# MULTIMODAL FILESYSTEM MEMORY FOR LONG-HORIZON EMBODIED AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Long-horizon agents need memory to retain and reuse experience across long interactions, but how to represent that memory remains unclear. Retaining complete interaction histories makes relevant experience difficult to find, while compressing them into summaries can discard visual and spatial details needed later. We introduce **MFM**, a multimodal filesystem memory architecture that maintains complementary forms of agent memory. **Episodic memory** preserves observations, actions, and outcomes, while **spatial memory** connects experiences to locations. **Semantic and procedural memory** accumulate knowledge and reusable ways of acting, and **working memory** maintains ongoing plans. MFM uses searchable textual records to link these representations to the underlying multimodal episodes. Agents can locate past experiences through contextual cues and inspect original observations to recover details absent from their textual descriptions. Through consolidation, agents turn experience into higher-level knowledge and procedures while retaining references to the supporting episodes. Across long-horizon Minecraft and computer-use tasks, MFM improves performance across three base models. With GPT-6-Astra, MFM completes *Mine Diamond from Scratch* and reaches 16 of 24 milestones on the *Ender Dragon* task, compared with 12.7 for the strongest baseline. On CUABench-KiCad, it achieves a mean reward of 0.440, compared with 0.400 for the strongest baseline and 0.267 without persistent memory.

## 1 INTRODUCTION

Intelligent embodied agents that operate autonomously over long horizons in complex, open-ended environments must remember what they have observed, discovered, and previously done (Park et al., 2025; Wang et al., 2026). Much of this memory the agent should remember is often visual or spatial in nature, such as the way back to an object it saw once. It is not obvious what the right form of compact memory the agent should store for such observations. Directly storing everything makes relevant experience hard to find as the history grows. On the other hand, summaries of the memory allows for faster retrieval, but may discard details that matter later (Lu et al., 2026; Zou et al., 2026).

We construct memory as a **multimodal filesystem** (MFM) that jointly stores complementary forms of memory. Our organization draws on distinctions among human memory systems (Tulving, 1972; Squire, 2004). In MFM, *episodic memory* preserves past observations and interactions, while *spatial memory* connects experiences to locations in the environment. *Semantic memory* records facts and relationships, *procedural memory* stores reusable ways of acting, and *working memory* maintains the current plan and immediate subgoals. These five memory functions are implemented through three shared storage components: episodic records, an associative index, and agent-written memory. These components are organized as folders and files in a shared filesystem, with references linking knowledge, procedures, and plans to their supporting episodes.

Prior filesystem-based agent memory has demonstrated the effectiveness of searchable textual interaction records (Fox et al., 2026). For embodied agents, textual descriptions may omit important visual or spatial details. VISTA (Han et al., 2026) preserves visual observations and supports revisiting frames through turn or frame identifiers, retaining information beyond its textual summaries. However, locating the relevant experience can become challenging, especially as memory grows. MFM provides an explicit organization of memory into a filesystem that enables fast context-based retrieval and aggregation of skills and knowledge.



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

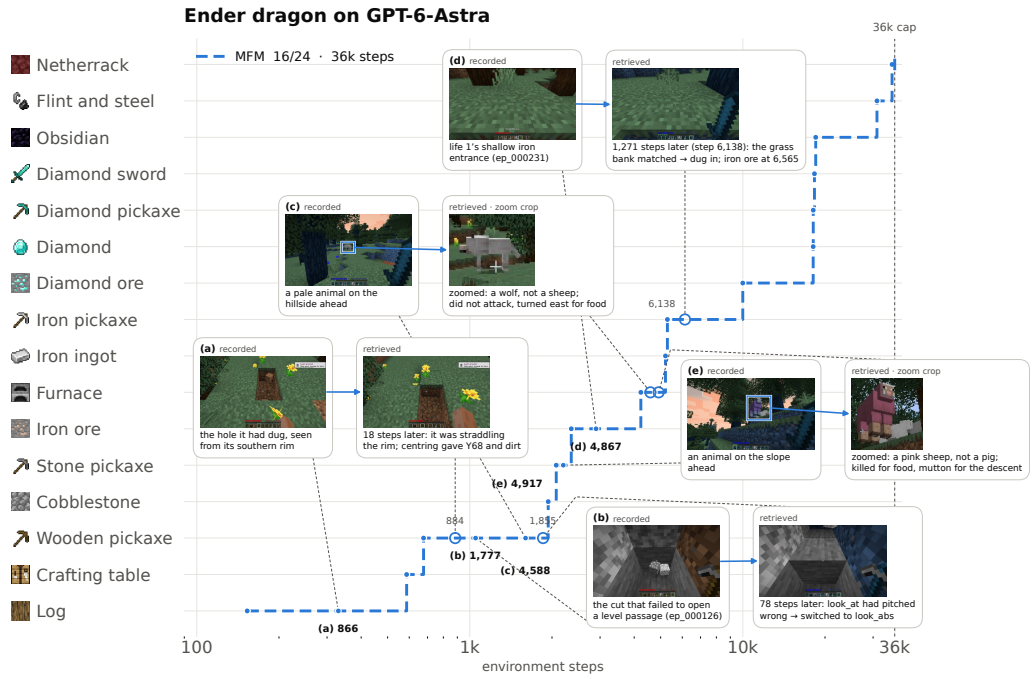



Figure 1: **MFM's Memory Evolution in Minecraft.** Over 36,000 steps, MFM continuously updates its memory and retrieves from it to support decision-making.

As illustrated in Fig. 2, **relational memory** connects these representations through searchable cues to stored evidence. Text provides an *addressing layer* over multimodal records: the agent first searches descriptions of events, actions, and locations, then follows references to inspect the corresponding observations. For example, it can locate an earlier visit to a cave through its coordinates or an attempted action, inspect the recorded view, and compare it with the current observation. The cues used to locate an episode do not need to describe every detail of the episode. This lets us navigate a long history of past episodes through compact records and preserve detailed visual information that may be relevant only later.

As experience accumulates, MFM uses **self-evolving consolidation** to update knowledge, skills, and ongoing working plans. Particular observations can support new facts or relationships, useful action sequences can become reusable procedures, and newly discovered information can change the agent's current plan. These updates connect episodic experience to semantic, procedural, and working memory. For example, the agent may associate a resource with a newly explored cave, record a reliable procedure for reaching it, and revise its route when a later observation shows that the entrance is blocked. Agent-written memories retain references to the episodes that support them, allowing the agent to revisit the original evidence when an interpretation becomes uncertain or contradicted. We illustrate the evolution of the memory in Fig. 1.

Our contributions are threefold. First, we propose a way to organize multimodal agent memory in one filesystem that covers episodic, spatial, semantic, procedural, and working memory. Second, we build this organization into MFM, which pairs cue-based retrieval and visual inspection with consolidation into revisable knowledge and reusable procedures. Third, we show on long-horizon Minecraft and computer-use tasks that MFM outperforms memory-free and memory-augmented baselines with three base models, and that it can reuse memory across tasks. With GPT-6-Astra, our system completes the MCU *Mine Diamond from Scratch* task in all runs and reaches 16 of 24 milestones on *Ender Dragon*, where the strongest baseline averages 12.7. On CUABench-KiCad, its mean reward is 0.440, against 0.400 for the strongest baseline and 0.267 for an agent without persistent memory.

## 2 Related Work

**Filesystem and Multimodal Memory.** Filesystem-based approaches provide persistent records that agents can search and manipulate. PRO-LONG shows that searchable textual interaction logs support long-horizon reasoning (Fox et al., 2026), while other work uses filesystems as organizational

Table 1: **MFM versus prior agent memory systems** across key memory properties.

| Method | Raw multimodal evidence | Index with evidence pointers | Spatial memory | Agent-writable memory | Knowledge linked to evidence | Retrieval |
|---|---|---|---|---|---|---|
| Voyager (Wang et al., 2024) | ✗ | ✗ | ✗ | ✓[a] | ✗ | Embedding similarity |
| JARVIS-1 (Wang et al., 2025) | ✓[b] | ✗ | ✗ | ✓[b] | ✗ | Multimodal similarity |
| MrSteve (Park et al., 2025) | ✓ | ✗ | ✓ | ✗ | ✗ | Place/event lookup |
| PRO-LONG (Fox et al., 2026) | ✗[c] | ✗ | ✗ | ✓[d] | ✗[d] | Regex / code over log |
| VISTA (Han et al., 2026) | ✓ | ✗[e] | ✗ | ✓[e] | ✗ | Inspect by frame ID |
| **MFM (ours)** | ✓ | ✓ | ✓ | ✓ | ✓ | Cue query + evidence inspection |

[a] Skill library (code) only. [b] Successful trajectories only. [c] Text log only. [d] Optional notes, not linked to log entries. [e] Frame IDs without a searchable index; free-form notes.

structures for agent memory (Zhou et al., 2026a). Multimodal memory systems organize long visual streams to enable question answering and reasoning (Fan et al., 2024; Liu et al., 2026a; Long et al., 2026). Additional research examines task-conditioned retention (Zou et al., 2026), retrieval reliability under temporal decay and conflicting evidence (Lu et al., 2026), query-aware relational organization (Li et al., 2026), and analytic operations on accumulated observations (Tian et al., 2026). Table 1 summarizes how MFM differs from related systems.

**Memory for Long-Horizon Interactive Agents.** Long-horizon interaction requires memory that supports sustained perception, exploration, and planning. Recent ARC-AGI-3 systems combine foundation models with interaction harnesses and external memory (ARC Prize Foundation, 2026; Fox et al., 2026; Zeng et al., 2026; Rodionov, 2026), and VISTA maintains visual histories that agents can revisit and compare (Han et al., 2026). In robotics, research focuses on multi-scale memory, learned history compression, and benchmarks for memory-dependent policies (Torne et al., 2026; Wang et al., 2026; Dai et al., 2026). Spatial memory further aids navigation and reasoning (Ge et al., 2026; Zhang et al., 2026). In Minecraft, Voyager builds a library of executable skills (Wang et al., 2024), JARVIS-1 retrieves successful plans from multimodal memory (Wang et al., 2025), and subsequent agents introduce hybrid multimodal memory (Li et al., 2024), what-where-when representations (Park et al., 2025), and accumulated knowledge for self-improvement (Xie et al., 2026). Recent benchmarks highlight exploration and long-horizon competence (Zheng et al., 2025; Ju et al., 2026; Zhou et al., 2026b). While PRO-LONG indexes only text and VISTA revisits frames without a searchable index, MFM offers a general-purpose multimodal episodic filesystem.

## 3 Multimodal Filesystem Memory

We introduce **MFM**, a multimodal filesystem memory architecture for long-horizon agents. MFM combines three complementary components: **episodic records** that preserve multimodal experience, a **searchable index** that provides textual access to those records, and **agent-written memory** that organizes plans, findings, environmental knowledge, and reusable procedures. Fig. 2 outlines this structure, and Fig. 3 walks through a Minecraft example, from episodic recording and indexing to retrieval and consolidation. Section 3.2 details the memory components and how searchable records link to multimodal evidence, Section 3.3 describes how the agent locates and inspects relevant episodes, and Section 3.4 explains how accumulated experience is consolidated into reusable knowledge without discarding the original evidence. Algorithm 1 summarizes the overall loop.

### 3.1 Problem Setup

An agent interacts with an environment over a long horizon to achieve a task objective $g$. At step $t$, it receives an observation $o_t = (v_t, x_t)$, where $v_t$ is a visual observation and $x_t$ contains textual or structured information, and selects an action $a_t$. Its interaction history is $H_t = (o_1, a_1, \ldots, a_{t-1}, o_t)$. As this history grows beyond the model's context budget, the agent must retain information outside its active context and retrieve it when needed. We represent this external memory as $M_t = (E_t, I_t, C_t)$, where $E_t = (e_1, \ldots, e_{N_t})$ stores the $N_t$ completed episodes retained at step $t$, $I_t$ provides an index for locating these episodes, and $C_t$ contains agent-written knowledge. Below, we discuss how these components support complementary forms of memory.

