasks, $\mathcal{S}$ contains observable states, and $\mathcal{E}$ contains typed relations. States include items, tools, locations, structures, and task completion. The graph describes what tasks require, produce, or consume without prescribing an action sequence. These three relation types form the strict execution graph. The remaining types, `enables` and `abstracts_to`, describe possible capability links and task categories; neither proves transfer. Appendix B.2 defines each edge type and gives their counts.

Tasks form stage chains $C = (\tau_1, \ldots, \tau_K)$. Each stage has a goal that the verifier checks against run evidence. The chain shows prerequisites and where execution stops; the agent remains free to choose its actions. Stage records reveal partial progress even when the final goal is not reached. Catalog and probe chains use the same task definitions and verifier rules. Reset settings, retained state, and step budgets are fixed before evaluation.

3.3 Paired Protocol and Metrics

Each comparison has a source phase and a target phase. The source agent interacts with registered tasks and forms z_S. The evaluator then resets the target world and inventory, freezes z_S, and runs the target task with the same verifier. Four target conditions serve different purposes. No-prior provides a baseline; related experience tests for a change; unrelated experience tests whether that change depends on task relevance; and shuffled related experience tests whether task order matters. We record source steps, termination, artifact identity, and target outcomes. Source episodes restart after early termination until the declared budget is used.

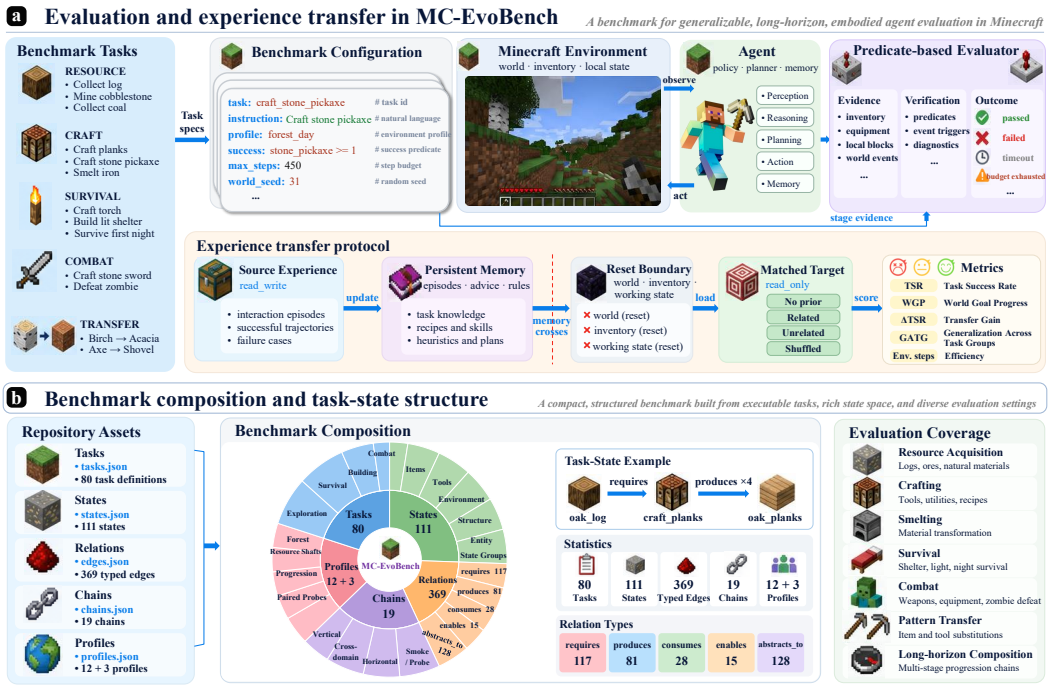

Figure 2: Benchmark design of MC-EvoBench. Panel (a) follows a paired evaluation from source interaction and artifact formation to target-world reset, target execution, stage verification, and metric reporting. Panel (b) shows how the task catalog is organized into tasks, states, relations, chains, and profiles. These components support graph-based progress measurement, controlled transfer probes, and coverage of embodied capabilities across long-horizon task chains.

The verifier records the evidence for each stage, changes in resources, pass status, and why the run stopped. Let v_i indicate whether stage i passes. For each strict execution edge e, w_e is its weight and q_e indicates verified progress. Let m_j indicate whether eligible milestone j is reached. The three direct metrics are

$$\mathrm{TSR} = \frac{\sum_{i=1}^{K} v_i}{K}, \tag{1}$$

$$\mathrm{WGP} = \frac{\sum_{e \in E_{\mathrm{exec}}} w_e q_e}{\sum_{e \in E_{\mathrm{exec}}} w_e}, \tag{2}$$

$$\mathrm{VMP} = \frac{\sum_{j=1}^{J} m_j}{J}. \tag{3}$$

TSR is the fraction of planned stages that pass; unreached stages remain in its denominator. WGP is the weighted fraction of strict graph edges supported by verified progress. VMP tracks eligible item or tool milestones observed before a chain ends. It captures partial progress, not full completion. We report all three metrics for each target condition and compare paired runs. Appendices C and D.6 give the verifier and reporting rules.

3.4 Benchmark Instantiation

MC-EvoBench implements this framework in Minecraft with a fixed task catalog, verifiers, probe profiles, and observation interface. A change to any of these components defines a new benchmark version. The same evaluation can therefore test different ways of storing experience, such as textual memories, skill libraries, prompts, or model updates, under a shared task and evidence specification.

Table 2: Direct capability matrix over the 12 primary evaluation chains, grouped by method family. All entries are measured and reported as TSR / WGP / VMP in percent; the final group is the weighted average over all 12 chains.

Method	Core progression (5)			Catalog transfer (4)			Diversity (3)			Average (12)		
	TSR	WGP	VMP	TSR	WGP	VMP	TSR	WGP	VMP	TSR	WGP	VMP
Environment baseline												
no-op	3.5	1.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.5	0.6	0.0
STEVE and JARVIS controller variants												
STEVE-1	12.9	8.8	36.1	7.5	6.9	27.1	1.9	1.9	5.6	8.4	6.4	25.5
JARVIS-1 (controller-only)	9.9	5.7	26.7	8.7	8.0	37.5	1.9	1.9	5.6	7.5	5.5	25.0
JARVIS-1 (planner:)	3.5	1.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.5	0.6	0.0
JARVIS-1 (planner:)	4.9	2.7	4.4	1.1	1.1	5.5	0.0	0.0	0.0	2.4	1.5	3.7
JARVIS-1 (planner:)	3.5	1.5	0.0	1.5	1.9	5.5	0.0	0.0	0.0	2.0	1.3	1.8
JARVIS-1 (planner:)	4.6	2.0	6.7	2.5	1.6	9.7	0.0	0.0	0.0	2.8	1.4	6.0
JARVIS-1 (planner:)	3.5	1.5	0.0	1.1	1.1	5.5	0.0	0.0	0.0	1.8	1.0	1.8
Optimus variants												
Optimus-1 (controller-only)	12.5	8.5	33.3	7.2	6.1	30.5	1.9	1.9	5.6	8.1	6.1	25.4
Optimus-1 (planner:)	6.7	4.1	11.7	1.7	1.3	5.5	0.0	0.0	0.0	3.4	2.1	6.7
Optimus-1 (planner:)	6.7	4.1	11.7	3.2	3.2	11.1	0.0	0.0	0.0	3.9	2.8	8.6
Optimus-1 (planner:)	7.4	4.4	15.6	3.3	1.9	13.9	1.9	1.9	5.6	4.7	2.9	12.5
Optimus-1 (planner:)	9.2	5.7	19.5	3.2	3.2	11.1	0.0	0.0	0.0	4.9	3.4	11.8
Optimus-1 (planner:)	6.7	4.1	11.7	3.3	1.9	13.9	0.0	0.0	0.0	3.9	2.3	9.5
Optimus-3 action path	3.5	1.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.5	0.6	0.0
VPT variants												
VPT (STEVE-1 checkpoint)	4.2	1.8	3.3	1.5	1.9	5.5	0.0	0.0	0.0	2.3	1.4	3.2
VPT (official 2x foundation weights)	4.9	2.1	6.7	2.3	0.9	12.5	5.2	5.2	13.9	4.1	2.5	10.4

3.4.1 Frozen Catalog and Probe Families

The catalog contains 80 tasks, 111 state nodes, 369 graph edges, 19 registered chains, 12 general profiles, and 3 paired-transfer profiles. Its 19 chains include 12 primary evaluation chains and 7 calibration or probe chains. Tasks span building, survival, exploration, and combat. Long chains test execution across many prerequisites. Short probes test local transfer that an early failure in a long chain could hide. The probes use catalog tasks and the same verifier rules. Appendices F and A.3 list the tasks, chains, and profiles.

The three probe families and their budgets are listed in Table D2 of Appendix D.3. In each family, related and target chains share a dependency pattern but end with a different object or tool. This lets us test transfer while keeping the required preparation steps. Each stage has a 450-step cap. After early termination, a new episode starts until the family budget is used.

3.4.2 Environment and Observation Interface

Paired-transfer runs use public world seed 31, an empty target inventory, and one of three fixed environment profiles (Appendix A.3); direct-capability runs use the seeds and per-stage profiles in Table F9. Profiles set time, weather, spawn, clearance, and public resources. They do not provide target items, routes, private coordinates, or verifier state. The agent must observe and collect resources through interaction. Each target condition starts with a world and inventory reset; only the declared agent artifact can transfer. The agent sees normal observations, while the graph, verifier rules, diagnostics, and hidden coordinates remain unavailable. Appendices A.2 and C.1 detail these access rules and verifier checks.

4 Experimental Results

We report two complementary views of the benchmark. The direct capability matrix measures immediate performance across the registered catalog without a source–target transfer claim. The paired probes then test whether a declared source artifact changes behavior on a later target after the world is reset. Keeping these views separate distinguishes basic execution ability from experience-driven improvement.

Table 3: Paired transfer results for the three probe families. Values are target-condition means over five matched seeds; each cell lists no-prior (N), related (R), unrelated (U), and shuffled (S) conditions in that order. TSR, WGP, and VMP measure stage completion, graph progress, and verified milestone progress, respectively, after the target world is reset.

Probe family	**Target-condition means**		
	TSR (N/R/U/S)	**WGP** (N/R/U/S)	**VMP** (N/R/U/S)
Wood species	0.400 / 0.400 / 0.200 / 0.500	0.360 / 0.400 / 0.200 / 0.480	0.400 / 0.400 / 0.200 / 0.500
Wooden tool	0.320 / 0.200 / 0.160 / 0.240	0.225 / 0.150 / 0.100 / 0.167	0.560 / 0.367 / 0.333 / 0.467
Stone tool	0.150 / 0.050 / 0.025 / 0.075	0.082 / 0.021 / 0.010 / 0.031	0.533 / 0.200 / 0.100 / 0.300

4.1 Evaluation Setup

The paired evaluation uses three probe families: wood-species crafting, wooden-tool transfer, and stone-tool transfer. Each family uses five paired agent seeds and four target conditions: no prior, related experience, unrelated experience, and shuffled related experience. Every target starts from the same world and inventory state for its family. Only the declared agent artifact may cross the source–target boundary. We report TSR, WGP, and VMP as defined in Section 3.3. Source exposure is matched within each family; planner calls and token usage are recorded separately because equal environment exposure does not imply equal computation.

4.2 Direct Capability Matrix

The direct matrix covers 12 primary evaluation chains (five core-progression, four catalog-transfer, and three diversity) and nine direct configurations, each evaluated with three seeds. Each cell in Table 2 averages TSR / WGP / VMP over the chains in a partition; the final columns weight the partitions by their sizes (5/4/3), and Appendix Table F8 gives per-chain scores. These runs measure direct capability rather than transfer; ExpeL (Zhao et al., 2024) has one separately marked report per chain and is excluded from these aggregates.