### 3.2 Memory Representation

As illustrated in Figure 2, MFM organizes memory into episodic records of multimodal experience, a searchable index, and agent-written memory. This organization draws on the distinction between episodic memory for particular experiences and semantic and procedural memory for knowledge and ways of acting (Tulving, 1972; Squire, 2004). The episodic store $E_t$ retains observations, states,

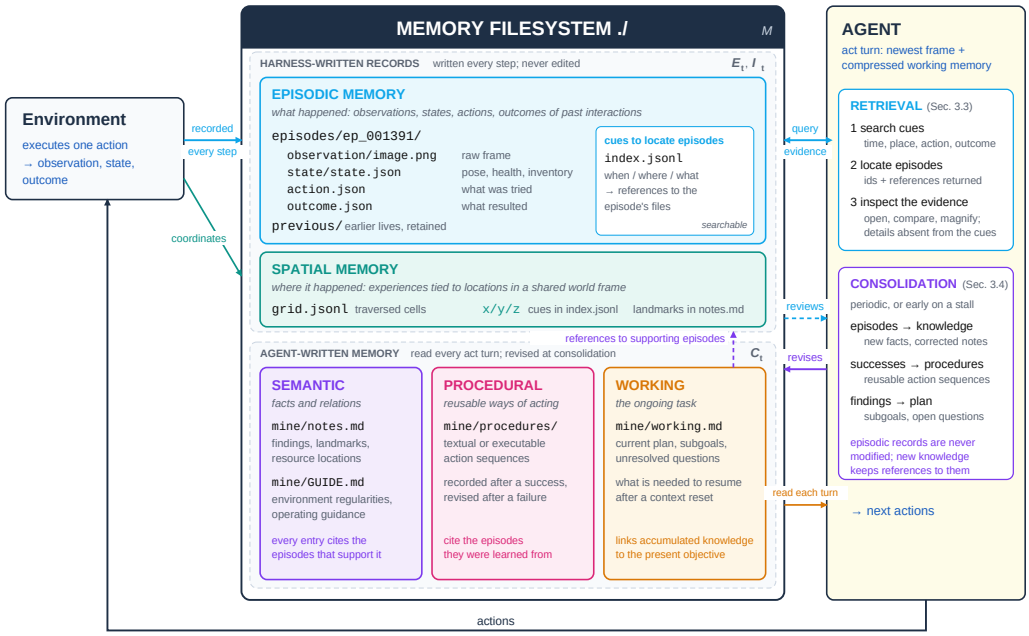



Figure 2: **Overview of MFM.** Five forms of memory in one filesystem. Harness-written **episodic memory** records observations, states, actions, and outcomes every step, with searchable cues for locating them; **spatial memory** ties experiences to locations. Agent-written **semantic**, **procedural**, and **working memory** hold facts, reusable procedures, and the current plan, each citing its supporting episodes. The agent retrieves by searching cues and inspecting original evidence (Sec. 3.3), and consolidates experience into knowledge, procedures, and plans without modifying episodic records (Sec. 3.4).

and actions, while the index $I_t$ provides cues for locating these records. Agent-written memory $C_t$ connects findings across episodes and maintains environmental knowledge, reusable procedures, and working plans. Below, we describe them in detail.

**Episodic memory.** To revisit a past experience, the agent needs access to what it observed, what it attempted, and what happened. We retain episodes as multimodal records,

$$e_i = \big(\mathrm{id}_i, \tau_i, \mathcal{V}_i, \mathcal{S}_i, \mathcal{A}_i, m_i\big),$$

where $\mathrm{id}_i$ is the episode identifier, $\tau_i = [b_i, d_i]$ is the episode's timestep interval, $\mathcal{V}_i$ contains recorded visual observations, $\mathcal{S}_i$ contains associated structured states, $\mathcal{A}_i$ records actions, and $m_i$ contains metadata such as plan progress and task milestones. Episode identifiers and timestep metadata link these records to the corresponding agent interactions. Images retain perceptual details that may be absent from a textual description, while states, actions, and outcomes allow the agent to reconstruct the circumstances of an event. Preserving these records allows the agent to inspect details whose relevance becomes apparent only later. Fig. 3(a) illustrates the contents of an episode.

**Associative indexing.** Episodic memory is useful only if the agent can retrieve relevant experiences when needed. MFM associates each episode with cues describing what happened, where it occurred, and when it took place. This use of cues to access detailed records is motivated by indexing accounts of episodic memory (Teyler & Rudy, 2007). We represent the index as

$$I_t = (z_1, \ldots, z_{N_t}), \qquad z_i = (\mathrm{id}_i, k_i, \rho_i),$$

where $k_i$ contains searchable cues and $\rho_i$ contains references to the stored evidence. The cues include temporal and available location metadata, action type, retrieval tags, and a short action–outcome description. The harness maintains one entry per completed episode in `index.jsonl`.

The index serves as an *addressing layer*: cues narrow the search to relevant episodes, while references provide access to their underlying observations. For example, the agent can locate an encounter near a cave through its recorded location, an object tag, or an attempted action, then inspect the corresponding image. This allows the agent to retrieve an experience through its context without first recalling its precise visual content. Details absent from the index remain available for inspection in the

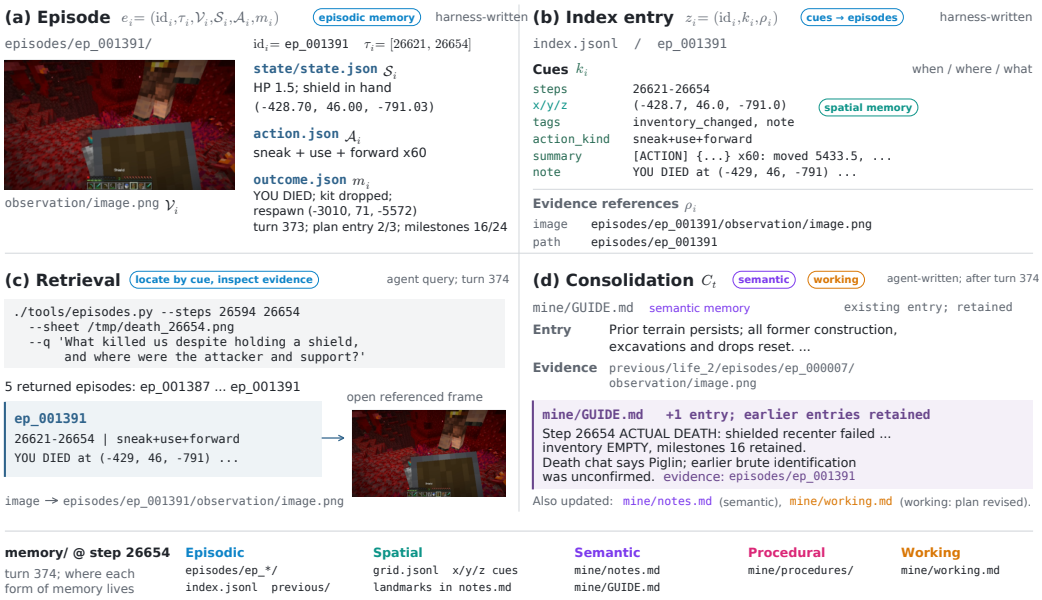



Figure 3: **Memory contents and operations in MFM.** A Minecraft trajectory shows how each form of memory is populated and used. **(a)** An episode: observation, state, action, and outcome stored as persistent files. **(b)** Its index entry: searchable cues, including spatial coordinates, with references to the records. **(c)** Retrieval: the agent locates episodes by step and inspects the referenced frame. **(d)** Consolidation: the evidence updates semantic (`GUIDE.md`) and working memory while earlier entries and episode references are preserved. Bottom: where each form of memory lives at this step.

stored observations. Fig. 3(b) illustrates this separation between cues and evidence, and Algorithm 2 summarizes episode storage and indexing.

**Spatial memory.** For embodied interaction, location provides a common frame for connecting experiences collected at different times. MFM supports this function through spatial information distributed across its episodic store, index, and agent-written memory. Our memory filesystem maintains an append-only grid of traversed cells in a world frame in `grid.jsonl`. Episode coordinates support retrieval by spatial proximity or vertical position, while stored egocentric views let the agent verify a retrieved location by its appearance (Fig. 1(a,d)). The agent further records landmark knowledge in `notes.md`, such as the entrance to a mine that has become blocked, together with coordinates and references to supporting episodes. These representations draw on complementary ideas from cognitive maps (Tolman, 1948), memory for events and their locations (Clayton & Dickinson, 1998), and visual place recognition (Cartwright & Collett, 1983).

**Semantic and procedural memory.** The agent also needs to accumulate knowledge that remains useful beyond a particular episode. MFM stores this knowledge in the agent-written collection $C_t$, which supports semantic and procedural memory alongside the working memory described below. Semantic memory records facts and relations across experiences. For example, a note can associate a resource with a cave, connect the cave to its coordinates, or relate an attempted action to its outcome. These associations are expressed in plain text and include episode identifiers or file paths, allowing the agent to revisit the supporting evidence. We store useful findings and local knowledge in `notes.md`, and broader environmental regularities and operating guidance in `GUIDE.md`.

Procedural memory stores reusable ways of acting. The `procedures/` directory contains textual or executable action sequences that the agent can reuse and revise through subsequent interaction. Periodic consolidation turns accumulated experience into these forms of knowledge and skill (Kumaran et al., 2016). The agent can add, correct, or reorganize them as new evidence arrives, while the underlying episodic records remain unchanged.

**Working memory.** To continue an ongoing task across context changes, MFM maintains the agent's current plan and immediate subgoals in `working.md` (part of $C_t$). This file records the current plan, unresolved questions, and information needed to resume after a context reset. It connects the agent's accumulated experience and knowledge to the decisions required by its present objective.

### 3.3 Memory Retrieval

Retrieving an experience requires both locating a relevant episode and determining what its observations reveal about the current task. MFM separates this into two stages. The agent first searches for episodes through textual or structured cues, then inspects selected multimodal evidence. As the cues determine where to look without requiring the agent to describe every detail, the agent can ask new questions of past observations, including ones whose relevance was not apparent when the experience was recorded. Fig. 3(c) illustrates this process, and Algorithm 3 summarizes the retrieval procedure.

**Agent-initiated search.** Given its current information need, the agent constructs a query $q_t$ over the episode index $I_t$ and receives matching entries $R_t$. A query specifies conditions on recorded context, such as a time interval or location. For example, an agent investigating a failed action can locate episodes around the time it occurred. Search then identifies episodes that match the requested context, while the agent judges which are useful for its current decision. Each result provides the episode identifier, contextual cues, an action–outcome summary, and references to the evidence. Matches are returned in chronological order, with an optional limit that retains only the most recent matches.

**Visual inspection.** Locating an episode provides access to information beyond its indexed description. From the returned entries, the agent selects references $\mathcal{P}_t \subseteq \bigcup_{z_i \in R_t} \rho_i$ and loads the corresponding evidence into context, $D_t = \{\mathrm{Load}(p) \mid p \in \mathcal{P}_t\}$, where Load resolves a reference to its stored frame, state, or other record. This selective loading lets the agent examine the evidence needed without bringing the full interaction history into context. For visual evidence, the agent can inspect an original frame, compare several frames in a contact sheet, juxtapose a past observation with the current view, or magnify a region of interest. These operations support different questions about the same experience, such as identifying an object, recognizing a location, or understanding an action's outcome. Because the observations remain available independently of the textual cues, the agent can examine details absent from $k_i$ and reassess an earlier interpretation. When an inspection includes a question, the agent retains the question alongside the result to preserve the purpose of the lookup.

### 3.4 Self-Evolving Consolidation

Episodic memory preserves past experiences, while semantic and procedural memory capture knowledge and skills for future scenarios. MFM connects these forms of memory through consolidation, drawing on how complementary memory systems support learning from experience (Kumaran et al., 2016). During consolidation, the agent examines accumulated evidence and updates its understanding of the environment, reusable procedures, and its current plan. These updates change the agent-written memory $C_t$ while preserving the recorded history $(E_t, I_t)$. Algorithm 4 summarizes this procedure.

**From experience to knowledge and skills.** Consolidation allows findings from individual episodes to inform subsequent behavior. The agent can record a useful observation as local knowledge in `notes.md`, or update `GUIDE.md` when experience supports a broader regularity. For example, an encounter may establish where a resource was found, while repeated encounters may suggest conditions under which an action succeeds or fails. The agent can also qualify earlier guidance when new observations reveal limits to its applicability.

Procedural updates capture useful ways of acting. The agent can turn an action sequence into a reusable textual or executable procedure, then revise it as subsequent attempts reveal necessary adjustments. Working-memory updates connect accumulated knowledge to the ongoing task: the agent revises `working.md` to reflect completed subgoals, plan changes, and unresolved questions. The agent chooses these updates based on the available evidence and each memory's role, and the resulting documents and procedures remain available during later interaction.

**Revision with preserved episodic evidence.** Knowledge derived from experience may become outdated or inaccurate. MFM links agent-written knowledge to its supporting episodic observations, enabling agents to revisit the basis of earlier beliefs. For example, having noted a sheep on a hillside, the agent later plans to return for food, magnifies the original frame, and finds the animal was a wolf; it corrects the note and revises its plan while keeping the episode reference. If the agent retained only the textual description, it would have lost the visual details needed to correct that interpretation. Preserving episodic evidence therefore allows the agent to reassess earlier conclusions as its understanding improves. Fig. 3(d) illustrates that the agent updates its `GUIDE` to record that the death message identifies a Piglin and that the earlier brute identification was unconfirmed.