Controller-based execution clearly outperforms planner-backed execution. STEVE-1 (Lifshitz et al., 2023) and the controller-only JARVIS-1 (Wang et al., 2024) and Optimus-1 (Li et al., 2024) variants reach average TSR/WGP/VMP of 8.4/6.4/25.5, 7.5/5.5/25.0, and 8.1/6.1/25.4, with the largest margins on the core and catalog-transfer partitions, where progress requires several dependent actions. The no-op baseline and the GPT-planned JARVIS-1 remain at the execution floor, confirming that the benchmark separates active task solving from background behavior. No planner dominates all three metrics: DeepSeek gives the highest average TSR for both JARVIS-1 (2.8) and Optimus-1 (4.9), whereas the highest VMP comes from DeepSeek for JARVIS-1 (6.0) but from GLM for Optimus-1 (12.5). The official VPT (Baker et al., 2022) foundation weights improve over the STEVE-1 checkpoint (average VMP 10.4 versus 3.2) but remain below the controller variants.

Finding 1: Capability is first bounded by execution. *Controllers consistently outperform planner-backed variants, and no planner leads on all three metrics.*

4.3 Paired Transfer Results

Table 3 reports mean target scores in the order no-prior / related / unrelated / shuffled. The study contains 105 source and target role reports in total, with 60 target reports contributing to the four-condition comparison.

For wood-species crafting, related experience gives scores close to no prior. The shuffled control scores at least as high on TSR and WGP. For wooden-tool and stone-tool transfer, related experience scores below no prior on all three metrics. On the stone-tool probe, related TSR, WGP, and VMP are 0.050, 0.021, and 0.200, compared with 0.150, 0.082, and 0.533 for no prior.

Finding 2: Stored experience is not acquired capability. *After a world reset with artifact-only transfer, related experience does not reliably exceed no prior and lowers all three metrics on both tool probes.*

Table 4: Paired transfer contrasts over five matched seeds. Columns are grouped by probe family and report the related-source change (ΔR), the diagnostic GATG contrast, and its two-sided 95% Student-t confidence interval. GATG is not an additional benchmark metric.

Metric	Wood species			Wooden tool			Stone tool		
	ΔR	GATG	95% CI	ΔR	GATG	95% CI	ΔR	GATG	95% CI
TSR	0.000	0.050	[-0.723, 0.823]	-0.120	0.000	[-0.152, 0.152]	-0.100	0.000	[-0.095, 0.095]
WGP	0.040	0.060	[-0.718, 0.838]	-0.075	0.017	[-0.132, 0.166]	-0.062	0.000	[-0.039, 0.039]
VMP	0.000	0.050	[-0.723, 0.823]	-0.193	-0.033	[-0.248, 0.181]	-0.333	0.000	[-0.380, 0.380]

Table 5: Observed source-side cost under matched family-level exposure. Values are mean ± standard deviation where applicable; environment steps are fixed within each probe family, while episode counts, source TSR, planner calls, and termination states are recorded from the runs.

Family	Source	Execution budget		Observed cost		Termination summary
		Env. steps	Episodes	Source TSR	LLM calls	
Wood species	Related	900 ±0	2.00 ±0.00	0.100 ±0.224	4.40 ±0.89	budget_exhausted=1; passed=1; timeout=9
Wood species	Unrelated	900 ±0	2.60 ±0.89	0.000 ±0.000	5.80 ±2.68	budget_exhausted=2; passed=6; timeout=8
Wood species	Shuffled	900 ±0	2.00 ±0.00	0.000 ±0.000	4.00 ±0.00	timeout=10
Wooden tool	Related	2,250 ±0	5.00 ±0.71	0.080 ±0.110	14.80 ±1.10	budget_exhausted=5; passed=12; timeout=20
Wooden tool	Unrelated	2,250 ±0	6.00 ±0.00	0.000 ±0.000	12.00 ±0.00	budget_exhausted=5; timeout=25
Wooden tool	Shuffled	2,250 ±0	5.00 ±0.00	0.000 ±0.000	10.00 ±0.00	timeout=25
Stone tool	Related	3,600 ±0	8.20 ±0.84	0.150 ±0.105	24.80 ±4.60	budget_exhausted=5; passed=21; timeout=36
Stone tool	Unrelated	3,600 ±0	8.00 ±0.00	0.000 ±0.000	16.40 ±0.89	budget_exhausted=1; passed=1; timeout=39
Stone tool	Shuffled	3,600 ±0	8.00 ±0.00	0.000 ±0.000	16.00 ±0.00	timeout=40

4.4 Paired Contrasts and Uncertainty

For each direct metric, let M_0, M_R, M_U, and M_S denote the no-prior, related, unrelated, and shuffled target scores. We report the related-source change $\Delta R = M_R - M_0$. For completeness, Table 4 also reports the adjusted contrast

$$\text{GATG} = M_R - \frac{1}{2}(M_U + M_S).$$

GATG is used here only as a descriptive paired contrast; TSR, WGP, and VMP remain the benchmark's direct metrics. Intervals are two-sided 95% Student-t intervals over the five paired seeds.

Table 4 separates the direct related-source change from the control-adjusted contrast. Relative to no prior, related experience improves only WGP on the wood-species probe ($\Delta R = +0.040$); TSR and VMP are unchanged. The two tool-transfer probes show negative changes on all three metrics, with the largest decrease in VMP for stone-tool transfer ($\Delta R = -0.333$). After adjusting for the unrelated and shuffled controls, the wood-species probe retains a small positive GATG on TSR, WGP, and VMP ($+0.050$, $+0.060$, and $+0.050$). The wooden-tool probe is mixed: GATG is near zero for TSR and slightly positive for WGP, but negative for VMP. All reported confidence intervals include zero, and the intervals are wide because they are estimated from five matched seeds.

Finding 3: Self-improvement cannot be claimed without controls. *Once unrelated and shuffled sources are accounted for, the largest contrast is only $+0.06$ and every 95% interval includes zero.*

4.5 Source Cost and Run Records

Matching environment steps does not equalize episodes or planner calls. Table 5 reports the observed source cost: step budgets are fixed within each family (900, 2,250, and 3,600), while episode counts, LLM calls, and termination states depend on how each run proceeds.

The related condition is the only one with nonzero source TSR (0.100, 0.080, and 0.150) and accounts for 34 of the 41 `passed` records, whereas shuffled runs end only by timeout. It also uses

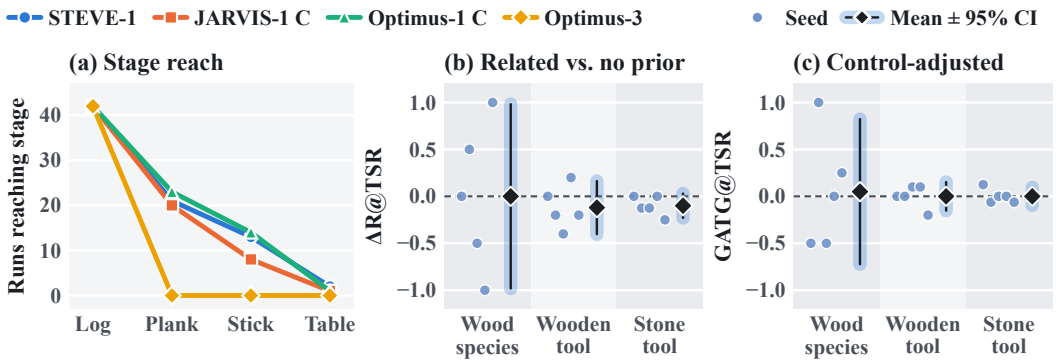

Figure 3: Failure-stage and per-seed diagnostics. (a) Runs, out of 42 attempts per configuration, reaching each wood and crafting stage. (b) Related-source ΔR and (c) control-adjusted GATG on TSR; dots are matched seeds; diamonds with shaded bars give the mean and 95% Student-t interval.

more planner calls on the two longer probes, reaching 24.80 ± 4.60 calls for stone-tool transfer versus 16.40 and 16.00 for the unrelated and shuffled controls.

Finding 4: More experience alone does not yield transfer. *On the longer probes, related sources gain more verified progress and planner calls than the controls, yet this extra experience yields no target gain.*

4.6 Failure Stages, Seed Variation, and Scope

In the failure-stage audit of the 57 direct-matrix reports per configuration (Figure 3a), attrition concentrates in the first crafting steps: of the 42 reports that include wood collection, the three controller variants bring 20–23 to plank crafting, at most 14 to sticks, and at most 2 to the crafting table, while the Optimus-3 (Li et al., 2025a) action path never passes the initial wood stage. A low final score can therefore reflect an early execution bottleneck rather than the failure of a later capability, which remains unobserved. Per-seed effects (Figure 3b–c) vary most for wood species, where the small positive mean GATG ($+0.050$) is driven by a single seed ($+1.0$) and falls to -0.19 without it; the tool probes vary less and have GATG means near zero. All three intervals cross zero, consistent with Table 4.

Taken together, the direct matrix measures immediate capability rather than transfer, and the paired probes should be read against its low execution floor. Under the current public instances, seeds, budgets, and reference method, related experience does not produce a stable positive transfer effect; this does not rule out gains under a different agent, artifact, task distribution, or source budget. Appendices C.3, D.6, and E.4 give the stage records, paired-reporting rules, and configuration fields needed to reproduce these comparisons.

Finding 5: Unobserved capabilities cannot be evaluated. *Most runs stop in the first crafting stages and the wood-species gain hinges on one seed, so progress needs stage-level evidence, not final success.*

5 Conclusion

We introduced MC-EvoBench, a controlled Minecraft benchmark for auditing whether experience yields durable and transferable embodied capabilities. Its task-state graph, stage-level verifiers, and paired reset-target protocol with matched control sources separate capability change from carried-over resources, unequal exposure, and early execution failures. In the reference study, controller-based agents clearly outperform planner-backed variants, yet the reference method gains no stable target benefit from related experience: the only positive contrast, on wood species, is driven by a single seed, and all intervals include zero even though related sources earn more verified progress and, on the longer probes, more computation. Most runs also end before crafting sticks, leaving later capabilities unobserved. These conclusions are limited to the tested agents, probes, seeds, and budgets; retention, composition, stronger agents, and more seeds are natural next steps.

AI USE STATEMENT

We used generative AI tools for English translation, language editing, shortening text, and LaTeX formatting. We did not use AI to generate data, run experiments, or draw the final conclusions. The authors checked all citations, equations, experimental settings, and results. They reviewed all AI-assisted text and take full responsibility for the paper.

ETHICS STATEMENT

This work evaluates agents in a virtual Minecraft environment. It does not involve human participants, personal data, or physical-world experiments. We use environment states and agent interaction traces to measure task execution and experience transfer. Any released code, task definitions, assets, and logs will follow their applicable licenses and include documentation for reproduction. The results apply to the tested methods, tasks, environments, and budgets. They do not establish that these agents are safe or capable in real-world settings.

REPRODUCIBILITY STATEMENT

Section 3 defines the task-state graph, source–target protocol, and metrics. Section 4 reports the reference results. Appendices A–C give the environment profiles, task and chain definitions, verifier rules, stage records, and metric calculations. Appendix D describes resets, source budgets, target roles, seeds, artifact transfer, and paired reporting. Appendix G gives the memory formats, prompts, example configuration, evaluation procedure, and reproduction commands. Run records include the fixed task and verifier settings, seeds, source exposure, artifact identities, and stage evidence needed to check the reported scores.