# 4 EXPERIMENT RESULTS

We evaluate MFM to understand how persistent multimodal memory affects task performance, decision-making, and interaction efficiency. We compare MFM with baseline methods on long-horizon embodied and computer-use tasks (Section 4.2), examine the memory component contributions via ablation studies (Section 4.3), and illustrate qualitative retrieval behaviors (Section 4.4), with a portal-building case study examining task completion and interaction efficiency, and analyze how the agent learns and reuses skills through memory (Section 4.5).

## 4.1 EXPERIMENTAL SETUP

We evaluate MFM across two diverse domains, averaging all main and ablation results over three random seeds (determining the Minecraft world or KiCad task order). We limit each configuration to three seeds because long-horizon runs with frontier models are expensive.

- **Minecraft**: **MCU-AgentBeats** (Zheng et al., 2025; Liu et al., 2026b): open-world exploration and interaction in Minecraft, scored via scene-specific milestones. The details of both tasks are included in E.1. For the Mine Diamond from Scratch task, we use 12,000 max steps; for the Ender Dragon task, we use 36,000 max steps.
- **CUABench-KiCad**. Built on the CUABench framework (Cua AI, Inc., 2025; Cua AI, 2026) via low-level action spaces, this environment evaluates visual GUI agents on realistic electronic design automation workflows in KiCad. It contains $N = 25$ tasks. For MFM, VISTA, and PRO-LONG, memory can carry over across tasks. A task may receive a score in $[0, 1]$ when the solution satisfies only a subset of the required criteria.

**Baselines and models.** We compare MFM against direct evaluation (no persistent memory), PRO-LONG (Fox et al., 2026), and VISTA (Han et al., 2026) using frontier base models (GPT-5.6-Sol, Claude Opus 5, and GPT-6-Astra). For a fair comparison, all methods share identical base agent prompts, decoding settings, environments, harness, and context management.

Table 2: **Finished milestones on long-horizon MCU tasks.** Number of milestones completed on Mine Diamond from Scratch and Ender Dragon. Results are mean ± std over 3 seeds; best in **bold**.

| Method | Model | Diamond (Max: 10) | Dragon (Max: 24) |
|---|---|---|---|
| Direct Evaluation | GPT-5.6-Sol | 4.0 ± 1.0 | 6.0 ± 1.0 |
| | Claude Opus 5 | 6.0 ± 1.0 | 6.0 ± 1.0 |
| | GPT-6-Astra | 6.3 ± 0.6 | 6.3 ± 0.6 |
| PRO-LONG (Fox et al., 2026) | GPT-5.6-Sol | 8.0 ± 1.0 | 8.3 ± 0.6 |
| | Claude Opus 5 | 8.0 ± 1.0 | 8.0 ± 1.0 |
| | GPT-6-Astra | 9.3 ± 0.6 | 11.0 ± 1.0 |
| VISTA (Han et al., 2026) | GPT-5.6-Sol | 9.3 ± 0.6 | 9.0 ± 1.0 |
| | Claude Opus 5 | 8.7 ± 0.6 | 8.7 ± 0.6 |
| | GPT-6-Astra | 9.7 ± 0.6 | 12.7 ± 0.6 |
| MFM (ours) | GPT-5.6-Sol | 9.7 ± 0.6 | 9.7 ± 0.6 |
| | Claude Opus 5 | 9.3 ± 0.6 | 9.3 ± 0.6 |
| | GPT-6-Astra | **10.0** ± 0.0 | **16.0** ± 0.0 |

Table 3: **Mean reward on CUABench-KiCad.** Results are mean ± std over 3 seeds; best in **bold**.

| Method | Model | Mean Reward |
|---|---|---|
| Direct Evaluation | GPT-5.6-Sol | 0.240 ± 0.026 |
| | Claude Opus 5 | 0.240 ± 0.023 |
| | GPT-6-Astra | 0.267 ± 0.021 |
| PRO-LONG (Fox et al., 2026) | GPT-5.6-Sol | 0.306 ± 0.014 |
| | Claude Opus 5 | 0.300 ± 0.016 |
| | GPT-6-Astra | 0.375 ± 0.012 |
| VISTA (Han et al., 2026) | GPT-5.6-Sol | 0.400 ± 0.011 |
| | Claude Opus 5 | 0.373 ± 0.013 |
| | GPT-6-Astra | 0.400 ± 0.010 |
| MFM (no cross-task memory) | GPT-5.6-Sol | 0.400 ± 0.009 |
| | Claude Opus 5 | 0.375 ± 0.010 |
| | GPT-6-Astra | 0.413 ± 0.009 |
| MFM (ours) | GPT-5.6-Sol | 0.413 ± 0.009 |
| | Claude Opus 5 | 0.389 ± 0.010 |
| | GPT-6-Astra | **0.440** ± 0.013 |

## 4.2 PERFORMANCE ON LONG-HORIZON TASKS

**MCU-AgentBeats**. Table 2 shows completion counts. MFM + GPT-6-Astra finishes all 10 Diamond milestones (vs. 6.3 for direct evaluation, 9.3 for PRO-LONG, and 9.7 for VISTA). On the harder Dragon task, MFM achieves 16/24 milestones, outperforming direct evaluation (6.3), PRO-LONG (11.0), and VISTA (12.7). We observe that every baseline plateaus, making no further progress after at most 13 milestones. MFM consistently improves performance across all base models.

**CUABench-KiCad**. Table 3 indicates that MFM's benefits extend to computer-use tasks, yielding mean rewards of 0.413 (GPT-5.6-Sol), 0.389 (Opus 5), and 0.440 (GPT-6): improving over direct evaluation baselines by 0.149–0.173 and outperforming memory baselines across models. Without cross-task memory, MFM's reward drops by 0.013–0.027 across models.

Table 4: **Ablation of MFM's memory components.** Effects of removing stored frames, associative indexing, agent-written procedures or consolidation on the Diamond and Dragon tasks with GPT-5.6-Sol.

| | No visual obs. | | Flat | | No agent procedures | | No consolidation | |
|---|---|---|---|---|---|---|---|---|
| Model | Diamond | Dragon | Diamond | Dragon | Diamond | Dragon | Diamond | Dragon |
| GPT-5.6-Sol | $8.0 \pm 1.0$ | $8.3 \pm 0.6$ | $8.7 \pm 0.6$ | $8.7 \pm 0.6$ | $8.7 \pm 0.6$ | $9.0 \pm 1.0$ | $8.3 \pm 0.6$ | $8.3 \pm 0.6$ |

**Temporal reach and interaction efficiency.** Fig. 6 reveals that MFM accesses much earlier history than VISTA (which favors recent experience) or PRO-LONG, demonstrating the effectiveness of MFM's addressing layer. Furthermore, Fig. 8 shows that MFM and PRO-LONG execute more actions per turn than VISTA, with fewer total model queries and reducing execution overhead.

### 4.3 Ablation Studies

We evaluate the contributions of MFM's memory components through four ablations that separately remove access to historical visual observations, associative indexing, agent-written procedures, and consolidation. Table 4 reports the ablation results. **Historical visual observations** appear essential for resolving perceptual ambiguities and spatial orientation (Section 4.4), since disabling stored frames drops completion to $8.0 \pm 1.0$ (Diamond) and $8.3 \pm 0.6$ (Dragon). **Associative indexing** suggests that a dedicated addressing layer helps; replacing it with chronological text matching over a flat log (Flat) reduces scores to $8.7 \pm 0.6$ on both tasks. Additionally, **agent-written procedures** seem to help compound past successes into reusable routines (Section 4.5), as disabling it tends to decrease performance (Diamond: 8.7; Dragon: 9.0). Finally, the removal of **consolidation** lowers performance to $8.3 \pm 0.6$ on both tasks. The consistent drops across all four ablations is the main evidence that each component contributes.

### 4.4 How Memory Retrieval Helps in Long-Horizon Tasks

We examine qualitative examples to understand how MFM supports decisions during long-horizon interaction. These examples illustrate how agents inspect visual evidence to revise earlier interpretations, recover spatial context from previous observations, and reuse evidence across tasks.

**MCU-AgentBeats**. *Visual inspection enables agents to correct uncertain observations.* In Fig. 9(a,b), magnifying ambiguous frames reveals a wolf instead of a sheep, preventing a costly attack; in (c,d), pink sheep observations guide resource gathering. *Retrieving earlier observations also helps the agent recover spatial context and connect past experience to the current environment.* Agents utilize past frames to locate hole rims (Fig. 10), identify mine entrances (Fig. 11), and correct unfavorable viewing angles (Fig. 12).

We further examine a controlled construction task requiring the agent to align visual observations across multiple actions with a textual specification (full prompt in Box 2). Starting on a cleared stone platform with sufficient resources, the agent must build and activate a Nether portal. This probes multimodal capability: the agent must translate textual geometric constraints into a 3D structure, track construction progress from first-person views, and adjust accordingly. Only MFM successfully built the portal under a 6,000-step budget (Fig. 4), requiring lower overhead (Fig. 5): 30 model calls, 77 tool calls, and 77 environment actions (vs. 141/140/129 for VISTA and 43/131/151 for PRO-LONG).



(a) MFM completes the portal. (b) VISTA (c) PRO-LONG

Figure 4: **Qualitative behavior on the Nether portal task.** MFM successfully completes the Nether portal, whereas VISTA and PRO-LONG fail to finish the structure.

**CUABench-KiCad.** Fig. 13 shows 3 tasks where MFM outperforms VISTA (GPT-6-Astra). In Task 55f2eefb (switching an LED circuit's supply from 9 V to 5 V), both methods log an "OK" from

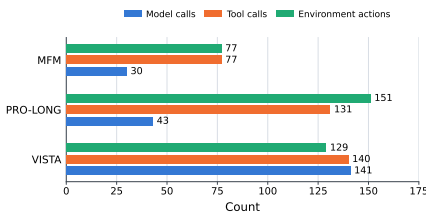



Figure 5: **Interaction efficiency on the Nether portal task.** Comparison of model calls, tool calls, and executed environment actions.

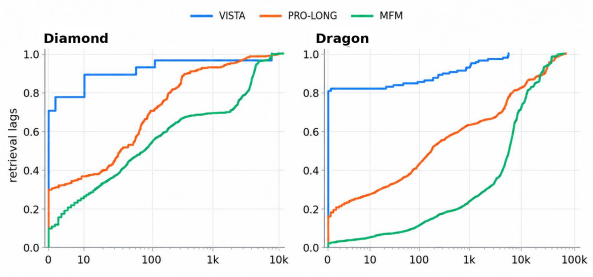



Figure 6: **Temporal reach of memory retrieval.** Retrieval-lag distributions on Diamond and Dragon. VISTA favors recent experience; MFM reaches further back, especially on Dragon.

Change Symbol, but only MFM verifies the resulting netlist: its final state reads +5V while VISTA's still reads +9V, underscoring the need to check task state through visual memory. On a buck-regulator task (c4baee36), both methods produce identical components and wiring but different supply labels; on the two-op-amp task (b7148878), MFM leaves 1 input pin unconnected versus 3 for VISTA. Fig. 15 shows episodic retrieval across tasks: an export sequence from one task is reused on another.

### 4.5 Skill Learning through Memory

To investigate whether MFM enabled *skill learning* rather than mere episodic recall, we examine whether the agent learned and reused behaviors or routines. Extended rollouts reveal that memory contained both reusable capabilities and world-specific knowledge about resources, hazards, and previously explored regions. For example, during Diamond runs, the agent abstracted frequently encountered repetitive subtasks into parameterized Python **procedures**. It established a reliable descent procedure: `shaft_down` C.1, and created `tunnel` C.2 after successfully excavating four cells by hand. Additionally, `probe` C.3 inserts a release action between attack sequences, preventing them from merging and enabling the intended 3-block excavation. `aim_speckles` C.4 was called to obtain the diamond. The agent then selectively reused and improved this procedural library. Skills may also stay as natural-language knowledge. For instance, `spruce_approach` C.5 started as a raw episodic record during the agent's first encounter. The second time, it distilled that into an entry in GUIDE.md and compiled it into its plan (`working.md`). By the third time, it refined the entry and maintained the place-specific verification habit.

## 5 Conclusion

We introduced MFM, a multimodal filesystem memory that organizes episodic, spatial, semantic, procedural, and working memory as folders and files in one shared filesystem. Episodic records preserve raw observations, states, actions, and outcomes; a searchable index of contextual cues addresses them; and agent-written knowledge, procedures, and plans cite the episodes that support them. An agent can thus locate past experience by when, where, or what happened, inspect the original observation for details text omits, and revise what it has learned without discarding the evidence. Across Minecraft and KiCad with three base models, MFM outperforms memory-free and memory-augmented baselines, each component contributes, and memory transfers across tasks. We hope MFM can serve as a foundation for memory in long-horizon multimodal and embodied agents.

**Limitations and Future Work**. MFM improves long-horizon performance, but exploration remains hard: even GPT-6-Astra at maximum reasoning effort did not complete Ender Dragon. Finding better ways of coupling memory to exploration and route planning could help agents find landmarks and avoid repeated traversal. Our evaluation is also limited to single agents in Minecraft and KiCad; shared memory across agents could combine experience from different regions or solution attempts, and broader environments would test how far the design transfers.

### AI USE STATEMENT

We used generative AI tools to assist with writing and editing, including improving the manuscript's clarity, grammar, and organization, and with software development, including code generation, debugging, and documentation. We also used generative AI tools during research to assist with literature discovery and brainstorming. The authors reviewed and revised all AI-assisted text, independently verified cited literature against the original sources, and inspected and tested AI-assisted code as part of our experimental workflow. Generative AI did not autonomously determine or fabricate experimental results, and all reported results came from our implemented experiments. We take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

### ETHICS STATEMENT

This work studies memory architectures for long-horizon AI agents in simulated and computer-use environments. Our experiments do not involve human subjects or the collection of private or personally identifiable information. While improved long-horizon memory may increase the capabilities of autonomous agents and could potentially be applied in higher-risk settings, our work focuses on evaluating memory and agent behavior in controlled benchmark environments. We encourage future work deploying such systems in real-world settings to consider appropriate access controls, privacy protections, and human oversight. To the best of our knowledge, this work raises no additional concerns about dataset consent, discrimination, or conflicts of interest beyond those associated with the underlying benchmarks and models used in our experiments.

### REPRODUCIBILITY STATEMENT

We provide detailed descriptions of MFM's memory representation, retrieval, consolidation, and revision mechanisms in Section 3, together with the experimental settings and baselines used in our evaluations. The appendix provides additional implementation details, prompts, hyperparameters, benchmark configurations, and ablation settings needed to reproduce our experiments. We will release the MFM implementation, including the memory harness, retrieval tools, and evaluation code, along with instructions for reproducing the reported experiments. Where applicable, we use publicly available benchmarks and document any modifications to their original evaluation protocols.

## REFERENCES

ARC Prize Foundation. Arc-agi-3. https://arcprize.org/, 2026. Accessed: 2026-08-15. 3

BA Cartwright and Thomas S Collett. Landmark learning in bees: experiments and models. *Journal of comparative physiology*, 151(4):521–543, 1983. 5

Nicola S Clayton and Anthony Dickinson. Episodic-like memory during cache recovery by scrub jays. *Nature*, 395(6699):272–274, 1998. 5

Cua AI. Cua-Bench Leaderboard. https://cua.ai/cuabench, 2026. Accessed: 2026-09-12. 7

Cua AI, Inc. Cua, 2025. URL https://github.com/trycua/cua. cua-bench KiCad EDA dataset (25 schematic-editing tasks, netlist-comparison scoring). MIT License. Accessed 2026-09-01. 7

Yinpei Dai, Hongze Fu, Jayjun Lee, Yuejiang Liu, Haoran Zhang, Jianing Yang, Chelsea Finn, Nima Fazeli, and Joyce Chai. Robomme: Benchmarking and understanding memory for robotic generalist policies. *arXiv preprint arXiv:2603.04639*, 2026. 3

Yue Fan, Xiaojian Ma, Rujie Wu, Yuntao Du, Jiaqi Li, Zhi Gao, and Qing Li. Videoagent: A memory-augmented multimodal agent for video understanding. In *European Conference on Computer Vision*, pp. 75–92, 2024. 3

Alexis Fox, Junlin Wang, Paul Rosu, and Bhuwan Dhingra. Pro-long: Programmatic memory enables long-horizon reasoning. *arXiv preprint arXiv:2607.20064*, 2026. 1, 2, 3, 7

Zuhao Ge, Xiaosong Jia, Chao Wu, Yuchen Zhou, Zuxuan Wu, and Yu-Gang Jiang. Evomemnav: Efficient self-evolving fine-grained memory for zero-shot embodied navigation. *arXiv preprint arXiv:2606.03509*, 2026. 3

Qiushi Han, Keya Hu, Linlu Qiu, Cathy Wu, and Kaiming He. VISTA: A visual harness for reasoning in an interactive world, aug 2026. URL https://vista-research.github.io/. 1, 3, 7

Tianjie Ju, Yueqing Sun, Zheng Wu, Wei Zhang, Yaqi Huo, Xi Su, Qi Gu, Xunliang Cai, Gongshen Liu, and Zhuosheng Zhang. Mineexplorer: Evaluating open-world exploration of mllm agents in minecraft. *arXiv preprint arXiv:2605.30931*, 2026. 3

Dharshan Kumaran, Demis Hassabis, and James L McClelland. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7): 512–534, 2016. 5, 6

Geng Li, Yuhao Wang, Dong Li, Jianye Hao, and Yuxin Peng. Graphmemix: Query-aware evidence forests for long-term multimodal agent memory. *arXiv preprint arXiv:2608.26983*, 2026. 3

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. 3

Runtao Liu, Ziyi Liu, Jiaqi Tang, Yue Ma, Renjie Pi, Jipeng Zhang, and Qifeng Chen. Longvideoagent: Multi-agent reasoning with long videos. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 40404–40416, 2026a. 3

Xiaoyuan Liu, Jianhong Tu, Yuqi Chen, Siyuan Xie, Sihan Ren, Tianneng Shi, Gal Gantar, Evan Sandoval, Donghyun Lee, Daniel Miao, et al. Agentbeats: Agentifying agent assessment for openness, standardization, and reproducibility. *arXiv preprint arXiv:2606.13608*, 2026b. 7

Lin Long, Yichen He, Wentao Ye, Yiyuan Pan, Yuan Lin, Hang Li, Junbo Zhao, and Wei Li. Seeing, listening, remembering, and reasoning: A multimodal agent with long-term memory. In *International Conference on Learning Representations*, volume 2026, pp. 146197–146246, 2026. 3

Yihao Lu, Wanru Cheng, Zeyu Zhang, and Hao Tang. Mma: Multimodal memory agent. *arXiv preprint arXiv:2602.16493*, 2026. 1, 3

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. 1, 3

Sergey Rodionov. Executable world models for arc-agi-3 in the era of coding agents. In *International Conference on Artificial General Intelligence*, pp. 198–210. Springer, 2026. 3

Larry R Squire. Memory systems of the brain: A brief history and current perspective. *Neurobiology of learning and memory*, 82(3):171–177, 2004. 1, 3

Timothy J Teyler and Jerry W Rudy. The hippocampal indexing theory and episodic memory: updating the index. *Hippocampus*, 17(12):1158–1169, 2007. 4

Zhoujin Tian, Yao Tian, Hao Zhang, Cheng Chen, Yakun Li, Lei Zhang, and Xiaofang Zhou. Beyond retrieval: Analytic memory for multimodal agents. *arXiv preprint arXiv:2607.29440*, 2026. 3

Edward C Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948. 5

Marcel Torne, Karl Pertsch, Homer Walke, Kyle Vedder, Suraj Nair, Brian Ichter, Allen Z Ren, Haohuan Wang, Jiaming Tang, Kyle Stachowicz, et al. Mem: Multi-scale embodied memory for vision language action models. *arXiv preprint arXiv:2603.03596*, 2026. 3

Endel Tulving. Episodic and semantic memory. *Organization of memory*, 1(381-403):1, 1972. 1, 3

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. *Transactions on Machine Learning Research*, 2024. 3

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, 2025. 3

Ziye Wang, Modi Shi, Chaojun Ni, Jiazhi Yang, Mengdi Li, Zhizhong Su, Tianwei Lin, and Hongyang Li. Nativemem: Native memory compression for long-horizon robotic manipulation. *arXiv preprint arXiv:2607.06678*, 2026. 1, 3

Zhengwei Xie, Zhisheng Chen, Ziyan Weng, Jinhan Li, Chenglong Li, Zikai Xiao, Jingwei Song, Jinhao Jing, Vireo Zhang, and Kun Wang. Mineevolve: Self-evolution with accumulated knowledge for long-horizon embodied minecraft agents. *arXiv preprint arXiv:2603.13131*, 2026. 3

Guanning Zeng, Jiani Wang, Wenjie Ma, Shaofeng Yin, Chenyang Wang, Shichen Liu, Angjoo Kanazawa, Wode Ni, Xiuyu Li, Andrea Zanette, and Haiwen Feng. [schema]: Frontier models with the right harness achieve ∼99% on arc-agi-3 public. Impossible Research. https://schema-harness.github.io/, 2026. URL https://schema-harness.github.io/. 3

Haokai Zhang, Yuhang Ding, Yunshu Zhou, Xinze Du, Shengtao Zhang, Zhiyue Zhao, Yuling Xi, and Hao Chen. Spatial memory agent: Experience-grounded procedure memory for spatial intelligence. *arXiv preprint arXiv:2608.12743*, 2026. 3

Xinyue Zheng, Haowei Lin, Kaichen He, Zihao Wang, Qiang Fu, Haobo Fu, Zilong Zheng, and Yitao Liang. Mcu: An evaluation framework for open-ended game agents. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. URL https://arxiv.org/abs/2310.08367. 3, 7

Sizhe Zhou, Sheldon Yu, Hui Wei, Junda Wu, Siru Ouyang, Yizhu Jiao, Shijia Pan, Julian McAuley, Yu Zhang, Tong Yu, et al. Filesystem-based memory for llm agents: Organization, evolution, and sustainability. *arXiv preprint arXiv:2607.26637*, 2026a. 3

Ziheng Zhou, Huacong Tang, Jinyuan Zhang, Haowei Lin, Bangcheng Yang, Qian Long, Fang Sun, Yizhou Sun, Yitao Liang, Ying Nian Wu, et al. Can current agents close the discovery-to-application gap? a case study in minecraft. *arXiv preprint arXiv:2604.24697*, 2026b. 3

Tao Zou, Yichen He, Tian Qiu, Yuan Lin, and Hang Li. Task-focused memorization for multimodal agents. *arXiv preprint arXiv:2605.31075*, 2026. 1, 3

# APPENDIX - TABLE OF CONTENTS

**A Additional Examples** **13**

A.1 Milestone Progression and Interaction in Minecraft . . . . . 13

A.2 Visual Memory Inspection in Minecraft . . . . . 13

A.3 Qualitative Examples on CUABench-KiCad . . . . . 14

**B Pseudocode** **19**

**C Skills** **21**

**D Harness and Implementation Details** **26**

D.1 Episode Index . . . . . 26

D.2 Spatial Grid . . . . . 26

D.3 Method Configurations . . . . . 27

D.4 Storage and Cost . . . . . 28

**E Benchmark Setup and Prompts** **31**

E.1 MCU Tasks Descriptions . . . . . 31

E.2 Prompts and Outputs Examples . . . . . 31

## A ADDITIONAL EXAMPLES

This appendix adds results and qualitative examples to Section 4.4. Section A.1 covers milestone progression and interaction on Minecraft. Section A.2 shows further cases of visual memory inspection. Section A.3 gives qualitative examples on CUABench-KiCad.

### A.1 MILESTONE PROGRESSION AND INTERACTION IN MINECRAFT

Fig. 7 tracks milestone progression on Kill the Ender Dragon with GPT-6-Astra, using one run per method. VISTA gets diamond equipment earlier than MFM, but its progress then stalls. MFM continues through obsidian and flint and steel and reaches netherrack within the same 36,000-step budget. Table 2 reports seed-averaged results for all methods and base models.

Fig. 8 shows how many environment steps each method executes between successive model calls. MFM and PRO-LONG generally execute more steps per turn than VISTA. In other words, they issue longer action plans and call the model less often. How MFM compares with PRO-LONG depends on the task. On Ender Dragon, MFM executes more steps per turn than both baselines for 42–56% of each run. On Mine Diamond, it does so for only 5–14% of each run, and PRO-LONG usually executes the most steps per turn.

### A.2 VISUAL MEMORY INSPECTION IN MINECRAFT

Figs. 9 to 12 expand on the Minecraft examples in Section 4.4. In each case, the agent finds a stored frame through its index entry. It then inspects the frame to answer a question that the textual record alone cannot: what kind of animal it saw (Fig. 9), where a hole it dug earlier is located (Fig. 10), whether a place seen in an earlier life matches the current view (Fig. 11), and why an earlier mining cut failed (Fig. 12).