REFERENCES

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

Mingguang Chen, Licheng Wang, and Bo Qu. Recursive self-improvement in ai: From bounded self-refinement to autonomous research loops. *arXiv preprint arXiv:2607.07663*, 2026.

Zhili Cheng, Ran Li, Jinyi Hu, Yuge Tu, Shiqi Dai, Shengding Hu, Yang Shi, Lei Shi, and Maosong Sun. Embodiedeval: Evaluate multimodal llms as embodied agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11420–11432, 2026.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35: 18343–18362, 2022.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, et al. Evoagentbench: Benchmarking agent self-evolution via ability transfer. *arXiv preprint arXiv:2607.05202*, 2026.

Junliang Guo, Yang Ye, Tianyu He, Haoyu Wu, Yushu Jiang, Tim Pearce, and Jiang Bian. Mineworld: a real-time and open-source interactive world model on minecraft. *arXiv preprint arXiv:2504.08388*, 2025.

William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

William Hill, Ireton Liu, Anita De Mello Koch, Damion Harvey, Nishanth Kumar, George Konidaris, and Steven James. Mineplanner: A benchmark for long-horizon planning in large minecraft worlds. *arXiv preprint arXiv:2312.12891*, 2023.

Junchao Huang, Xinting Hu, Boyao Han, Shaoshuai Shi, Zhuotao Tian, Tianyu He, and Li Jiang. Memory forcing: Spatio-temporal memory for consistent scene generation on minecraft. *arXiv preprint arXiv:2510.03198*, 2025.

Chenghao Li, Jun Liu, Songbo Zhang, Huadong Jian, Hao Ni, Lik-Hang Lee, Sung-Ho Bae, Guoqing Wang, Yang Yang, and Chaoning Zhang. Experience transfer for multimodal llm agents in minecraft game. *arXiv preprint arXiv:2604.05533*, 2026.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks. *Advances in neural information processing systems*, 37:49881–49913, 2024.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Weili Guan, Dongmei Jiang, Yaowei Wang, and Liqiang Nie. Optimus-3: Dual-router aligned mixture-of-experts agent with dual-granularity reasoning-aware policy optimization. *arXiv preprint arXiv:2506.10357*, 2025a.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-2: Multimodal minecraft agent with goal-observation-action conditioned policy. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9039–9049. IEEE, 2025b.

Shalev Lifshitz, Keiran Paster, Harris Chan, Jimmy Ba, and Sheila McIlraith. Steve-1: A generative model for text-to-behavior in minecraft. *Advances in Neural Information Processing Systems*, 36: 69900–69929, 2023.

Yulin Luo, Chun-Kai Fan, Menghang Dong, Jiayu Shi, Xiangju Mi, Mengdi Zhao, Bo-Wen Zhang, Cheng Chi, Jiaming Liu, Gaole Dai, et al. Robobench: A comprehensive evaluation benchmark for multimodal large language models as embodied brain. *arXiv preprint arXiv:2510.17801*, 2025.

Fei Ni, Min Zhang, Pengyi Li, Yifu Yuan, Lingfeng Zhang, Yuecheng Liu, Peilong Han, Longxin Kou, Shaojin Ma, Jinbin Qiao, et al. Embodied arena: A comprehensive, unified, and evolving evaluation platform for embodied ai. *arXiv preprint arXiv:2509.15273*, 2025.

Junyeong Park, Junmo Cho, and Sungjin Ahn. Mrsteve: Instruction-following agents in minecraft with what-where-when memory. In *International Conference on Learning Representations*, volume 2025, pp. 16922–16960, 2025.

Zhe Ren, Yimeng Chen, Dandan Guo, Guowei Rong, Tonghui Li, RB Xiong, Qingfeng Lan, Wenyi Wang, Li Nanbo, Yibo Yang, et al. Self-improvements in modern agentic systems: A survey. *arXiv preprint arXiv:2607.13104*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

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. *arXiv preprint arXiv:2305.16291*, 2023.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, et al. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(3):1894–1907, 2024.

Zihao Wang, Muyao Li, Kaichen He, Xiangyu Wang, Zhancun Mu, Anji Liu, and Yitao Liang. Openha: A series of open-source hierarchical agentic models in minecraft. *arXiv preprint arXiv:2509.13347*, 2025.

Shuhan Xue, Zixin Ding, Yichen Shen, Yinjie Wang, Zhenfei Yin, Yingcheng Wu, Yuxin Chen, Mengdi Wang, and Ling Yang. Past-bench: Benchmarking the foundations of recursive self-improvement in personal agents. *arXiv preprint arXiv:2608.04003*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

Xinyue Zheng, Haowei Lin, Kaichen He, Zihao Wang, Zilong Zheng, and Yitao Liang. Mcu: An evaluation framework for open-ended game agents. *arXiv preprint arXiv:2310.08367*, 2023.

A MINECRAFT ENVIRONMENT SETTINGS

MC-EvoBench evaluates whether interaction experience can improve subsequent embodied execution. Its experimental unit joins a source experience, a persistent agent artifact, and a separately initialized target evaluation. This appendix specifies the environment, task construction, verification, experimental protocol, agent interfaces, task suite, and reproducibility materials.

A.1 GAME WORLD AND INTERACTION INTERFACE

The primary environment is Minecraft 1.16.5 through MineRL, with a custom survival specification named `MCBenchHumanSurvival-v0`. It extends the HumanSurvival environment with observations for inventory, local resources, world conditions, and task events. The agent interacts with the game through an adapter, while a separate evaluator converts the environment response into a normalized observation snapshot. This separation supports a common task definition across agents with different planning and control architectures.

The native control interface consists of movement, camera rotation, attack, use, and inventory operations. Camera commands change pitch and yaw; the meaning of attack and use depends on whether the agent is interacting with the world or an inventory interface. These environment actions produce the observations used for task verification. Text-conditioned controllers translate a task instruction into this action interface. A programmatic agent may use a different bridge, in which case its action granularity is reported with the method settings in Appendix E.

Table A1. Interaction and observation interfaces. The evaluator consumes the evidence needed by a task; an agent consumes the modalities exposed by its declared adapter.

Component	Representation	Role
Visual input	First-person RGB frame	Scene perception and low-level control
Movement	Forward, back, left, right; jump, sneak, sprint	Navigation and locomotion
Camera	Relative pitch and yaw	Viewpoint and interaction targeting
Interaction	Attack, use, drop, inventory, hotbar selection	Resource collection, construction, and item manipulation
Inventory and equipment	Item counts; normalized equipped-item names	Agent context when exposed; task verification
Local resource evidence	Ray observation and block-count grid	Structured perception or evaluator evidence, according to the adapter
World and mission evidence	Time, light, survival state, entity-kill counters	Event verification and execution diagnostics

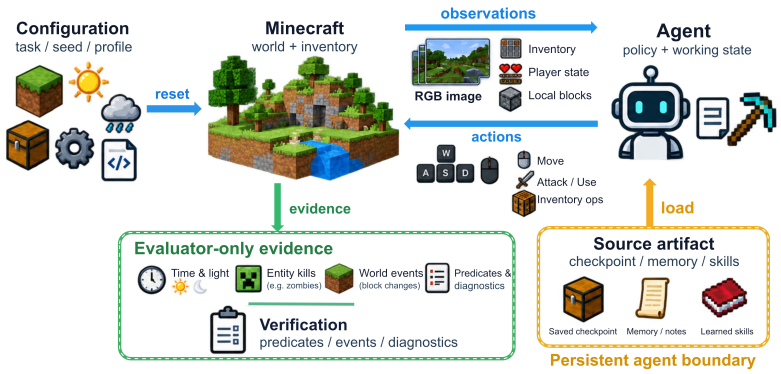

Figure A1. The task specification connects initialization, agent execution, and evaluation. World configuration fixes the problem instance. Agent inputs follow the selected interface, while success is evaluated from environment evidence. Persistent experience crosses source and target runs through the artifact boundary defined in Appendix D.

A.2 Observation access and normalized evidence

The catalog distinguishes `native_perception` and `structured_state` observation sources. Each source records a path, its primary or diagnostic role, its interpretation, and an access declaration. Inventory, equipment, and basic survival measurements are examples of declared agent-visible sources. Local block counts, mission flags, kill counters, and world-light diagnostics are classified as structured evidence. RGB-based and structured-state methods therefore remain distinguishable even when they share the same task verifier.

At runtime, the snapshot groups observations into inventory, equipment, player statistics, counters, flags, raw observations, and environment metadata. Equipment names are normalized across MineRL representations, and raw entity-kill observations can populate the corresponding normalized counter. The evaluator preserves the raw channel alongside the normalized value, which makes it possible to trace a predicate back to its observation source. The actual model inputs are determined by the adapter; Appendix E describes the built-in interfaces.

The local block grid spans offsets from $(-3, -2, -3)$ to $(3, 2, 3)$ around the player, giving a $7 \times 5 \times 7$ observation volume. The ray channel supplies a block identity and distance. Grid counts support local structure checks; ray observations support direct resource visibility. Profile-provided resource markers describe the configured scene and serve as diagnostics. The coal-discovery predicate uses coal observed in the ray channel or local block grid.

A.3 Environment profiles and controlled resources

An environment profile specifies a world seed, time of day, weather, initial inventory, and optional placement metadata. The benchmark includes 12 general environment profiles, all with empty initial inventories. Four forest profiles provide progression, daytime, nighttime, and dusk settings. Controlled-resource profiles place scarce materials near the spawn so that a finite evaluation budget can probe tool preparation, mining, collection, and downstream use. Ore is a world object that the agent must interact with.

Table A2. General environment profile registry. Codes are used in Appendix F. Seeds are profile defaults; an experiment can override them explicitly. All entries initialize an empty inventory.

Code	Profile	Seed	Time / controlled setting
P01	`forest_progression`	42	Tick 1,000; time progresses
P02	`forest_day`	7	Tick 1,000; daylight
P03	`forest_night`	11	Tick 14,000; hostile spawning enabled
P04	`forest_dusk`	9	Tick 12,000; dusk
P05	`fixed_coal_shaft`	21	Exposed coal; radius 4; catalog y-band 73–74
P06	`fixed_iron_shaft`	21	Iron; radius 4; catalog y-band 70–71
P07	`fixed_gold_shaft`	21	Gold; radius 4; catalog y-band 70–71
P08	`fixed_redstone_shaft`	21	Redstone; radius 4; catalog y-band 70–71
P09	`fixed_lapis_shaft`	21	Lapis; radius 4; catalog y-band 70–71
P10	`fixed_diamond_shaft`	21	Diamond; radius 4; catalog y-band 70–71
P11	`fixed_emerald_shaft`	21	Emerald; radius 4; catalog y-band 70–71
P12	`fixed_ore_progression_shaft`	21	Seven-resource gallery; radius 6; catalog y-band 70–71

Resource placement is specified relative to the configured or actual spawn, as recorded in the profile. The multi-resource gallery contains coal, iron, gold, redstone, lapis, diamond, and emerald. These local arrangements standardize access to task materials; they measure execution under controlled resource availability. The source files retain block placement, facing direction, resource identity, and allowed-task metadata.

The paired-transfer configurations use three additional profiles: a mixed wood-species grove, an oak grove, and a stone gallery. They use world seed 31, daylight, clear weather, disabled spawning, and an empty initial inventory. Their nearby resources support paired source and target tasks with comparable access conditions. Related and unrelated source content, interaction budgets, and target substitutions are specified together in Appendix D.