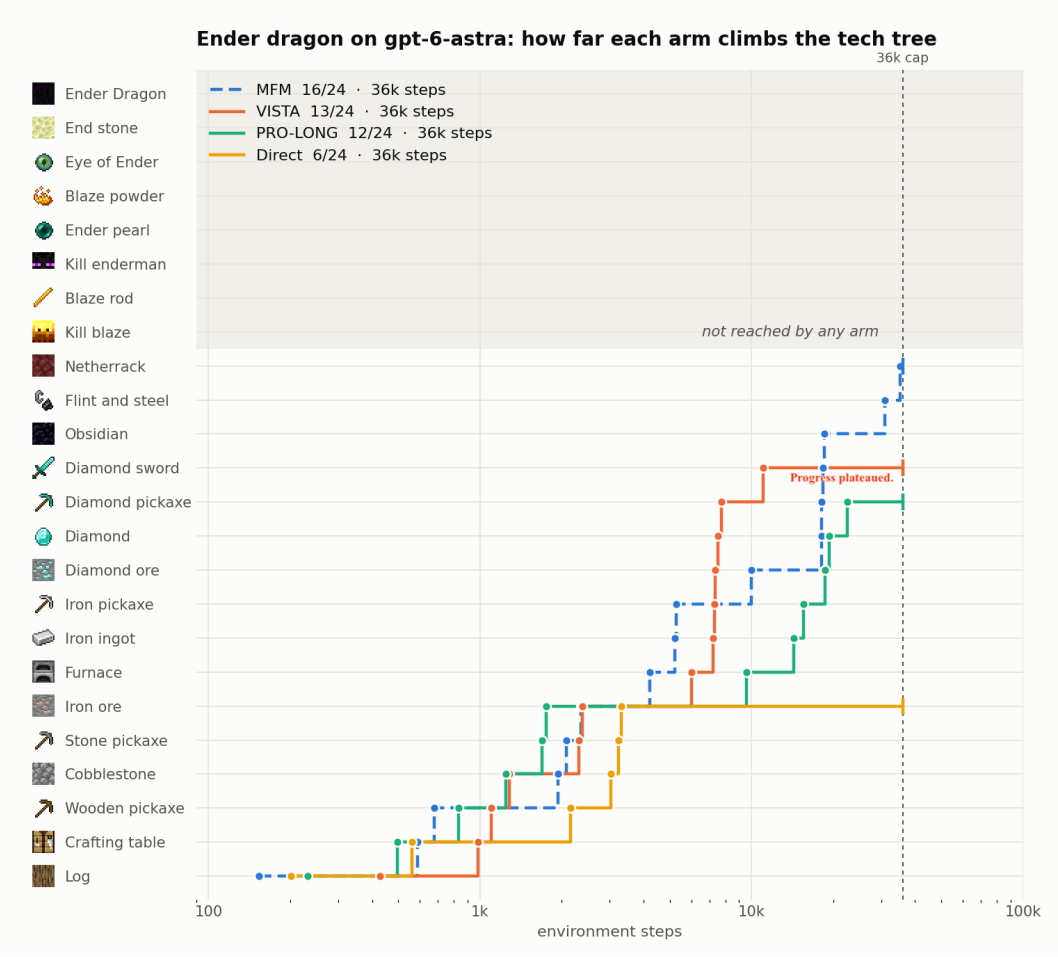



Figure 7: **Milestone progression on Kill the Ender Dragon with GPT-6-Astra.** Each curve shows one run's progression through the Minecraft technology tree against environment steps (log scale). Within the 36,000-step budget (dashed line), MFM reaches 16 of 24 milestones, compared with 13 for VISTA, 12 for PRO-LONG, and 6 for Direct Evaluation. The shaded region marks milestones that no method reached. One run is shown per method; Table 2 reports seed-averaged results.

### A.3 Qualitative Examples on CUABench-KiCad

Fig. 13 shows the final schematics that MFM and VISTA produce on the three tasks discussed in Section 4.4. On Task 55f2eefb, both agents get an "OK" from Change Symbol, but only MFM checks the resulting value; VISTA's schematic still reads +9 V. Because MFM checks the resulting application state instead of trusting the "OK" returned by an edit operation, it catches incomplete edits that VISTA records as successful. On Task c4baee36, both agents place the same components and wiring but use different supply labels. On Task b7148878, MFM leaves one pin unconnected, compared with three for VISTA.

Fig. 14 shows how MFM verifies its own edits on three other tasks. In each task, the agent follows the same three-step routine. First, it inspects the schematic to confirm that component values and net labels match the specification. Second, it runs the Electrical Rules Checker (ERC). Third, it reads the ERC report to confirm that only the expected undriven-power errors remain and that no pins are left unconnected. The agent first used this routine in an earlier task and later retrieved it from episodic memory, similar to the netlist-export reuse in Fig. 15.

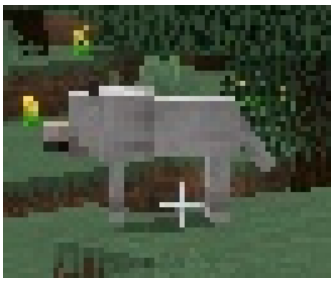
Fig. 15 shows episode reuse across tasks. While exporting a netlist in Task `a6414tw4`, the agent retrieves an episode from the earlier Task `4c913e30` whose action sequence is nearly identical, and follows it as a precedent.




Figure 8: **Environment interaction per agent turn.** Rolling median of environment steps executed per agent turn over each run. Columns are the three base models; rows are Mine Diamond from Scratch (top) and Kill the Ender Dragon (bottom). Higher values mean more environment interaction between successive model calls; the vertical axis is logarithmic. Shaded intervals mark where MFM exceeds both VISTA and PRO-LONG, and the annotation in each panel gives the shaded fraction.



(a) Stored frame: a pale animal on the hillside.



(b) Magnified crop: a wolf.



(c) Stored frame: a possible pink sheep.



(d) Magnified crop: a pink sheep.

Figure 9: **Correcting and confirming perception with visual memory.** (a–b) At step 4,588, the agent records a pale animal on the hillside. Magnifying the stored frame shows a wolf rather than a sheep, so the agent does not attack and looks for food elsewhere. (c–d) At step 4,917, the agent is unsure whether an animal on the slope is a pink sheep. Magnifying the stored frame confirms a sheep rather than a pig, and the agent hunts it for food.



(a) Stored frame of the hole the agent dug.

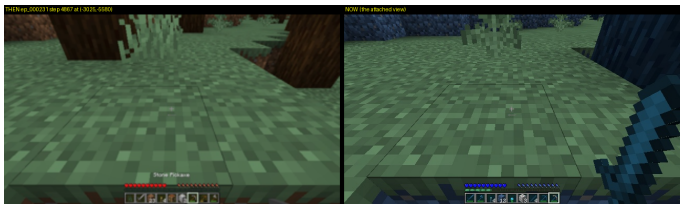


(b) Retrieved frame (left) beside the current view (right).

Figure 10: **Memory-guided spatial relocation.** The retrieved and current views line up closely, which lets the agent identify the hole's rim and recover its exact location.



(a) Life 1: stored frame of a shallow iron-mine entrance.



(b) Life 2: retrieved frame (left) beside the current view (right).

Figure 11: **Cross-life visual memory retrieval.** After dying, the agent retrieves a frame recorded in an earlier life and compares it with its current view. It recognizes the grass bank ahead as the entrance to that life's iron mine, digs in, and mines iron ore at step 6,565.

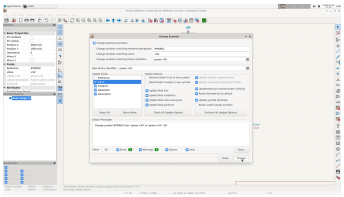


(a) Retrieved frame: the earlier cut was pitched upward.

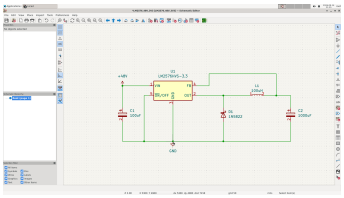


(b) Re-aimed view facing the mining face.

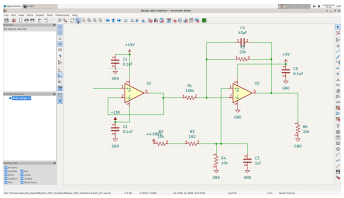
Figure 12: **Using visual memory to recover mining orientation.** The agent inspects the frame of an earlier cut that failed to open a level passage. It sees that the view was pitched upward rather than at the mining face. It then re-aims with `look_abs` and resumes mining at the correct orientation.



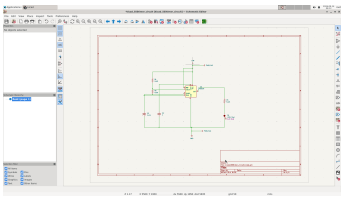
(a) MFM, 55f2eefb: supply reads +5 V.



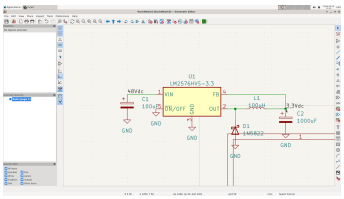
(b) MFM, c4baee36: +48 V and GND supply symbols.



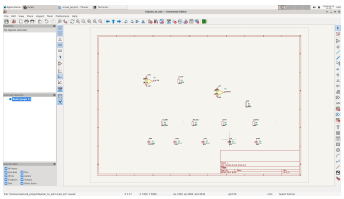
(c) MFM, b7148878: one unconnected pin.



(d) VISTA, 55f2eefb: label still reads +9 V.

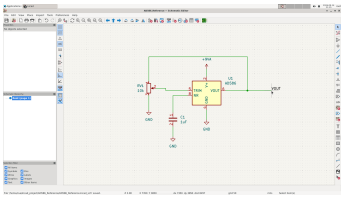


(e) VISTA, c4baee36: local label `3.3VDC` on the output net.

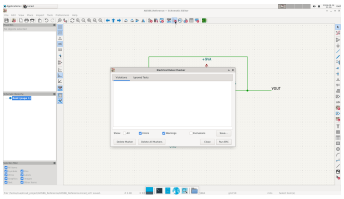


(f) VISTA, b7148878: pins R1.1, U2.1, and U2.2 unconnected.

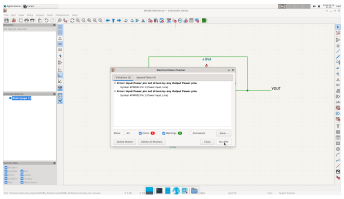
Figure 13: **Qualitative comparison on CUABench-KiCad.** Final schematics produced by MFM (top) and VISTA (bottom); each column is one task. (a, d) Task 55f2eefb, which switches an LED circuit's supply from 9 V to 5 V: VISTA's label still reads +9 V even though Change Symbol returned "OK". (b, e) Task c4baee36, a buck regulator: both agents place the same components and wiring but use different supply labels. (c, f) Task b7148878, a two-op-amp circuit: MFM leaves one pin unconnected, compared with three for VISTA.



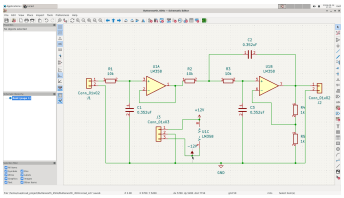
(a) `ba34f1c3`: the label and all five nets match the design; the title bar confirms the save.



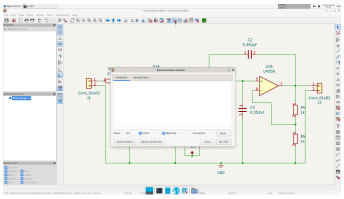
(b) ERC dialog open; no error feedback from earlier actions.



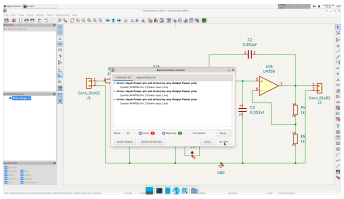
(c) Only the two expected undriven-power errors; zero warnings.



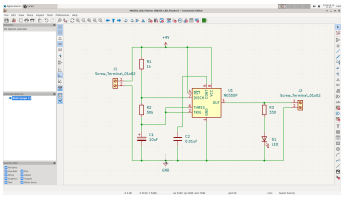
(d) `4c913e30`: wiring matches the committed design; the schematic is saved in the required directory.



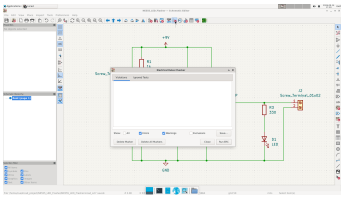
(e) ERC dialog open; no error feedback from earlier actions.



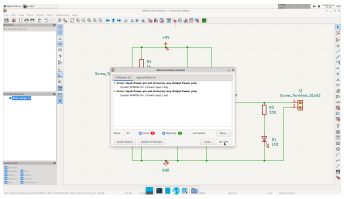
(f) Only the three expected undriven-power errors (external supplies); zero warnings.



(g) `5dbc9e02`: values, polarities, and all seven nets match the design.



(h) The title and status bar confirm the save location; ERC dialog open.

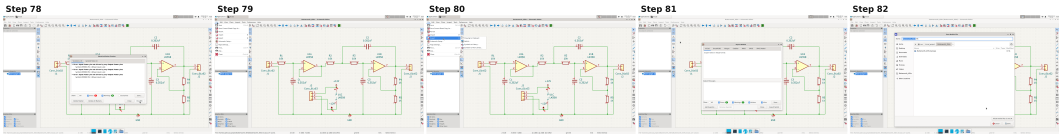


(i) Two expected undriven-power errors; zero warnings and no unconnected pins.

Figure 14: **Self-verification of edits on CUABench-KiCad.** Each row shows one task. MFM inspects the schematic against the specification (left), opens the Electrical Rules Checker (middle), and reads the report to confirm that only the expected errors remain (right).




(a) Current task `a6414tw4`, steps 6–10.




(b) Episode retrieved from the earlier task `4c913e30`, steps 78–82.

Figure 15: **Cross-task reuse of episodic memory in KiCad.** While working on the current task (top), the agent retrieves an episode from an earlier task (bottom) with a nearly identical action sequence. The sequence is: save the blank schematic, open *Export* → *Netlist*, select the native KiCad format, and set the output path. The retrieved episode serves as a concrete precedent for the current task.

## B PSEUDOCODE

**Notation.** Let $g$ denote the task objective, $o_t = (v_t, x_t)$ an observation, and $M_t = (E_t, I_t, C_t)$ the memory at step $t$. Here $E_t$ stores completed multimodal episodes, $I_t$ provides searchable cues and references to those episodes, and $C_t$ contains revisable agent-written knowledge. The algorithms use $E$, $I$, and $C$ for their current values. The initial memory may be empty or contain retained experience. An episode groups a contiguous interaction interval; $B$ holds the interval currently being recorded, and $\|$ denotes sequence concatenation.

**Algorithm 1** MFM: Multimodal Filesystem Memory

**Require:** Objective $g$; environment $\mathcal{E}$; initial memory $M$; horizon $T$
**Ensure:** Updated memory $M$

$(E, I, C) \leftarrow M$; $B \leftarrow \langle\rangle$; $o_1 \leftarrow \mathcal{E}.\textsc{Observe}()$
**for** $t = 1, \ldots, T$ **do**
  $D_t \leftarrow \emptyset$
  **if** past experience is needed **then**
    $q_t \leftarrow \textsc{FormQuery}(g, C, \{o_t\})$
    $D_t \leftarrow \textsc{Retrieve}(q_t, I)$
  **end if**
  **if** consolidation is triggered **then**
    $C \leftarrow \textsc{Consolidate}(g, C, E, I, \{o_t\} \cup D_t)$
  **end if**
  $a_t \leftarrow \textsc{SelectAction}(g, o_t, C, D_t)$
  $(o_{t+1}, f_t) \leftarrow \mathcal{E}.\textsc{Step}(a_t)$
  Persist $(o_t, a_t, o_{t+1}, f_t)$ with their interaction identifiers
  Append references to these records to $B$
  **if** an episode boundary is reached **then**
    $(E, I) \leftarrow \textsc{RecordEpisode}(B, E, I)$; $B \leftarrow \langle\rangle$
  **end if**
  **if** execution terminates **then**
    **break**
  **end if**
**end for**
**if** $B \neq \langle\rangle$ **then**
  $(E, I) \leftarrow \textsc{RecordEpisode}(B, E, I)$
**end if**
**return** $M = (E, I, C)$

Here $f_t$ denotes available interaction feedback. Episode boundaries and consolidation triggers are supplied by the interaction protocol. Retrieval may be repeated as needed, and action selection may reuse an existing plan or procedure. Each resulting observation is recorded before its episode is finalized, including at termination or the end of the horizon.

**Algorithm 2** Episodic recording and indexing

**function** $\textsc{RecordEpisode}(B, E, I)$
  $id_i \leftarrow$ a unique episode identifier
  $(\tau_i, V_i, S_i, A_i, m_i) \leftarrow \textsc{OrganizeRecords}(B)$
  $e_i \leftarrow (id_i, \tau_i, V_i, S_i, A_i, m_i)$
  $\rho_i \leftarrow \textsc{StoreEpisode}(e_i)$ ▷ Preserve the original evidence
  $k_i \leftarrow \textsc{BuildCues}(e_i)$
  $z_i \leftarrow (id_i, k_i, \rho_i)$
  **return** $(E \| \langle e_i \rangle,\ I \| \langle z_i \rangle)$
**end function**

The episode contains its interval $\tau_i$, visual observations $V_i$, structured states $S_i$, attempted actions $A_i$, and metadata $m_i$. The cues $k_i$ include available temporal, spatial, action, and outcome descriptions;

**Algorithm 3** Retrieval through textual cues and evidence references

**function** RETRIEVE($q, I$)
  $R \leftarrow \text{FILTER}(I, q)$ ▷ Retain entries satisfying all query predicates
  $P \leftarrow \text{SELECTREFERENCES}(q, R)$
  $D \leftarrow \{\text{LOAD}(p) : p \in P\}$
  **return** $D$
**end function**

**Algorithm 4** Self-evolving consolidation

**function** CONSOLIDATE($g, C, E, I, D$)
  $D^{\text{cons}} \leftarrow D$
  **while** additional evidence is needed **do**
    $q \leftarrow \text{FORMQUERY}(g, C, D^{\text{cons}})$
    $D^{\text{cons}} \leftarrow D^{\text{cons}} \cup \text{RETRIEVE}(q, I)$
  **end while**
  $C^{+} \leftarrow \mathcal{U}(C, D^{\text{cons}}, g)$
  **return** $C^{+}$ ▷ $E$ and $I$ remain unchanged
**end function**

$\rho_i$ references the underlying stored evidence. Organizing and indexing records does not replace or compress their original multimodal content.

Filtering preserves index order and may restrict results by recency when requested. The agent selects $P \subseteq \bigcup_{z_i \in R} \rho_i$, specifying which referenced observations or records to inspect. Textual cues locate candidate episodes; the underlying evidence can contain details absent from those cues.

The agent's update $\mathcal{U}$ revises plans, findings, relational memory, general guidance, or reusable procedures as appropriate. Relations may be expressed as $\ell = (u, r, v; P_\ell)$, where $P_\ell$ references supporting evidence. Consolidation changes interpretations while preserving the episode archive and index; it does not execute environment actions.

# C SKILLS



**Skill C.1:** Shaft down

```
import json
import math
import sys

PARAMS = {"target_y": "target feet height, minimum 12", "pick": "
    pickaxe hotbar slot 1-9", "max_steps": "mining budget, maximum 370
    "}
arguments = json.loads(sys.argv[sys.argv.index("--args") + 1]) if "--
    args" in sys.argv else {}
state = json.load(sys.stdin)
position = state.get("pos")
if not position or state.get("gui_open"):
   raise SystemExit("close the screen and stand on solid ground first"
       )
if abs(position["y"] - round(position["y"])) > 0.05:
   raise SystemExit("not standing at an integer block height")
for coordinate in ("x", "z"):
   fraction = position[coordinate] - math.floor(position[coordinate])
   if not 0.30 <= fraction <= 0.70:
      raise SystemExit("center within the block before descending")
speeds = {"stone_pickaxe": (22, 19), "iron_pickaxe": (18, 15), "
    diamond_pickaxe": (15, 12)}
pickaxes = [entry for entry in state["hotbar"] if entry.get("item") in
     speeds]
if "pick" in arguments:
   pickaxes = [entry for entry in pickaxes if int(entry["slot"]) + 1
       == int(arguments["pick"])]
if not pickaxes:
   raise SystemExit("requested pickaxe is not in the hotbar")
pickaxe = min(pickaxes, key=lambda entry: speeds[entry["item"]][0])
height = int(round(position["y"]))
target = max(12, int(arguments.get("target_y", 12)))
if height <= target:
   raise SystemExit("target depth reached; mine horizontally")
normal_speed, minimum_speed = speeds[pickaxe["item"]]
steps = min(370, max(1, int(arguments.get("max_steps", 300))), (height
     - target) * normal_speed, (height - 11) * minimum_speed)
print(json.dumps({"actions": [
   {"procedure": "equip_hotbar", "args": {"slot": int(pickaxe["slot"])
       + 1}},
   {"procedure": "look_abs", "args": {"pitch": 88, "steps": 8}},
   {"procedure": "mine_ahead", "args": {"n": steps, "ore_watch": True
       }},
   {"procedure": "look_abs", "args": {"pitch": 25, "steps": 8}}
]}))
```



**Skill C.2:** Tunnel

```
"""tunnel: extend a level two-block-high tunnel by up to four cells,
    watching for ore and hazards."""
import json
import sys
```

```
PARAMS = {"cells": "one to four cells", "yaw": "cardinal tunnel
    heading", "pick": "pickaxe hotbar slot 1-9"}
arguments = json.loads(sys.argv[sys.argv.index("--args") + 1]) if "--
    args" in sys.argv else {}
state = json.load(sys.stdin)
position = state.get("pos")
if not position or state.get("gui_open"):
   raise SystemExit("close the screen before tunneling")
if abs(position["y"] - round(position["y"])) > 0.05:
   raise SystemExit("stand on solid ground before tunneling")
if position["y"] < 11:
   raise SystemExit("restore feet to y11 or higher before tunneling")
heading = float(arguments.get("yaw", position["yaw"]))
if abs(heading / 90 - round(heading / 90)) > 0.01:
   raise SystemExit("choose a cardinal heading")
slot = int(arguments.get("pick", 7))
pickaxes = [entry for entry in state["hotbar"] if int(entry["slot"]) +
     1 == slot and entry.get("item") in ("iron_pickaxe", "
    stone_pickaxe", "diamond_pickaxe")]
if not pickaxes:
   raise SystemExit("requested pickaxe is missing")
cells = min(4, max(1, int(arguments.get("cells", 4))))
actions = [{"procedure": "equip_hotbar", "args": {"slot": slot}}]
for cell in range(cells):
   actions.extend([
      {"procedure": "look_abs", "args": {"pitch": 0, "yaw": heading, "
          steps": 8}},
      {"action": {"attack": 1}, "repeat": 30},
      {"procedure": "look_abs", "args": {"pitch": 65, "yaw": heading,
          "steps": 8}},
      {"action": {"attack": 1}, "repeat": 30},
      {"action": {"forward": 1}, "repeat": 12}
   ])
actions.append({"procedure": "look_abs", "args": {"pitch": 0, "yaw":
    heading, "steps": 8}})
print(json.dumps({"actions": actions}))
```

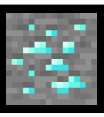


**Skill C.3:** Probe

```
"""probe: mine a stationary three-block side probe at a gentle
    downward angle, watching for ore."""
import json
import sys

PARAMS = {"yaw": "cardinal probe heading", "pick": "pickaxe hotbar
    slot 1-9"}
arguments = json.loads(sys.argv[sys.argv.index("--args") + 1]) if "--
    args" in sys.argv else {}
state = json.load(sys.stdin)
position = state.get("pos")
if not position or state.get("gui_open"):
   raise SystemExit("close the screen before probing")
if position["y"] < 11 or abs(position["y"] - round(position["y"])) >
    0.05:
   raise SystemExit("stand on solid ground at y11 or higher before
       probing")
heading = float(arguments.get("yaw", position["yaw"]))
if abs(heading / 90 - round(heading / 90)) > 0.01:
```

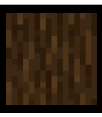
```
    raise SystemExit("choose a cardinal heading")
slot = int(arguments.get("pick", 8))
pickaxes = [entry for entry in state["hotbar"] if int(entry["slot"]) +
     1 == slot and entry.get("item") in ("iron_pickaxe", "
    stone_pickaxe", "diamond_pickaxe")]
if not pickaxes:
    raise SystemExit("requested pickaxe is missing")
actions = [
    {"procedure": "equip_hotbar", "args": {"slot": slot}},
    {"procedure": "look_abs", "args": {"pitch": 15, "yaw": heading, "
        steps": 8}}
]
for block in range(3):
    actions.extend([
        {"action": {"attack": 1}, "repeat": 30},
        {"action": {}, "repeat": 1}
    ])
print(json.dumps({"actions": actions}))
```