A.4 STATE CONTINUITY WITHIN A TASK CHAIN

The chain specification selects either a task-level or chain-level profile policy. Under the task-level policy, the runner selects each task's profile; with the default per-stage reset setting, a profile change causes an environment reset. Under the chain-level policy, one selected profile governs the chain. Consecutive stages that remain in the same environment can reuse world resources and inventory, making state continuity an explicit part of chain evaluation.

This within-chain continuity is separate from the source-to-target boundary. A target transfer evaluation starts from its prescribed environment state and receives only the permitted persistent artifact. A tool retained between two stages changes their execution conditions; a tool reconstructed after a target reset provides evidence about execution under the retained agent state. The protocol records these two situations separately.

B Task Construction and Composition

B.1 Task specification and benchmark coverage

A task is a compact executable contract: an identifier, an instruction, a domain, a step budget, an environment profile, a success specification, and observation-source declarations. Instructions describe ordinary Minecraft objectives such as collecting a resource or crafting a tool. The success specification supplies the measurable endpoint. Keeping these fields together allows the same task to be placed in a short probe, a longer chain, or an independent evaluation without redefining its meaning.

The task catalog contains 80 tasks across building, survival, exploration, and combat. The domain counts are 40, 26, 7, and 7, respectively. The accompanying graph contains 111 state nodes and 369 edges. Its composition rules connect resource acquisition, crafting, tool use, environmental conditions, and high-level events. Appendix F provides the complete task and chain definitions; the machine-readable tables retain the original instructions and predicates.

Table B1. Catalog partitions and their execution denominators. A task identifier can occur more than once in a chain or across chains. Stage occurrences are therefore counted separately from distinct task membership.

Partition	**Declared tasks**	**Chains**	**Stage occurrences**	**Purpose**
Core progression	33	5	71	Resource, tool-tier, survival, and combat progression
Transfer	31	4	50	Item, recipe, and schema substitutions
Supplementary diversity	16	3	19	Breadth of collection and resource interaction
Calibration and probe	0	7	51	Execution and verifier checks
Primary evaluation total	80	12	140	Core, transfer, and diversity combined

The first three partitions define the primary evaluation suite. The calibration partition reuses task definitions for engineering checks and is reported separately. This organization supplies both a stable comparison set and a small set of diagnostic entry points. The task-membership split describes the catalog taxonomy; shared prerequisites can still recur in the executable chains.

B.2 Task–state graph

Let $\mathcal{T}$ denote task nodes and $\mathcal{S}$ state nodes. A state represents an item, tool, location, environmental condition, player condition, structure, entity event, or abstract group. Directed typed edges specify how a task relates to these states. The resulting graph is

$$\begin{aligned} G &= (\mathcal{T}, \mathcal{S}, \mathcal{E}), \\ \mathcal{E} &= \mathcal{E}_{\text{req}} \cup \mathcal{E}_{\text{prod}} \cup \mathcal{E}_{\text{cons}} \cup \mathcal{E}_{\text{enable}} \cup \mathcal{E}_{\text{abs}}. \end{aligned} \tag{B.1}$$

Table B2. Edge types in the released catalog. Counts describe graph definitions, before expansion into repeated stage occurrences.

Relation	**Direction**	**Count**	**Meaning**
`requires`	State to task	117	Hard prerequisite for ordered composition
`produces`	Task to state	81	Declared output of a completed task
`consumes`	Task to state	28	Resource used by the task
`enables`	State to task	15	Soft capability relation for probe design
`abstracts_to`	Node to group	128	Semantic grouping across concrete tasks and states

The relations in Eq. B.1 are listed in Table B2. The first three define the execution paths used in weighted graph progress. Soft enabling and abstraction edges organize capability relationships without contributing to this execution score. This makes the graph useful at two levels: concrete dependencies support composition, while abstract groups organize transfer questions across items or recipes.

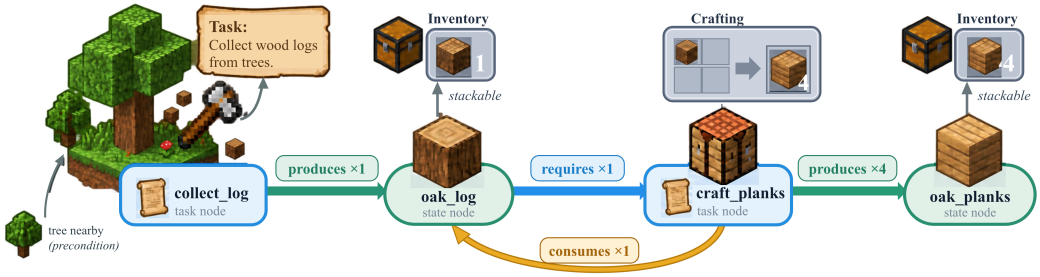

Figure B1. A catalog fragment links log collection to plank production. Task nodes and state nodes have distinct shapes; edge labels retain their direction and quantity. The graph describes a declared dependency structure. Runtime observations and task predicates determine which execution stages actually pass.

B.3 Ordered chain compilation

A chain stores an ordered task sequence, initial symbolic states, an environment profile, and a profile policy. The compiler traverses the sequence and extracts five paths for every occurrence: hard prerequisites, produced resources, consumed resources, soft enabling relations, and abstraction relations. Repeated task identifiers become distinct compiled stages, so their positions and local context remain available to the evaluator.

The default compiler checks prerequisite presence in the evolving symbolic state set. An optional strict-quantity mode also checks declared resource counts before consuming and producing them. Let $q_j(s)$ be the symbolic availability of state s before stage j. In the strict mode, a stage requires sufficient declared counts, after which its resource bookkeeping follows

$$q_{j+1}(s) = q_j(s) - c_j(s) + p_j(s), \tag{B.2}$$

where c_j and p_j are the stage's consumption and production counts. Initial symbolic states seed this calculation; they do not grant physical inventory. The simulator independently realizes movement, mining, crafting, and resource use.

This separation is useful when constructing a benchmark: a chain can be checked for intended ordering before a game is launched, and its execution can then be evaluated from observations. Equation B.2 describes symbolic bookkeeping only. The compilerś output also gives each stage a fixed metric context, with graph weights fixed by the planned chain before execution.

B.4 Composition patterns

Resource progression chains connect acquisition to increasingly demanding use. For example, the ore progression chain moves through wood preparation, basic tools, stone tools, ore collection, smelting, and higher-tier mining. Cross-domain chains join heterogeneous endpoints: locating coal, obtaining fuel, creating illumination, building shelter, and remaining alive, or preparing a weapon before a combat event. Transfer chains substitute wood species, tool recipes, or item roles while retaining comparable dependency patterns.

The task suite also includes short probes. These isolate resource collection or limited crafting sequences, helping distinguish an early execution bottleneck from a long-horizon planning requirement. A short probe and a long chain answer different questions, even when they contain the same task identifier. Their sequence, start state, budget, and profile policy accompany their scores.

A completed possession task establishes its predicate at that stage. If the same item remains in inventory, a later occurrence with the same predicate may pass immediately. The ordered stage

record makes this behavior explicit: initial-state passes have zero action steps, while newly acquired items appear in the observed state transition. Composition is consequently described through stage outcomes and state continuity, with experience transfer assessed by the reset protocol in Appendix D.

B.5 Extending the catalog

A new task can be added by defining its instruction and measurable endpoint, identifying the required observation channels, selecting an environment profile, and attaching its graph relations. A new chain then specifies an ordered list of existing or new task identifiers. Compilation checks the declared dependency order, and an execution check verifies that the configured environment exposes the evidence required by its predicates.

The reusable components are the task contract, graph relations, profile registry, and runner interface. A method's planner, retrieval rule, or memory format remains outside the task definition. This allows a future agent to implement a different experience representation while being evaluated against the same target objectives and evidence rules.

C SUCCESS VERIFICATION AND EXECUTION METRICS

C.1 PREDICATE-BASED TASK EVALUATION

MC-EvoBench evaluates success from environment observations. A task contains a conjunction of predicates, with optional disjunctive subexpressions. Inventory thresholds check resource or item possession; equipment predicates accept an item in a specified slot; counters capture events such as an entity defeat; snapshot comparisons read a named observation field. These primitives cover both simple item goals and objectives that combine player state with local environmental conditions.

Table C1. Supported predicate families and their measurement roles.

Predicate family	**Accepted evidence**	**Example**
Inventory threshold	Named item count at or above a threshold	At least four oak planks
Equipment equality	Normalized item in the specified equipment slot	Stone sword in the main hand
Counter threshold	Observed event count	At least one zombie defeated
Player-stat threshold	Named player measurement	Positive life or health
Snapshot comparison	Named path, comparison operator, and value	Observed coal-ore indicator at least one
Flag membership	Flag produced by the configured event logic	A mission state supported by its observation rule
Disjunction	At least one successful child predicate	Item in inventory or equipped

A snapshot-path comparison fails when its required path is absent. Inventory and counter thresholds use zero for a missing count. The diagnostic record stores the predicate and its observed value, making the distinction between a satisfied condition and unavailable supporting evidence inspectable. Item possession is measured directly from environment observations.

The evaluator checks the initial snapshot before requesting an action. If all predicates already hold, the stage passes with zero steps. Otherwise, it checks after each environment transition until success, a mission failure, environment termination, or budget exhaustion. A stage limit is the smaller of its catalog budget and any configured cap. The normal chain runner stops at the first non-passing stage, preserving the distinction between the observed prefix and unattempted suffix.

C.2 MISSION EVENTS

Four mission-event definitions extend the basic predicates with observation-derived conditions. Their triggers combine local perception, lighting, elapsed time, and entity events. Event evaluation can attach a normalized flag or counter to the snapshot, after which the ordinary task verifier reads the corresponding evidence. This approach keeps event semantics shared between the runner and the task specification.

Table C2. Mission-event rules. Thresholds are taken from the catalog. The shelter rule is a local structure-and-light criterion; the night rule measures a defined survival interval.

Event	**Trigger**	**Endpoint**
Coal source observed	Coal ore appears in the ray channel or local block grid	Observed resource discovery
Lit shelter observed	At least one local torch; at least 60 non-air blocks; effective light at least 7; sky light at most 7 or sky blocked	Local sheltered and lit condition
Night interval survived	Player alive; elapsed time at least 1,800 ticks; final time in the night window from 12,000 to 24,000	Survival over the configured interval
Zombie defeat observed	Normalized zombie counter or raw zombie-kill count at least one	Observed combat event

For `survive_first_night`, the runner also derives `night_survived` from player survival and elapsed time reaching the effective stage limit. This additional path uses the total-time difference when available; its world-time fallback also requires a night endpoint. Thus the effective interval follows a configured stage cap, while Table C2 specifies the catalog event rule. Survival, shelter, and combat events include a player-death failure condition. Event records preserve the evidence that caused a terminal decision, so later analysis can connect a stage outcome to the underlying world measurement.

C.3 Stage records and evidence organization

Each stage record identifies the task occurrence, terminal status, action-step count, and failed predicates. Its diagnostics include start and final snapshots, declared observation sources, actual predicate values, mission evidence, and the compiled graph paths. The chain report retains the planned stage count and planned graph weight even when execution stops early.

Table C3. Terminal statuses and their interpretation in chain aggregation.

Status	**Interpretation**	**Contribution to passed-stage numerator**
`passed`	All success predicates hold	One
`failed`	A configured mission failure is observed	Zero
`terminated`	The environment ends before success	Zero
`timeout`	The effective stage limit is exhausted	Zero
`budget_exhausted`	A smaller remaining chain/source budget is exhausted	Zero
Unattempted suffix	Earlier stopping prevents execution	Zero, with missing execution recorded separately

This record supports analysis at several resolutions. A chain score summarizes completed requirements; stage records locate its execution breakpoint; start and final states describe resource changes; and optional action traces support a finer behavioral account. These resolutions share the same task identity and sequence position.

C.4 Task success rate and weighted graph progress

For chain c, let L_c be its number of planned stage occurrences and y_{cj} indicate whether occurrence j passes. Task success rate is

$$\mathrm{TSR} = \frac{1}{L_c} \sum_{j=1}^{L_c} y_{cj}. \tag{C.1}$$

In Eq. C.1, unattempted stages remain in the planned denominator. At split level, the aggregate pools passed occurrences and planned occurrences across the included chains. This weights every planned occurrence equally, while completed-chain counts offer a complementary measure of end-to-end execution.

For chain c, let $E_{\mathrm{exec},c}$ be its strict execution edges, formed by the `requires`, `produces`, and `consumes` relations. Let w_e be the edge weight and q_e indicate whether the corresponding verified stage contributes. Edge weights default to one when unspecified. Weighted graph progress is

$$\mathrm{WGP} = \frac{\sum_{e \in E_{\mathrm{exec},c}} w_e q_e}{\sum_{e \in E_{\mathrm{exec},c}} w_e}. \tag{C.2}$$

In Eq. C.2, an edge is credited when its corresponding verified stage contributes. Shared resources can appear in several stage paths, and each occurrence contributes its local edge weight. Split WGP pools completed and planned edge weights across chains. A chain with no planned execution-edge weight falls back to TSR. Together, TSR and WGP summarize stage completion and its execution structure using fixed planned denominators.

C.5 MILESTONE AND EFFICIENCY DIAGNOSTICS

Verified milestone progress (VMP) describes observed acquisition of graph-linked items or tools. For each reported stage, the metric extracts item/tool states from its execution paths and assigns each state the largest included edge-count threshold. A state is eligible if the first recorded snapshot is below its threshold; it is credited if a later recorded stage-start or stage-final snapshot reaches the threshold. The score is

$$\mathrm{VMP} = \frac{\text{eligible states reaching their thresholds}}{\text{eligible states}}, \tag{C.3}$$

Equation C.3 returns zero when the eligible set is empty. The associated eligible count is part of its interpretation. Because this set follows the reported prefix, VMP is used as a local acquisition diagnostic alongside the fixed-denominator TSR and WGP endpoints. A task passes when its success predicates hold.

Efficiency summaries include environment steps, stage budgets, steps per passed stage, and observed deaths. Resource-use diagnostics identify consumed items whose inventory decreases during a non-passing stage. This provides a traceable signal of material expenditure associated with unsuccessful execution. Recovery events require an explicit adapter record and can then be aligned with stage transitions.

C.6 VERIFICATION EXAMPLES

The task `craft_planks` requires at least four oak planks. The objective is therefore satisfied by the corresponding inventory count. For `craft_stone_sword`, the catalog accepts a stone sword either in inventory or in the main hand, matching the game's distinction between carrying and equipping a tool. The coal-discovery task uses observed coal in a ray or grid channel; the subsequent coal-collection task checks the collected item.

These examples illustrate why the task contract separates an instruction, its observation sources, and its terminal predicate. The instruction gives the agent a goal, the observation contract defines the evidence channels, and the predicate fixes the endpoint. The same decomposition supports independently checking an item goal, a temporal objective, or a cross-domain chain through explicit predicates.

D EVALUATION PROTOCOL

D.1 EXPERIENCE, ARTIFACT, AND TARGET EVALUATION

MC-EvoBench separates capability measurement from the experience that may change an agent. A paired evaluation first measures a target with the base agent, then measures the same target with an artifact produced through a specified source curriculum. The artifact may contain updated parameters, memory, a skill library, or another declared persistent representation. Its identity and provenance define the agent state being evaluated. This interface accommodates different adaptation mechanisms while keeping the target task and its verifier fixed.

At the source-to-target boundary, the target world and inventory are reset. Only the frozen method-produced artifact crosses this boundary. Consequently, tools, blocks, completed constructions, and resource inventories accumulated during source interaction do not supply target progress. In the supplied memory configuration, source interaction uses `read_write` memory and target evaluation uses `read_only` memory. Each target role constructs a fresh agent and environment; persistent source experience is loaded from its declared artifact.

Within a compiled chain, stages execute in order and preserve the environment state permitted by the chain profile. The runner stops at the first non-passing stage. The transfer probes use one chain-level profile throughout an episode. Source episodes can restart to complete their interaction budget, as described in D.3. These distinct boundaries preserve sequential task dependencies within an episode and isolate learned experience across evaluations.



Figure D1. Each source condition produces its own artifact from the same base agent. Target evaluations share task, world, budget, and seed settings. At the boundary, the world, inventory, and transient agent state are reset while the declared persistent artifact is retained.

D.2 PAIRED TARGET ROLES AND CONTROLS

Each probe and replicate defines three source roles and four target roles. All four target roles use the same target task sequence, verifier, environment profile, world seed, stage limits, and agent seed. They differ in the artifact supplied to the agent.

Table D1. Target roles and the question addressed by each source condition.

Target role	**Artifact supplied**	**Comparison supported**
No prior	Base artifact without the evaluated source experience	Initial target capability
Related	Artifact after graph-related source tasks	Transfer from relevant experience
Unrelated	Artifact after unrelated source tasks	Effect of generic extra experience
Shuffled	Artifact after the related task multiset in reversed order	Sensitivity to source ordering

The unrelated and shuffled source conditions receive the same family-level environment-step budget as the related condition. The shuffled source uses the identical task multiset, including repeated tasks. Its probe explicitly allows compilation with unmet prerequisites, making the reversed sequence executable without adding inventory or changing success predicates. Normal runtime stopping rules still apply, so the resulting experience can include repeated unsuccessful episodes.

The target input contract provides the natural-language instruction and agent observation, together with the declared artifact. It reserves graph-relation manifests, verifier predicates, verifier diagnostics, and hidden resource coordinates for evaluation. The built-in method inputs are detailed in Appendix E. Public resource placement defines the controlled task instance. It does not supply target inventory items, forced success evidence, private route hints, or scripted actions.

D.3 Frozen transfer probes and source exposure

The protocol provides three shorter graph-chain families assembled from catalog tasks. Shorter targets make transfer at intermediate capabilities observable even when a long progression chain stops at an early prerequisite. The plan freezes task sequences, resource profiles, source–target relations, and budgets before adaptive paired evaluation.

Table D2. Transfer probe families. Source exposure is matched within each row; the target sequence is held fixed across artifact conditions.

Probe family	**Related source → target variation**	**Stages per sequence**	**Source steps per condition**
Wood species	Birch logs and planks → acacia logs and planks	2	900
Wooden tools	Wood preparation and wooden axe → wooden shovel	5	2,250
Stone tools	Wood/stone preparation and stone pickaxe → stone shovel	8	3,600

The wooden-tool pair shares the log, plank, stick, and crafting-table preparation sequence. The stone-tool pair additionally shares wooden-pickaxe preparation and two cobblestone-mining stages. Unrelated source sequences contain the same number of stages and use collection tasks outside the selected production chain. The resource profiles provide local world resources while retaining the need to acquire and transform them through agent interaction.

The probe configuration imposes a 450-step ceiling on each stage, retaining a lower catalog limit if one exists. The fixed-exposure source runner is enabled by `experiment.source_interaction_budget_steps`. Source exposure is measured as actual environment interactions. If a source episode ends before its family budget is consumed, the agent receives normal terminal feedback, the source environment resets, and interaction resumes with the persistent method artifact. The last episode is truncated at the remaining budget. Thus early completion or failure changes the number of episodes without reducing the prescribed source exposure. The source report records every episode and the total interaction count; a final paired comparison requires all three source conditions to consume the common budget.

D.4 Replicates and reproducibility

The headline paired protocol uses five final agent seeds, 5100–5104. Separate calibration checks may use at least three direct seeds, but they are not part of the 105 role configurations or the headline results. The replicate axis is agent randomness on a fixed controlled world: the public world seed is 31. Fixing the world preserves the declared spawn placement and local resource layout across conditions. The resulting paired estimates characterize variation across agent seeds on these controlled instances.

Run labels such as `seed0` identify records; numerical `world_seed` and `agent_seed` fields determine randomization. The runner applies the world seed to the environment and seeds Python, NumPy, and PyTorch before constructing the policy. Reports retain both seeds and their provenance. Every role in a paired replicate uses the same numerical seeds, and the same replicate list is used across conditions.

Before rollout, the launcher checks the declared resource profile through an environment reset. Artifact manifests record identities and paths, with SHA-256 fingerprints available for files and directory bundles. Together, the frozen plan, explicit seeds, artifact identity, and runtime configuration specify a reproducible comparison.

D.5 TRANSFER CONTRASTS

TSR, WGP, and VMP follow the definitions in Appendix C. Let M_0, M_R, M_U, M_S denote a target score with no prior, related, unrelated, and shuffled artifacts, respectively. Define each source condition's change from the no-prior target as

$$\Delta_c = M_c - M_0, \qquad c \in \{R, U, S\}. \tag{D.1}$$

The paired differences in Eq. D.1 are interpreted across the three direct metrics. A related-source change is compared with both the no-prior baseline and the matched unrelated and shuffled controls; these are contrasts over TSR, WGP, and VMP rather than additional benchmark metrics. Target TSR is the primary channel, while WGP and VMP provide complementary completion and acquisition diagnostics.

D.6 STATISTICAL AND DIAGNOSTIC REPORTING

Compute contrasts within each matched replicate before summarizing across seeds. Report the four target scores and paired differences for each direct metric, with their means and sample standard deviations. The aggregation helper also computes two-sided 95% Student-t intervals on paired seed contrasts:

$$\bar{d} \pm t_{0.975,n-1} \frac{s_d}{\sqrt{n}}. \tag{D.2}$$

For the interval in Eq. D.2, n is the number of valid pairs and s_d is the sample standard deviation of paired differences. A beneficial-transfer interpretation requires a positive related-source change relative to the no-prior baseline and matched controls, supported by paired uncertainty. Metric configurations are also characterized as floor-limited, ceiling-limited, or interior. The accompanying progress channels localize capability acquisition across these score regimes.

Report environment steps alongside source episode counts, outcomes, memory updates, planner calls, and token usage. Equal environment exposure can produce different amounts of planning and updating because episodes terminate at different points. These diagnostics characterize the resources used to create each artifact and support computation-aware comparisons. Per-seed target scores and source exposure remain available alongside aggregate statistics, allowing transfer and efficiency to be assessed together.

D.7 PER-SEED TRANSFER RECORDS

Table D3 lists the per-seed target TSR values behind Figure 3b–c and Table 4, computed within each matched replicate as described in Appendix D.6.

Table D3. Per-seed target TSR under the four target conditions: no prior (N), related (R), unrelated (U), and shuffled (S). $\Delta R = R - N$ and $\mathrm{GATG} = R - \frac{1}{2}(U + S)$ are computed within each seed; values are rounded to three decimals.