**Skill C.4:** Aim speckles

```
"""aim_speckles: turn the crosshair onto the (x, y) pixel a 'mining
    stopped: ...-ore-like speckles' [NOTE] names, then mine that block
     and sweep its drop."""
# Why it exists: the life that found the diamond did exactly this
    within 20 steps of the
# watcher's [NOTE] -- aim at the speckles, mine the block under the
    crosshair, walk to the
# drop. Lives that answered a sighting with "finish the column first"
    or "craft a sword
# first" never mined the ore at all: by the next turn the view had
    moved and the speckles
# were gone. So: a sighting [NOTE] outranks everything; call this from
     its (x, y) BEFORE any
# other look or step, while the view is still the flag-time view (the
    runner dropped the
# rest of the plan for exactly that).
# The frame the watcher reads is 640x360 with a 70-degree vertical
    field of view: the focal
# length is 180 / tan(35 deg) = 257 px, so an offset of d pixels from
    the centre (320, 180) is
# atan(d / 257) degrees -- 104 px right is 22 degrees of yaw, not the
    17 a linear scale gives.
# Two 30-step holds: ore takes longer than stone, and the hold is cut
    the moment the block
# breaks anyway. collect_drops then sweeps the drop; the milestone is
    the PICKUP, not the break.
import json, math, sys
PARAMS = {"x": "screen x from the [NOTE] (0-640)", "y": "screen y from
     the [NOTE] (0-360)",
          "pick": "hotbar slot 1-9 with the pickaxe (default: the best
              in the hotbar)"}
args = json.loads(sys.argv[sys.argv.index("--args") + 1]) if "--args"
    in sys.argv else {}
s = json.load(sys.stdin)
if s.get("gui_open"):
    raise SystemExit("a screen is open: close_inventory first")
if "x" not in args or "y" not in args:
```

```
    raise SystemExit("pass x and y: the pixel the sighting [NOTE] names
        , e.g. (424, 145)")
x, y = float(args["x"]), float(args["y"])
if not (0 <= x <= 640 and 0 <= y <= 360):
    raise SystemExit("x must be 0-640 and y 0-360 (the [NOTE]'s frame)"
        )
rank = {"diamond_pickaxe": 0, "iron_pickaxe": 1, "stone_pickaxe": 2, "
    wooden_pickaxe": 3}
picks = [h for h in s["hotbar"] if h.get("item") in rank]
if args.get("pick") is not None:
    picks = [h for h in picks if int(h["slot"]) + 1 == int(args["pick"
        ])]
if not picks:
    raise SystemExit("no pickaxe in the hotbar (diamond ore needs IRON
        or better to drop a diamond)")
pick = min(picks, key=lambda h: rank[h["item"]])
F = 180.0 / math.tan(math.radians(35.0)) # 257 px
dpitch = round(math.degrees(math.atan((y - 180.0) / F)), 1) # + is
    down, like the screen
dyaw = round(math.degrees(math.atan((x - 320.0) / F)), 1) # + is right
    , like the screen
plan = [{"procedure": "equip_hotbar", "args": {"slot": int(pick["slot"
    ]) + 1}},
      {"action": {"camera": [dpitch, dyaw]}, "repeat": 1},
      {"action": {"attack": 1}, "repeat": 30},
      {"action": {"attack": 1}, "repeat": 30},
      {"procedure": "collect_drops", "args": {"steps": 48}}]
print(json.dumps({"actions": plan}))
```



**Skill C.5:** Spruce Approach

```
First time:
```
Action 1 | Step 1 [ACTION] look(pitch=0,yaw=18) -> 1 ticks
Action 2 | Step 26 [ACTION] {"attack":1,"forward":1} x25
                [STATE] pos=(-3010.70, 71.00, -5569.30) pitch=0 yaw
                    =18 moved=3.42
                [EVENT] mine_block spruce_leaves +1 (x2)
Action 3 | Step 226 [ACTION] mine_ahead(n=200) -> 200 ticks
                [EVENT] mine_block spruce_leaves +1 (x4)
                [NOTE] mine_ahead: finished 200 ticks, broke 4 blocks.

Action 4 | Step 227 [PLAN] OBSERVED vs EXPECTED: Leaves broke, but no
    wood; the exposed trunk is left of the crosshair.
                [ACTION] look(yaw=-28,pitch=0) -> 1 ticks
Action 5 | Step 297 [ACTION] mine_ahead(n=100) -> 100 ticks
                [EVENT] MILESTONE mine_log verified
                [EVENT] mine_block spruce_log +1
```

Second time:
```json
{"actions":[{"procedure":"look_abs","args":{"pitch":0,"yaw":18,"steps"
    :8}},
         {"action":{"attack":1,"forward":1},"repeat":25},
         {"procedure":"mine_ahead","args":{"n":70}},
         {"procedure":"look_abs","args":{"pitch":0,"yaw":-10,"steps"
             :8}},
         {"procedure":"mine_ahead","args":{"n":150}}]
```
```

````
}
```


Third time:
> Spawn spruce approach verified two times: yaw18, attack+forward25,
    mine_ahead70, yaw-10, mine150.
````
````

# D HARNESS AND IMPLEMENTATION DETAILS

This appendix covers five things: the episode index (Section D.1), the spatial grid (Section D.2), the configuration of each method and the action interface shared by all methods (Section D.3), and storage and cost statistics (Section D.4).

## D.1 EPISODE INDEX

The harness appends one row to `index.jsonl` for each completed episode (Box 1). Each row contains:

- the episode identifier and the path to the episode's directory;
- the first and last environment steps of the episode;
- the kind of action executed;
- a one-line summary;
- a list of tags;
- the agent's position $(x, y, z)$.

The summary records the procedure call with its arguments, followed by its outcome: how long it ran, how far the agent moved, how the inventory changed, and any milestone reached. Tags flag notable events such as a milestone, an inventory change, or a failure to move (`stuck`). The harness computes all of these fields from the structured game state; the model does not write them.

**Box 1: Example rows from `index.jsonl` (one Minecraft run).**


```
{"action_kind": "use_at", "first_step": 2318, "last_step": 2370, "id": "ep_000043", "
    kind": "episode", "path": "episodes/ep_000043", "summary": "[ACTION] use_at(x=3168,
    y=64, z=1420, item=water_bucket, face=below, hold=1) -> 53 steps: moved 0.0, inv {'
    bucket': 1, 'water_bucket': -1}, milestone ['log_1']", "tags": ["milestone", "
    inventory_changed", "note"], "x": 3168.66, "y": 64.0, "z": 1419.47}
{"action_kind": "go_to", "first_step": 2444, "last_step": 2703, "id": "ep_000048", "kind
    ": "episode", "path": "episodes/ep_000048", "summary": "[ACTION] go_to(x=3153.4, z
    =1401.4, within=0.8, n=260) -> 260 steps: moved 0.2", "tags": ["stuck", "note"], "x
    ": 3158.30, "y": 57.0, "z": 1400.53}
```


## D.2 SPATIAL GRID

The harness also maintains `grid.jsonl`, an append-only record of the world cells the agent has interacted with. Each row gives a cell's integer coordinates, the step at which it was recorded, and one of five kinds:

- `walked`: the agent stood in the cell;
- `placed`: the agent placed an item in the cell;
- `opened`: the agent opened the screen of a block in the cell;
- `blocked`: the agent could not move into the cell;
- `hazard`: the agent took damage in the cell.

Rows of kind `placed` and `opened` also record the item and the agent's pose at that moment. Rows of kind `hazard` record the amount and type of damage. Below we give one example of each kind.

Example rows from `grid.jsonl`, one for each cell kind.


```
{"cell": [-3010, 71, -5573], "kind": "walked", "step": 1}
{"cell": [-3017, 49, -5569], "item": "furnace", "kind": "placed", "pose": {"pitch": 55.0,
     "x": -3017.7, "y": 49.0, "yaw": 270.0, "z": -5568.37}, "step": 4841}
{"cell": [-3009, 69, -5566], "item": "crafting_table", "kind": "opened", "pose": {"pitch
    ": 55.0, "x": -3007.43, "y": 69.0, "yaw": 90.0, "z": -5565.54}, "step": 1206}
{"cell": [-3008, 61, -5569], "kind": "blocked", "step": 1961}
```

Table 5: **Configuration of each method on Minecraft tasks.** All methods share the action and procedure reference, the knowledge sections, the ground-truth rules, and the per-turn state block; Direct Evaluation omits the block's `[RECENT]` and `[MAP]` fields. All methods can write files and execute stored procedures through the same interface.

| Capability | Direct | PRO-LONG | VISTA | MFM |
|---|---|---|---|---|
| Standing brief | 12 KB | 12 KB | 12 KB | 12 KB |
| Procedure set | 28 atoms | 28 atoms | 28 atoms | 28 atoms |
| Per-turn `[RECENT]` / `[MAP]` | no (as direct agent is not supposed to have any memory) | yes | yes | yes |
| Episode store and query tool | no | no | archived frames via `inspect` | yes |
| Write access to own files; procedure execution | yes | yes | yes | yes |
| Agent-written procedures (observed) | 0 | 0 | 0 | 4 written / 14 calls |
| Plan cap | 400 steps | 400 steps | 400 steps | 400 steps |
| Images per call | current frame | current frame | current frame + inspected frames | current frame + inspected frames |


```
{"cell": [-3019, 34, -5569], "damage": 3.0, "kind": "hazard", "step": 6361, "what": "
    damage"}
```


### D.3 Method Configurations

Table 5 lists the configuration of each method on Minecraft. All methods receive the same 12 KB standing brief in prompt. All methods act through the same 28 atomic procedures (Table 6). The atoms are closed-loop: they read the structured game state at every step. No method has access to task-specific macros such as tree harvesting, strip mining, or portal building.

We reimplemented PRO-LONG and VISTA in our harness without changing their core algorithms. PRO-LONG keeps its programmatic text logs, and VISTA keeps its archive of visual observations and its inspect interface.

**Matched inspection, different addressing.** VISTA and MFM have equivalent visual inspection capabilities: both store every observation losslessly with its step identity, and both let the agent reopen earlier frames, crop and enlarge regions, and compare several past or current views in one context. The methods differ only in how the agent locates evidence and what it does with it. VISTA addresses frames by turn and frame identifier; MFM additionally provides a harness-written index of cues (time, position, action type, tags, outcome summary) that the agent can query, and agent-written memory that cites the supporting episodes. Our comparison with VISTA therefore isolates the contribution of the addressing layer and consolidation, not of the inspection tools.

**Why only MFM writes executable procedures.** Neither baseline provides a persistent place for agent-written procedures: PRO-LONG's durable memory is a single harness-owned log, and VISTA's writable memory is limited to two notes files. Turning an action sequence into a procedure also requires checking whether that sequence has succeeded before. MFM's associative index makes this check direct: the agent can query past episodes by action type and outcome tags, then inspect the linked frames to confirm the result. VISTA addresses frames only by turn identifier, and PRO-LONG's text log, while searchable, has no structured action or outcome fields and no links to visual evidence, so the same check requires scanning and reconstructing past attempts. Without these supports, the baselines kept their procedural knowledge as prose notes rather than callable procedures.

### D.4 Storage and Cost

All main and ablation results are averaged over 3 seeds; each seed fixes the Minecraft world or KiCad task order, with further variation from stochastic model sampling. In total we ran 96 Minecraft runs (72 main + 24 ablation, capped at 12k steps for Diamond and 36k for Dragon) and 1,125 KiCad episodes (25 tasks $\times$ 15 configurations $\times$ 3 seeds). All 48 Dragon runs exhaust their budget (Fig. 8), consuming $\approx$1.7M steps and up to 14 hours each. In the Claude Opus 5 on Dragon configuration, a single run makes 0.9–3.9k model calls and costs \$108–\$969 (Table 8).

Table 7 reports storage, retrieval, and memory-use statistics for one MFM run. Table 8 reports the cost of one Kill the Ender Dragon run per method with Claude Opus 5. Both tables come from single runs and are meant to show orders of magnitude.