Probe family	**Seed**	**N**	**R**	**U**	**S**	$\boldsymbol{\Delta R}$	**GATG**
Wood species	`seed0`	0.000	0.000	0.000	1.000	0.000	−0.500
	`seed1`	0.500	1.000	0.000	0.000	0.500	1.000
	`seed2`	0.500	0.000	0.000	1.000	−0.500	−0.500
	`seed3`	1.000	0.000	0.000	0.000	−1.000	0.000
	`seed4`	0.000	1.000	1.000	0.500	1.000	0.250
Wooden tool	`seed0`	0.200	0.200	0.200	0.200	0.000	0.000
	`seed1`	0.200	0.000	0.000	0.000	−0.200	0.000
	`seed2`	0.800	0.400	0.400	0.200	−0.400	0.100
	`seed3`	0.200	0.400	0.200	0.400	0.200	0.100
	`seed4`	0.200	0.000	0.000	0.400	−0.200	−0.200
Stone tool	`seed0`	0.125	0.125	0.000	0.000	0.000	0.125
	`seed1`	0.125	0.000	0.000	0.125	−0.125	−0.063
	`seed2`	0.125	0.000	0.000	0.000	−0.125	0.000
	`seed3`	0.125	0.125	0.125	0.125	0.000	0.000
	`seed4`	0.250	0.000	0.000	0.125	−0.250	−0.063

E AGENT IMPLEMENTATIONS

E.1 OBSERVATION AND ACTION INTERFACES

MC-EvoBench connects policies to a common chain runner through a policy factory, a reset operation, and an action function. At each interaction, the adapter supplies the task instruction and current observation and receives a MineRL action mapping. The default prompt template is the instruction itself. Optional callbacks provide stage status and the number of executed steps, allowing methods with persistent memory to update after source experience. Table E1 specifies the implemented method variants and their effective inputs; these interfaces define the observation, action, and persistent-state conditions used in each comparison. The table documents supported adapters; the headline reference study reported in the main paper uses only the JARVIS-1 online extension, while the other entries do not contribute to the paired-transfer results.

Table E1. Implemented agent interfaces. "Persistent state" identifies information retained by the bridge for later decisions.

Variant	Effective decision input	Action and persistent state
VPT	Raw MineRL observation; instruction unused	Released visual policy; recurrent controller state
STEVE-1 / JARVIS-1 controller	Observation and text embedding	Released STEVE-I actions; cached prompt embeddings
JARVIS online extension	Planner: inventory, equipment, position, retrieved episodes; controller: observation and subgoal	Local text planner with released controller; episode memory
ExpeL adaptation	Inventory, equipment, position/orientation, nearby blocks, line of sight, episodes and insights	LLM-generated action mapping; episodes, insights and weighted rules
Optimus-1 action bridge	Screenshot file and instruction	Released action model; direct controller execution
Optimus-1 planner bridge	Planner: screenshot and instruction; controller: screenshot and subgoal	Local visual planner with released action model; diagnostic trace
Optimus-3 action path	Image and task-conditioned embedding	Released MLLM, task router and action head
Voyager skill bridge	Fixed task-to-skill sequence and Mineflayer state	Released JavaScript skills and primitives; skill files

The STEVE-1 and JARVIS controller bridges encode text with MineCLIP and a learned prior, cache embeddings by prompt, and reset the conditional controller with guidance scale 6.0. VPT loads its network specification and policy-head settings from the model file; its sampling temperature is inherited from that file. Optimus-3 creates a task representation when the instruction changes, then combines it with successive frames.

E.2 PERSISTENT EXPERIENCE AND ADAPTATION

The JARVIS online extension stores stage-level textual records containing the instruction, proposed plan, dispatched controller prompts, status, interaction count and visible-state summary. Retrieval uses token-set Jaccard similarity between the current instruction and each stored instruction, adds 0.15 for a successful record, and breaks score ties in favour of newer records. Up to four positive-scoring records condition planning. Memory retains the latest 256 records and is written atomically. The planner returns one to six short controller prompts; reflection produces a practical sentence capped at 500 characters. The reflection input distinguishes a proposed plan from prompts actually dispatched and labels the state summary as the last pre-action observation.

ExpeL uses the same episode-retrieval score and supplements episodes with retrieved textual insights and a weighted rule list. Insight retrieval uses lexical overlap; rule retrieval adds a weight contribution capped at 0.10. With the configured insight limit of eight, the planner may receive up to eight insights and eight rules. Reflection proposes ADD, EDIT, REMOVE and AGREE operations. New rules start with weight two; agreement and editing increment weight, while removal decrements it. The persistent artifact records these updates alongside episode actions. Both mem-

ory methods support read-write source interaction and read-only target evaluation; read-only stage callbacks suppress reflection and memory writes.

E.3 Planning and execution cadence

The JARVIS extension plans when a new stage instruction becomes active. It dispatches successive subgoals for fixed 120-step intervals and retains the final subgoal for any remaining stage steps. Stage completion invalidates the active instruction so repeated task text can trigger a new plan. Optimus-1's visual planner uses the same interval rule and a JSON subgoal format. Its input is a JPEG thumbnail bounded by 384 $\times$ 384 pixels at quality 85. The parser filters subgoals containing "open," "place," or "access" followed by a space, rewrites punch/collect/gather to chop, and retains up to six subgoals. ExpeL instead requests a low-level action mapping and can hold that action across a configured number of environment steps. The 19-chain configuration sets this hold to 32, while the factory default is one. Its prompt also supplies explicit line-of-sight guidance for approaching and mining visible resources.

The Voyager skill bridge follows a separate execution protocol. The runner loads released JavaScript skills and control primitives, resets through the Mineflayer bridge with empty inventory and equipment, and invokes a fixed skill sequence. A configured starting position is applied during setup. The runner checks the returned inventory for its designated target item. Its step count denotes a bridge request, and the implementation invokes no LLM; this execution unit is recorded separately from MineRL interaction steps.

E.4 Configured parameters and replication

Table E2 reports configuration values and the defaults used where configurations omit a field. Planner model identifiers are recorded literally because the standard and relay matrices select different providers. Planner calls leave temperature unspecified. Token limits bound generated output, and timeout or retry settings describe request handling rather than additional environment interaction.

Table E2. Configuration values and implementation defaults. J = JARVIS online extension; O = Optimus-1 planner bridge; E = ExpeL adaptation.

Component	**Configured value or applicable default**
Text-conditioned controllers	VPT `2x.model`; STEVE-1 weights and prior; guidance 6.0 for STEVE-1/JARVIS
J standard / relay planner	`gpt-5.4` / `gpt-5.6-sol`; relay uses Responses with high reasoning effort
O / E planner	`~openai/gpt-luna-latest` / `gpt-5.6-sol`; E uses Responses, high effort
Output limit; timeout	J: 4,096 tokens, 60 s; O: 4,096, 90 s; E: 2,048, 60 s
Memory and cadence	256 records; 4 retrieved episodes; J/O horizon 120; E insight/rule limits 8 each, action hold 32

The standard and relay configurations instantiate the same benchmark roles through different planner interfaces. A reproducible method specification records the selected matrix, policy checkpoint, observation channels, action cadence, planner identity, and memory mode. Replicate seeds and source budgets follow Appendix D. These fields allow the effect of the experience mechanism to be interpreted alongside its controller and computational settings.

F Complete Benchmark Suite

F.1 Task inventory

Tables F1–F3 enumerate the 80 task definitions once. The code T01–T80 is a compact cross-reference used only in this appendix; the adjacent identifier is the executable catalog key. "Steps" is the catalog stage budget, before an experiment applies a smaller cap. Profile codes refer to Table A2. Success conditions specify the required observed endpoint; event abbreviations refer to the exact rules in Appendix C.

The supplementary CSV preserves each original natural-language instruction and the full JSON success specification. This keeps the printed table compact while providing an exact machine-readable mapping between the manuscript, task identifiers, and evaluator predicates. In a chain, each occurrence retains its task's definition and receives the profile selected by the chain policy.

Table F1. Core Progression: 33 task definitions.

Code / catalog task	Success condition	Steps	Profile
T01 · `scout_coal_source`	Coal visible in ray or local grid (C.2)	500	P05
T02 · `collect_log`	oak log $\geq$1	400	P02
T03 · `craft_planks`	oak planks $\geq$4	600	P02
T04 · `craft_table`	crafting table $\geq$1	700	P02
T05 · `craft_wooden_pickaxe`	wooden pickaxe $\geq$1	1000	P02
T06 · `collect_coal`	coal $\geq$1	700	P05
T07 · `craft_torch`	torch $\geq$1	220	P05
T08 · `build_lit_shelter`	Local structure and lighting rule (C.2)	1200	P04
T09 · `survive_first_night`	Night-survival event (C.2)	1800	P03
T10 · `mine_cobblestone`	cobblestone $\geq$2	700	P02
T11 · `craft_stone_sword`	Stone sword in inventory $\geq$1 or in main hand	280	P02
T12 · `defeat_zombie`	Zombie-defeat counter $\geq$1	1200	P03
T13 · `craft_sticks`	stick $\geq$4	300	P02
T14 · `craft_furnace`	furnace $\geq$1	700	P02
T15 · `collect_iron_ore_basic`	iron ore $\geq$1	1200	P12
T16 · `smelt_iron_ingot`	iron ingot $\geq$1	1200	P12
T17 · `collect_string`	string $\geq$1	900	P02
T18 · `collect_iron_ore`	iron ore $\geq$1	1200	P12
T19 · `collect_gold_ore`	gold ore $\geq$1	1400	P12
T20 · `collect_redstone`	redstone $\geq$1	1500	P12
T21 · `collect_lapis_lazuli`	lapis lazuli $\geq$1	1500	P12
T22 · `collect_diamond`	diamond $\geq$1	2000	P12
T23 · `craft_stone_pickaxe`	stone pickaxe $\geq$1	800	P02
T24 · `smelt_charcoal`	charcoal $\geq$1	900	P02
T25 · `smelt_glass`	glass $\geq$1	900	P02
T26 · `smelt_gold_ingot`	gold ingot $\geq$1	1000	P12
T27 · `craft_iron_pickaxe`	iron pickaxe $\geq$1	1200	P12
T28 · `collect_white_wool`	white wool $\geq$1	700	P02
T29 · `craft_white_bed`	white bed $\geq$1	800	P02
T30 · `craft_bow`	bow $\geq$1	800	P02
T31 · `craft_shield`	shield $\geq$1	1000	P02
T32 · `craft_bucket`	bucket $\geq$1	1000	P02
T33 · `collect_water_bucket`	water bucket $\geq$1	700	P02

Table F2. Transfer and Schema Generalization: 31 task definitions.

Code / catalog task	Success condition	Steps	Profile
T34 · `craft_chest`	chest $\geq$1	500	P02

Code / catalog task	Success condition	Steps	Profile
T35 · craft_oak_boat	oak boat ≥1	600	P02
T36 · craft_wooden_axe	wooden axe ≥1	800	P02
T37 · craft_wooden_sword	wooden sword ≥1	600	P02
T38 · craft_wooden_shovel	wooden shovel ≥1	600	P02
T39 · collect_birch_log	birch log ≥1	450	P02
T40 · craft_birch_planks	birch planks ≥4	600	P02
T41 · craft_birch_boat	birch boat ≥1	700	P02
T42 · collect_spruce_log	spruce log ≥1	450	P02
T43 · craft_spruce_planks	spruce planks ≥4	600	P02
T44 · craft_spruce_boat	spruce boat ≥1	700	P02
T45 · collect_jungle_log	jungle log ≥1	450	P02
T46 · craft_jungle_planks	jungle planks ≥4	600	P02
T47 · craft_jungle_boat	jungle boat ≥1	700	P02
T48 · collect_acacia_log	acacia log ≥1	450	P02
T49 · craft_acacia_planks	acacia planks ≥4	600	P02
T50 · craft_acacia_boat	acacia boat ≥1	700	P02
T51 · collect_dark_oak_log	dark oak log ≥1	450	P02
T52 · craft_dark_oak_planks	dark oak planks ≥4	600	P02
T53 · craft_dark_oak_boat	dark oak boat ≥1	700	P02
T54 · craft_bowl	bowl ≥1	400	P02
T55 · craft_ladder	ladder ≥1	500	P02
T56 · craft_oak_sign	oak sign ≥1	600	P02
T57 · craft_oak_fence	oak fence ≥1	600	P02
T58 · craft_oak_door	oak door ≥1	600	P02
T59 · craft_oak_trapdoor	oak trapdoor ≥1	600	P02
T60 · craft_oak_pressure_plate	oak pressure plate ≥1	500	P02
T61 · craft_stone_axe	stone axe ≥1	800	P02
T62 · craft_stone_shovel	stone shovel ≥1	700	P02
T63 · craft_stone_hoe	stone hoe ≥1	700	P02
T64 · smelt_stone	stone ≥1	900	P02

Table F3. Supplementary Diversity and Exploration: 16 task definitions.

Code / catalog task	Success condition	Steps	Profile
T65 · collect_dirt	dirt ≥1	200	P02
T66 · collect_sand	sand ≥1	300	P02
T67 · collect_gravel	gravel ≥1	350	P02
T68 · collect_flint	flint ≥1	600	P02
T69 · collect_seeds	wheat seeds ≥1	300	P02
T70 · collect_flower	poppy ≥1	300	P02
T71 · collect_clay_ball	clay ball ≥1	400	P02
T72 · collect_cactus	cactus ≥1	500	P02
T73 · collect_sugar_cane	sugar cane ≥1	500	P02
T74 · collect_apple	apple ≥1	700	P02
T75 · collect_pumpkin	pumpkin ≥1	700	P02
T76 · collect_melon_slice	melon slice ≥1	800	P02
T77 · collect_red_mushroom	red mushroom ≥1	700	P02
T78 · collect_brown_mushroom	brown mushroom ≥1	700	P02
T79 · collect_bone	bone ≥1	900	P02
T80 · collect_emerald	emerald ≥1	2200	P12

F.2 ORDERED CHAIN REGISTRY

A sequence below gives every task occurrence, including repetitions. Task codes expand through Tables F1–F3. Each chain begins with the symbolic state `environment:spawned_world`; no physical resource is created by that symbolic entry. The `night_combat_probe` chain additionally declares symbolic initial states `structure:lit_shelter` and `tool:stone_sword`; these declarations do not inject physical resources. "Task" selects the stage's task profile, whereas "chain" retains the listed chain profile. The runtime state-continuity rule is defined in Appendix A.4.

Table F4. Core Progression: exact chain sequences.

Chain / profile / policy	Length	Ordered task codes
`cross_domain_safe_night`; P01; task	10	T01, T02, T03, T13, T04, T05, T06, T07, T08, T09
`combat_readiness`; P01; task	8	T02, T03, T13, T04, T05, T10, T11, T12
`ore_mining_progression`; P12; chain	19	T02, T03, T13, T04, T05, T10, T23, T14, T01, T06, T15, T16, T27, T18, T19, T20, T21, T22, T80
`smelting_transformations`; P12; chain	21	T02, T03, T13, T04, T05, T10, T14, T01, T06, T66, T02, T24, T10, T64, T25, T15, T16, T27, T19, T06, T26
`full_mixed_progression`; P01; task	13	T01, T02, T03, T13, T04, T05, T06, T07, T08, T09, T10, T11, T12

Table F5. Transfer and Schema Generalization: exact chain sequences.

Chain / profile / policy	Length	Ordered task codes
`wood_species_crafting`; P02; task	15	T39, T40, T41, T42, T43, T44, T45, T46, T47, T48, T49, T50, T51, T52, T53
`oak_utility_crafting`; P02; task	11	T02, T03, T13, T04, T54, T55, T56, T57, T58, T59, T60
`stone_tool_suite`; P02; task	10	T02, T03, T13, T04, T05, T10, T23, T61, T62, T63
`wooden_utility_tool_suite`; P02; task	14	T02, T03, T34, T02, T03, T35, T02, T03, T13, T04, T36, T02, T03, T38

Table F6. Supplementary Diversity and Exploration: exact chain sequences.

Chain / profile / policy	Length	Ordered task codes
`basic_resource_sampler`; P02; task	6	T65, T66, T67, T68, T69, T70
`wood_species_collection`; P02; task	5	T39, T42, T45, T48, T51
`natural_material_sampler`; P02; task	8	T71, T72, T73, T74, T75, T76, T77, T78

Table F7. Calibration and Probe: exact chain sequences.

Chain / profile / policy	Length	Ordered task codes
`collect_log_smoke`; P02; task	1	T02
`wooden_tools_smoke`; P02; task	5	T02, T03, T13, T04, T05
`coal_smoke`; P05; chain	7	T02, T03, T13, T04, T05, T01, T06
`wood_crafting_basics`; P02; task	5	T02, T03, T13, T04, T05
`stone_crafting_basics`; P02; task	7	T02, T03, T13, T04, T05, T10, T14
`survival_utility_suite`; P01; task	24	T02, T03, T13, T04, T37, T17, T30, T79, T28, T02, T03, T29, T05, T10, T14, T01, T06, T15, T16, T02, T03, T31, T32, T33
`night_combat_probe`; P03; task	2	T09, T12

F.3 DETAILED DIRECT CAPABILITY RESULTS

The direct capability matrix in Table 2 reports partition-level averages. Table F8 expands those values to the 12 primary chains and the nine direct configurations. Each chain occupies three rows that report TSR, WGP, and VMP. The seven calibration and probe chains are retained in the complete chain registry but are not included in this expansion. Table F9 fixes the protocol under which these direct results were collected.

Table F8. Per-chain direct capability results for the 12 primary evaluation chains underlying Table 2. Each chain occupies three rows reporting TSR, WGP, and VMP in percent. The two planner-backed columns report GPT-5.6-sol runs. The nine scored direct configurations exclude the separately marked ExpeL runs. A superscript 2 marks a two-seed report.

Chain	Metric	STEVE-1 controller	JARVIS-1 controller	Optimus-1 controller	Optimus-3 action path	VPT STEVE-1 ckpt.	VPT official 2x foundation	no-op baseline	JARVIS-1 planner	Optimus-1 planner
Core progression										
cross_domain safe_night	TSR	30.0	16.7	26.7	10.0	13.3	10.0[2]	10.0	10.0	23.3
	WGP	21.7	10.1	17.4	4.3	5.8	4.3	4.3	4.3	15.9
	VMP	66.7	22.2	61.1	0.0	16.7	0.0	0.0	0.0	41.7
combat readiness	TSR	12.5	16.7	12.5	0.0	0.0	4.2	0.0	0.0	0.0
	WGP	8.1	9.8	8.1	0.0	0.0	1.6	0.0	0.0	0.0
	VMP	38.9	55.6	38.9	0.0	0.0	16.7	0.0	0.0	0.0
full mixed_progression	TSR	20.5	12.8	23.1	7.7	7.7[2]	10.3	7.7	7.7	10.3
	WGP	13.8	5.7	17.2	3.4	3.4	4.6	3.4	3.4	4.6
	VMP	58.3	33.3	66.7	0.0	0.0	16.7	0.0	0.0	16.7
ore mining_progression	TSR	0.0	3.5	0.0	0.0	0.0	0.0[2]	0.0[2]	0.0	0.0
	WGP	0.0	2.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	VMP	0.0	22.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
smelting transformations	TSR	1.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	WGP	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	VMP	16.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Catalog transfer										
wood_species crafting	TSR	0.0	4.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	WGP	0.0	4.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	VMP	0.0	22.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
oak_utility crafting	TSR	6.1	6.1	6.1	0.0	6.1	0.0	0.0	0.0	0.0
	WGP	7.6	7.6	7.6	0.0	7.6	0.0	0.0	0.0	0.0
	VMP	22.2	22.2	22.2	0.0	22.2	0.0	0.0	0.0	0.0
stone_tool suite	TSR	16.7	10.0	13.3	0.0	0.0	6.7	0.0	0.0	6.7
	WGP	14.4	6.5	7.8	0.0	0.0	2.6	0.0	0.0	5.2
	VMP	47.2	38.9	55.6	0.0	0.0	33.3	0.0	0.0	22.2
wooden_utility tool_suite	TSR	7.1	14.3	9.5	0.0	0.0	2.4	0.0	0.0	0.0[2]
	WGP	5.6	13.6	9.0	0.0	0.0	1.1	0.0	0.0	0.0
	VMP	38.9	66.7	44.4	0.0	0.0	16.7	0.0	0.0	0.0
Diversity										
basic resource_sampler	TSR	5.6	5.6	5.6	0.0	0.0	5.6	0.0	0.0	0.0
	WGP	5.6	5.6	5.6	0.0	0.0	5.6	0.0	0.0	0.0
	VMP	16.7	16.7	16.7	0.0	0.0	16.7	0.0	0.0	0.0
wood species_collection	TSR	0.0	0.0	0.0	0.0	0.0	10.0[2]	0.0	0.0	0.0
	WGP	0.0	0.0	0.0	0.0	0.0	10.0	0.0	0.0	0.0
	VMP	0.0	0.0	0.0	0.0	0.0	25.0	0.0	0.0	0.0
natural material_sampler	TSR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0[2]	0.0
	WGP	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	VMP	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table F9. Direct capability protocol. These runs measure immediate capability and do not define a source–target transfer relation.

Item	Fixed setting
Protocol identifier	`mcbench_full19_direct_matrix_v1`
Chains	All 19 registered chains are run; the 12 primary chains (5 core progression, 4 catalog transfer, 3 diversity) are scored, and the 7 calibration and probe chains are reported separately
Scale	Nine direct configurations × three seeds × 19 chains: 513 planned units, 505 valid reports; units without a valid report leave some cells with two seeds
Seeds	World seeds 3101, 3102, 3103; agent seeds 4101, 4102, 4103
Environment profiles	`per_stage_profiles=true`; each stage is initialized from its published profile
Observation and action	MineRL POV observation with a unified task description and action interface; the task graph, verifier predicates, and hidden resource coordinates are not exposed
Planner-backed runs	`gpt-5.6-sol` through the Responses interface with high reasoning effort; the planner produces subgoals only, and actions come from the underlying controller
Experience state	No source–target transfer; planner-backed configurations start from an empty, read-only memory
Reported fields	Stage records, TSR, WGP, VMP, environment steps, and termination status
ExpeL	One separately marked reference report per chain, excluded from the nine-configuration aggregates

Table F10 gives the stage counts behind Figure 3a.

Table F10. Failure-stage audit over the 57 direct-matrix reports per configuration: passed / reached reports and conditional pass rate (%) at the wood and crafting stages; the full-chain column counts reports whose entire chain passed.

Configuration	Collect log	Craft planks	Craft sticks	Crafting table	Full chain
STEVE-1	24/42 (57.1)	14/21 (66.7)	2/13 (15.4)	0/2 (0.0)	3/57 (5.3)
JARVIS-1 (controller-only)	22/42 (52.4)	11/20 (55.0)	1/8 (12.5)	0/1 (0.0)	2/57 (3.5)
Optimus-1 (controller-only)	24/42 (57.1)	16/23 (69.6)	1/14 (7.1)	0/1 (0.0)	1/57 (1.8)

F.4 SOURCE AND TARGET SEQUENCES FOR PAIRED TRANSFER

Table F11 expands the three transfer probe families into their exact source and target streams. Task codes refer to Tables F1–F3. The shuffled condition reverses the related-source sequence, preserving its task multiset. The empty-artifact baseline has no source stream and uses the same target sequence.

Table F11. Frozen source/target sequences. Repeated codes denote repeated stage occurrences.