Table 6: **The 28 atomic procedures shared by all methods** (`procedures.ATOMS`). They include closed-loop movement and aiming primitives, GUI cursor control, one-call crafting, and placement.

| Atom | Description |
|---|---|
| *Aiming and walking* | |
| `look` | Turn the camera by a relative (pitch, yaw) in degrees, in safe increments. |
| `look_at` | Aim the crosshair at a world coordinate, using the exact position from `[STATE]`. |
| `look_abs` | Steer the camera to an absolute pitch and yaw, closed-loop against `[STATE]` at every step; works from any starting pose. |
| `go_to` | Walk to a world coordinate $(x, z)$, re-aiming against `[STATE]` at every step until within reach. |
| `watch` | Wait up to `n` steps, stopping as soon as a condition read from the game state holds: none (plain wait), inventory (any count changes), gui (a screen opens), health, damage, `y_below:<v>`, `y_above:<v>`, or `moved:<blocks>`. |
| *Mining and pickup* | |
| `mine_ahead` | Hold attack on the block under the crosshair, without re-aiming or walking. |
| `collect_drops` | Walk a short out-and-back sweep to pick up item drops; stop as soon as the inventory count rises, and report whether it did. |
| *Inventory and GUI cursor* | |
| `equip_hotbar` | Select hotbar slot 1–9. |
| `open_inventory` | Press E to open the inventory screen. |
| `close_inventory` | Press E to close the current screen. |
| `cursor_to_slot` | Move the cursor onto a player inventory slot, given its `[INV]` slot ID. |
| `cursor_to_cell` | Move the cursor onto a crafting or furnace cell, or onto the output cell. |
| `gui_click` | Left-click at the cursor (pick up or place a stack). |
| `gui_right_click` | Right-click at the cursor (place one item). |
| `stow_items` | Move the cursor stack and the named crafting cells into empty inventory slots, so the screen can close without dropping anything. |
| `discard_stack` | Throw away the stack in `slot` (there is no drop key): open the inventory, lift the stack, force-close the screen so it lands on the ground, and step back to avoid picking it up again. |
| `drop_and_pickup` | Drop the stack in `slot` and walk over it, so the game registers a pickup. |
| *Crafting and smelting* | |
| `craft` | Craft a recipe from the current inventory in one call: place each ingredient, take the output `count` times, return leftovers, move tools to the hotbar, and verify the result. |
| `place_and_open` | Place a crafting table or furnace on the ground one block ahead and open its screen, moving it to the hotbar first if needed; verify the placement and the screen type. |
| `furnace_load` | Load `n` of an item into the open furnace (0 = the whole stack) along with enough fuel, chosen in the order coal, charcoal, planks, logs, sticks. |
| `furnace_take` | Take the finished output from the open furnace into the inventory, after checking that the output cell is not empty; report what was taken. |
| *Placing and using items* | |
| `place_at` | Place `item` into the cell $(x, y, z)$ by clicking a face of an adjacent block (by default, the block below the cell). |
| `use_at` | Right-click a face of the block at $(x, y, z)$ with `item` selected, e.g., to pour water, light with flint and steel, till soil, plant seeds, or open a door or chest; with no `item`, use the block with whatever is held. |
| `use_held` | Hold use with the selected item, e.g., to draw a bow (release after $\geq$25 steps), drink a potion, look through a spyglass, or raise a shield. |
| `fill_bucket` | Fill an empty bucket from visible water: sweep the crosshair over the ground ahead (pitch 18–60°, yaw $\pm$40°), pressing use at each aim until a filled bucket appears; report which liquid was collected. |
| `throw` | Throw a snowball, egg, ender pearl, or eye of ender `n` times; verify that the item count decreases. |
| `eat` | Eat the named food, or the best food carried; holds use for about 40 steps. |
| `pillar_up` | Jump and place `n` blocks underfoot, verifying each by the measured height; used to build a lookout or escape a mob. |

Table 7: **Storage, retrieval, and memory-use statistics** for the GPT-6-Astra Mine Diamond run that reached all 10 milestones.

| **Quantity** | **Value** |
|---|---|
| Environment steps | 12,000 |
| Model calls | 130 |
| Episodes stored | 636 |
| Episode directory on disk | 46 MB |
| Index (`index.jsonl`) size | 313 KB |
| Full-resolution frames on disk | 720 MB |
| Query time over the full index | 0.3–0.4 s |
| `view_image` calls | 87 |
| Agent-written procedures (written / calls) | 4 / 14 |

Table 8: Costs of Ender Dragon runs with Claude Opus 5 (36,000 steps) using Claude Code.

| **Method** | **Milest. obs.** | **Calls** | **Tokens (M)** | **USD ($)** |
|---|---|---|---|---|
| MFM | 10 | 2,129 | 47 | 108 |
| VISTA | 9 | 2,177 | 69 | 172 |
| PRO-LONG | 8 | 895 | 57 | 168 |
| Direct | 6 | 3,934 | 155 | 969 |

# E BENCHMARK SETUP AND PROMPTS

## E.1 MCU TASKS DESCRIPTIONS

**Mine Diamond from Scratch.** The agent starts with no resources and must progress through the Minecraft technology tree: gathering wood, stone, and iron, crafting increasingly capable tools, and ultimately mining a diamond.

**Kill Ender Dragon.** The agent completes the same early-game progression, then prepares diamond equipment, enters the Nether to obtain blaze materials, crafts Eyes of Ender, reaches the End, and defeats the Ender Dragon.

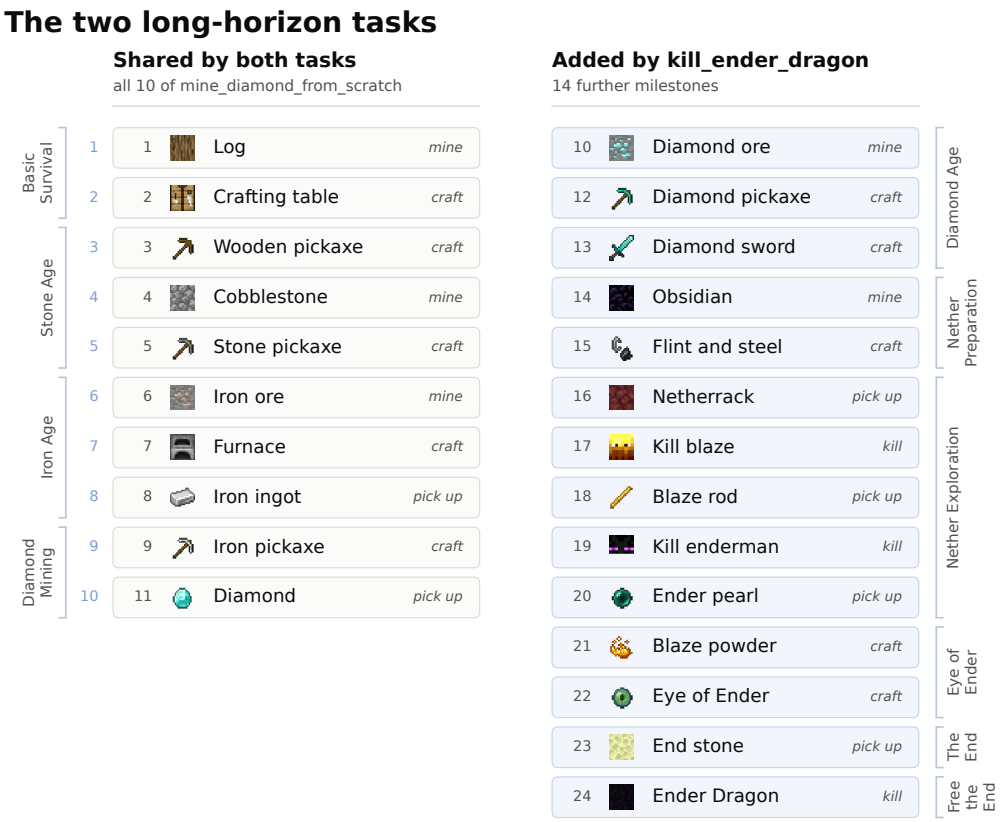



Figure 16: **Milestone progression for the Minecraft tasks.** Mine Diamond from Scratch contains 10 milestones, while Kill Ender Dragon shares the same initial progression and adds 14 milestones required to reach and defeat the Ender Dragon. Numbers in the left column index the Mine Diamond list; inner and right-column numbers index the Ender Dragon list, in which Diamond ore is inserted as milestone 10, so Diamond becomes 11 and the added milestones continue from 12. **Note that the milestones do not need to be finished strictly in this order.**

## E.2 PROMPTS AND OUTPUTS EXAMPLES

In this section, we provide examples of prompts used.

**Box 2: Prompts of the Nether Portal case study**

Build a nether portal and step through it. You are standing on a cleared stone pad and you already carry everything you need, with plenty to spare: 64 obsidian, 64 cobblestone, a flint and steel and a diamond pickaxe. The frame is a vertical rectangle 4 wide and 5 tall lying in one plane, either north-south or east-west. Its interior is 2 blocks wide and 3 blocks tall, and

every one of those 6 interior cells must be empty air. The 10 blocks that matter are 2 below the interior, 2 above it, and 3 up each side; the 4 corners of the rectangle are not part of the portal and may be anything or nothing. Then strike the flint and steel inside the frame, on top of one of the two, and the interior turns purple; walk into it and wait to arrive in the Nether. The only thing this task asks of you is to get the shape right. You have far more obsidian than a portal needs, the cobblestone is there to stand on so you can reach the upper rows, and the diamond pickaxe will take back any block you want to move.

**Box 3: The Memory paragraph of the standing brief.**

**Memory.** ‘./logs.txt‘ and ‘episodes/‘ are the complete record of this life (`./tools/episodes.py --last N | --steps A B | --near X Z | --tag T | --grep text [--sheet out.png] [--then-now <id>] --q "<question>"`); `mine/` is yours and is carried to your next life, and `mine/working.md` is put in front of you every turn. The log knows where; the frame knows what: `view_image` a frame before you walk to a remembered place and, on arrival, `./tools/episodes.py --then-now <that episode> -q "..."` once; look after a hit, and when a plan’s outcome surprised you.

**Box 4: Consolidation prompt.**

**Working memory.** ‘mine/working.md‘ is yours and is put in front of you every turn as ‘[WORKING]‘: the plan you are on and why, its next visible prediction, the contradictions you have not resolved, and the entries or frames you would re-open (paths). It survives a context reset, so the turn after one starts from it instead of from the log. Rewrite it when the plan changes, not every turn; keep it under 40 lines. Your digest or ‘GUIDE.md‘ under ‘mine/‘ is what is true of this world across lives; ‘working.md‘ is what you are doing now.

**Box 5: Minecraft: per-turn observation block.**

Step 8/36000 – 35992 steps remain. Environment-verified progress: 1/24 milestones. Score = COUNT achieved, in ANY order – bank whichever are already within reach (a furnace needs only 8 cobblestone) before chasing the next in list order. First unmet in list order: ‘crafting_table‘ (craft_item of crafting_table).
Act turns since anything verifiably changed (milestone or inventory): 0.
Milestones:
[x] mine_log (step 8)
[ ] crafting_table
[ ] wooden_pickaxe
[STATE] pos=(3.4, 64.0, 2.0) pitch=0 yaw=0
Vitals: health=20/20 food=20/20
Inventory (35 slots free): oak_logx2
Hotbar: hotbar.1=oak_logx2
Held: unknown
Nearby entities: none reported
GUI open: False
[CURSOR] no screen open
[RECENT] the last entries that finished, from the harness’s record:
- step 5 "forward":1 x5: moved 0.0
- step 8 "attack":1 x3: moved 0.9, inv oak_log +1, events MILESTONE mine_log verified; mine_block oak_log +1
[MAP] what the runner knows for certain from this life’s own positions and events (exact unless marked a sighting):
- You stand in cell (3, 64, 2); 3 cells walked this life (a route over them is walkable again).

**Box 6: KiCad: action reference.**

Coordinates are normalized to [0, 1000] on both axes: (0,0) is the top-left of the screen, (1000,1000) the bottom-right — always fractions of the screen times 1000, NEVER raw pixels.
Available actions (each an object in an "actions" list):
"action": "click", "coordinate": [x, y], "button": "left|right|middle", "clicks": 1|2|3
"action": "mouse_move", "coordinate": [x, y]
"action": "drag", "start_coordinate": [x, y], "coordinate": [x, y], "button": "left"
"action": "scroll", "direction": "up|down|left|right", "amount": N, "coordinate": [x, y]
"action": "type", "text": "...", "press_enter": false
"action": "key", "keys": ["ctrl", "s"]
"action": "key_down", "keys": [...] / "action": "key_up", "keys": [...]
"action": "hold_key", "keys": [...], "duration": seconds
"action": "wait", "duration": seconds
"action": "screenshot"

**Box 7: KiCad: task framing and rules.**

You are operating a real Linux desktop through a GUI to complete a task. The desktop is XFCE with KiCad installed; work inside KiCad's own editors (schematic editor etc.) using its menus and keyboard shortcuts (e.g. 'a' to add a symbol, 'w' to wire, ctrl+s to save). The evaluator checks real system state at the end – it compares the saved project's netlist against a reference – so save the project before terminating.
**Rules that have cost episodes before**: - Before typing into any app or field, verify in the CURRENT screenshot that the target window/field exists and is focused – text typed blind goes to whatever is active and is lost.
- Never batch input in the same turn that opens the app or dialog it targets; open, look, then act.
- After typing something long, check the screenshot that it appears complete before pressing enter.
- Save the project (ctrl+s, confirm any dialog) and verify the title bar no longer shows an unsaved marker before terminating. Terminate as soon as the task is durably done; every further step is budget, not progress.
- If the screen has not changed after the same action twice, the action is not doing what you think: zoom in, re-read the log, and change approach.