Family / condition	**Ordered task codes**
`wood_species_crafting` / Related	T39, T40
`wood_species_crafting` / Unrelated	T72, T73
`wood_species_crafting` / Shuffled	T40, T39
`wood_species_crafting` / Target	T48, T49
`wooden_tool_transfer` / Related	T02, T03, T13, T04, T36
`wooden_tool_transfer` / Unrelated	T71, T72, T73, T77, T70
`wooden_tool_transfer` / Shuffled	T36, T04, T13, T03, T02
`wooden_tool_transfer` / Target	T02, T03, T13, T04, T38
`stone_tool_transfer` / Related	T02, T03, T13, T04, T05, T10, T10, T23
`stone_tool_transfer` / Unrelated	T72, T73, T75, T76, T77, T78, T67, T71
`stone_tool_transfer` / Shuffled	T23, T10, T10, T05, T04, T13, T03, T02
`stone_tool_transfer` / Target	T02, T03, T13, T04, T05, T10, T10, T62

G PROMPTS AND REPRODUCIBILITY

G.1 PLANNING AND REFLECTION TEMPLATES

The online-memory reference implementation uses two system templates. The planning request combines the current goal, an agent-visible state summary, and retrieved experience. It returns a small sequence of text subgoals for the visual controller. The reflection request summarizes a completed source episode for later retrieval. Cards G1 and G2 specify the input–output contracts and system prompts.

Card G1. Online-memory planning

Purpose. Translate the current goal and retrieved experience into controller subgoals.
Input. `goal`, `visible_state`, `retrieved_experience`
Output. JSON `steps` with 1–6 text prompts.

```
You are a Minecraft task planner controlling a visual low-level policy. Use only the
natural-language goal, visible state, and prior experience. Do not assume access to reward
predicates, task graphs, hidden world coordinates, or tools outside Minecraft. Return JSON
only: {"steps":["short visual action prompt", ...]}. Provide 1 to 6 concise prompts
suitable for a STEVE-I-like action controller.
```

Card G2. Experience reflection

Purpose. Convert source-stage feedback into reusable advice.
Input. Goal, pre-action state, dispatched prompts, stage feedback.
Output. One practical sentence stored as `advice`.

```
You summarize one Minecraft episode for later reuse. Use only the supplied goal,
agent-visible state summary, controller dispatch evidence, and stage feedback. The plan is
proposed text, not proof that every prompt was attempted. Only dispatched_plan entries were
sent to the controller and returned actions; this does not prove their intended effects
occurred. The visible_state is the last pre-action observation, not a post-action terminal
observation. Do not invent hidden information or unobserved events. Return one concise
practical sentence.
```

The planning payload contains `goal`, `visible_state`, and `retrieved_experience`. Retrieved records expose the instruction, plan, status, and advice. The planning parser returns up to six non-empty steps, using the task instruction as its default. Reflection receives stage feedback together with the actually dispatched prompts, allowing the stored advice to distinguish planned behavior from observed execution context.

G.2 PERSISTENT MEMORY REPRESENTATION

The artifact is a JSON object containing a record count and a list of episode records. Each record associates a task instruction with a proposed plan, the prompts dispatched to the controller, terminal status, interaction count, visible state, and advice. The memory writer keeps the latest configured number of records and replaces the artifact atomically. Retrieval and update rules are specified in Appendix E; the schema below describes the fields required to inspect the stored experience.

Table G1. Persistent memory fields and their interpretation.

Field	Content	Role in reproduction
`format`	Memory schema identifier	Identifies the memory format
`record_count`	Number of retained records	Checks artifact size
`instruction`	Task text	Retrieval key
`plan`	Proposed controller prompts	Records planner output
`dispatched_plan`	Prompts actually sent to the controller	Links planning to dispatch
`status`, `steps`	Stage outcome and executed steps	Captures feedback supplied to the method
`visible_state`	Agent-side state summary	Records the reflection context
`advice`	Concise text retained for reuse	Reusable experience content

The source artifact is copied or otherwise identified before target evaluation, and its fingerprint accompanies the target role. Read-only target memory keeps the supplied experience fixed throughout evaluation. This makes the artifact itself an inspectable component of a paired comparison.

G.3 ExpeL action and rule templates

The ExpeL adaptation uses separate system templates for selecting an environment action and extracting reusable experience. Cards G3 and G4 specify the action and reflection interfaces. The action payload contains `instruction`, `observation`, `experiences`, and `insights`. The final field combines insight text with retrieved rules formatted as `[ExpeL rule weight=...] ...`. This representation exposes the rule's persistence weight alongside its content. The observation is the structured state summary supplied by the adapter.

Card G3. ExpeL action selection

Purpose. Map visible state and retrieved experience to a game action.
Input. `instruction`, `observation`, `experiences`, `insights`
Output. A JSON `action` mapping for MineRL.

```
You are the adapted ExpeL Minecraft policy for MC-EvoBench. Use only the task,
agent-visible observation, retrieved episodes, and insights. Never infer hidden verifier
predicates or coordinates. Return JSON only with an action object suitable for MineRL, e.g.
{"action": {"forward": 1}}. The observation may contain public nearby_blocks counts and
one-hot line_of_sight block indicators. If line_of_sight contains the requested resource
(for example oak_log=1), preserve the aim: when its numeric distance is at most 4, use
{"attack": 1} to mine it; when it is farther away, use {"forward": 1} without turning so
the agent can approach and collect the drop. Use small camera changes to search only when
the target is not in line of sight.
```

The action parser accepts a mapping directly or JSON text, including a fenced JSON response. It accepts either a nested `action` mapping or the mapping itself. A singleton {`"noop": true`} or {`"noop": 1`} becomes the adapter's no-op marker. The action parser validates the response mapping; task success is evaluated from environment observations.

Card G4. ExpeL rule update

Purpose. Extract reusable experience and update persistent rules.
Input. Stage observations, dispatched actions, and status.
Output. JSON `insight` and `operations`.

```
Extract one concise reusable insight from this Minecraft episode and update the persistent
ExpeL rule list. Use only supplied observations, dispatched actions, and status; do not
claim an effect unless it is explicitly observed. Return JSON only: {"insight": "...",
"operations": [{"op": "ADD|EDIT|REMOVE|AGREE", "rule_number": 1, "rule": "..."}]}. Use at
most four operations and use an empty operations list when no durable general rule is
justified.
```

Reflection receives the stage episode: `instruction`, `status`, `steps`, `actions`, `visible_state`, `visible_state_timing`, and `method`. The state is explicitly labeled as the last pre-action observation, and the action list includes each returned action, including repetitions. The reflection parser accepts `insight` or `reflection` for the insight, and `operations` or `rules` for the operation list. It normalizes the response into insight text and rule operations. The prompt requests up to four rule operations per reflection.

Table G2. ExpeL persistent artifact fields. Episode records, extracted insights, and rule-update history are stored separately.

Field	Stored content
`format`	ExpeL memory schema identifier
`record_count`, `insight_count`	Sizes of the retained episode and insight lists
`records`	Stage episodes with action and feedback fields
`insights`	Whitespace-normalized reusable text
`rules`	Rule objects containing `text` and integer `weight`
`rule_events`	Normalized `operations`, `list_full`, and the resulting `rules` snapshot

The artifact therefore preserves both the interaction record and the rule state used for subsequent decisions. The loader validates the artifact's format identifier.

G.4 Configuration example

Card G5 gives the protocol fields from the wood-species target-after-related configuration. It includes the selected environment, agent randomness, memory mode, and target sequence. The complete repository file also specifies checkpoint locations, planner settings, artifact paths, role metadata, and output locations. Deployment paths and artifact format identifiers are resolved through `reproduction/paths.json`.

Card G5. Matched target configuration

Purpose. Fix the target conditions while selecting a source artifact.
Input. A frozen source artifact and the wood-species probe.
Output. A read-only target run under the listed field values.

```
{
  "environment_profile": "evolution_wood_species_grove",
  "max_stage_steps": 450,
  "environment": {
    "kind": "minerl",
    "seed": 31,
    "per_stage_profiles": false
  },
  "agent": {
    "kind": "jarvis1_online",
    "random_seed": 5100,
    "memory_mode": "read_only",
    "memory_max_records": 256,
    "retrieval_limit": 4,
    "skill_horizon_steps": 120
  },
  "probe": {
    "profile_policy": "chain",
    "task_ids": [
      "collect_acacia_log",
      "craft_acacia_planks"
    ]
  }
}
```

The target collects acacia logs and crafts acacia planks while using the artifact generated by the related birch source. The chain profile remains constant within this episode. Stage limits are applied by the runner, and task success uses the catalog predicates. The source role uses its own task sequence and enables memory writing; all target roles retain the same task definition.

G.5 Evaluation algorithm

The following pseudocode summarizes the common orchestration contract. Source interaction, artifact persistence, environment reset, and target evaluation are separate operations. The same control flow can host parameter updates, text memory, or executable skills, provided that the permitted persistent state and input interface are declared.

Card G6. Paired experience evaluation

Purpose. Isolate the effect of related source experience.
Input. Probe family, paired seed, base agent, matched source budget.
Output. Four matched target scores and within-seed contrasts.

```
for each probe family and paired seed:
    base = load_base_agent_state()
    score[0] = evaluate_target(base, fresh_target_world())
    for condition in [related, unrelated, shuffled]:
        state = independent_copy(base)
        remaining = source_budget[family]
        while remaining > 0:
            episode = run_source_episode(state, condition, remaining)
            update_source_artifact(state, episode)
            remaining -= episode.environment_steps
        artifact = freeze_and_identify(state.persistent_state)
        target_agent = load_for_read_only_evaluation(base, artifact)
        score[condition] = evaluate_target(
            target_agent, fresh_target_world())
    save_paired_contrasts(score, source_costs, artifact_identities)
```

At the task level, evaluation first observes the initial state, then alternates action and predicate evaluation until a terminal condition is reached. At the chain level, the runner applies the selected profile policy and records the first non-passing stage. At the experiment level, matched roles are combined within a seed before aggregation. These layers align environment execution, task success, and experience comparison at their respective units of analysis.

G.6 Reproduction entry points

Table G3 identifies the small set of source components that define the benchmark. The accompanying task and chain CSV files provide compact access to the catalog, while the JSON source files retain the executable definitions.

Table G3. Source map for reproducing the definitions in this appendix.

Component	Repository location
Tasks, states, edges, chains	`catalog` in `reproduction/paths.json`
Schema, graph, compilation	`src/mcbench/schema.py`, `graph.py`, `chains.py`
Environment and profiles	`src/mcbench/adapters/minerl.py`, `profiles.py`
Predicates and stage loop	`src/mcbench/verification.py`, `runner.py`
Execution and transfer metrics	`src/mcbench/metrics.py`, `evolution.py`
Paired transfer plan and profiles	`paired_plan`, `paired_profiles` in `reproduction/paths.json`
Reference memory implementation	`src/mcbench/adapters/jarvis1_online.py`
Configuration execution	`src/mcbench/cli.py`

Catalog validation checks the task definitions and compiled chains. A dry run then exercises the reporting path with the repository's diagnostic adapter. Real execution additionally requires the selected runtime, policy checkpoints, and method-specific dependencies. The experiment configuration records these choices alongside the catalog snapshot. Card G7 runs from the appendix bundle directory and resolves the catalog through the companion path map.

Card G7. Reproduction entry points

Purpose. Validate the catalog and inspect the reporting path.
Input. The companion path map and repository source.
Output. Catalog checks and a diagnostic chain report.

```
python3 tools/reproduce.py validate
python3 tools/reproduce.py dry-run cross_domain_safe_night
```

A reproducible experiment release associates the frozen catalog, configuration, model identity, seeds, and artifact identity with each chain report. Stage evidence and source-cost records preserve the path from a task definition to a paired contrast.