# AUTOMEM: Automated Learning of Memory as a Cognitive Skill

**Shengguang Wu** **Hao Zhu** **Yuhui Zhang** **Xiaohan Wang** **Serena Yeung-Levy**
Stanford University

https://autolearnmem.github.io/

## Abstract

Memory expertise is a learned skill: knowing what to encode, when to retrieve, and how to organize knowledge—a capacity known in cognitive science as *metamemory*. We bring this perspective to LLMs by treating memory management as a trainable skill. We promote file-system operations to first-class memory actions alongside task actions, letting the model itself decide how to manage its memory. This memory skill improves along two axes: the ***structure*** that supports it (prompts, file schemas, action vocabulary), and the ***proficiency*** of the model exercising it. Both axes resist manual optimization: episodes in long-horizon tasks run for thousands of steps, and a single memory mistake can hide long before it surfaces, making human review of full trajectories impractical. We introduce AUTOMEM, a framework that **automates** both axes. In the first loop, a strong LLM reviews complete agent trajectories and iteratively revises the memory *structure* that shapes how the agent interacts with its memory files. In the second loop, the agent's own good memory decisions are identified from many episodes and used as training signal to sharpen the model's memory *proficiency* directly. Across three procedurally generated long-horizon games (Crafter, MiniHack, and NetHack), optimizing memory alone—without modifying the model's task-action behavior—improved the base agent's performance ∼2×–4×, bringing a 32B open-weight model competitive with frontier systems such as Claude Opus 4.5 and Gemini 3.1 Pro Thinking. Our results show that memory management is an independently learnable skill, and a high-leverage objective yielding large gains on long-horizon tasks.

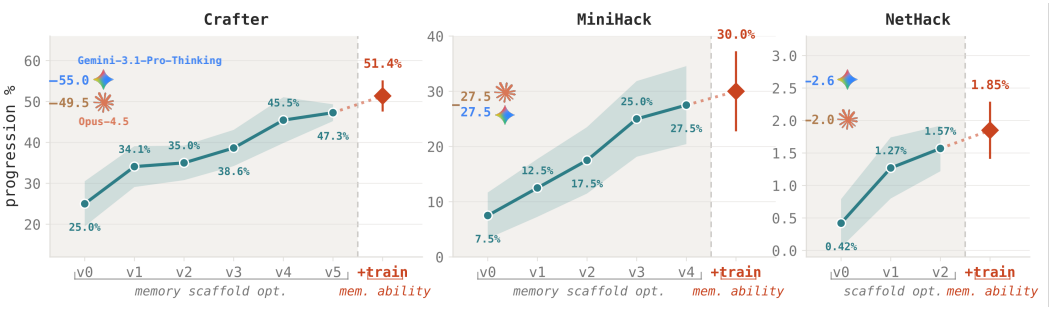


Figure 1: **Memory skill optimization** with `Qwen2.5-32B-Instruct`. Starting from a base agent equipped with file-system memory (v0), AUTOMEM progressively improves performance through *memory scaffold optimization* (v0–v5/v4/v2), followed by *memory proficiency training* (+train) that yields further gains on top of the optimized scaffold.

# 1 Introduction

Humans routinely manage information beyond what can be held in mind at any one moment. Cognitive scientists call this capacity ***metamemory***: the learned skill of deciding what is worth remembering, when to retrieve it, and how to organize what is known [Flavell, 1979, Nelson, 1990]. Metamemory develops with practice, and skilled use of external aids—notes, indices, files—is part of how people extend cognition beyond working memory [Clark and Chalmers, 1998].

LLMs face an analogous bottleneck. Their context window plays the role of working memory, *i.e.,* a fixed-size buffer that bounds what the model can attend to at once. Long-horizon tasks routinely exceed this capacity, and external memory has been explored in various forms, including retrieval databases, vector stores, scratchpads, and summary buffers [Lewis et al., 2020, Packer et al., 2023, Park et al., 2023, Xu et al., 2025, Sumers et al., 2023, Zhang et al., 2024b, Zhong et al., 2024]. These approaches typically treat memory as an architectural module: a fixed mechanism designed into the system. We take a different view, one inspired by *metamemory*: **memory management is an active, trainable skill**, and the model itself decides what to store, what to look up, and how to structure its records.

Concretely, we promote file-system operations (read, write, search, append, create) as *first-class memory actions* in the model's action space, on equal footing with the actions it uses to act on the world [Yao et al., 2022]. The same forward pass that picks a task action can also select memory file operations (*e.g.,* `<|APPEND|>` or `<|SEARCH|>`). This minimal design gives the model full control over its external memory while keeping behavior cleanly observable: every memory decision is a traceable action in the trajectory.

A learned memory skill improves along two axes. First, there is the ***structure*** that supports it (the prompts, the file schema, the validation logic, the action vocabulary), which determines what memory operations are available and how the model is guided to use them. Second, there is the ***proficiency*** of the agent exercising the skill—the model's parametric ability to decide well among the available operations.

Both axes resist manual optimization. A single episode can run for tens of thousands of steps, and the effect of a memory operation (or a mistake) may remain hidden for many steps before it surfaces as a guiding signal or a missed objective. The learning of memory skill in long-horizon tasks is therefore almost intractable for human review.

The key observation behind our approach is that a sufficiently capable LLM—acting as a *meta-LLM*—can review an agent's complete episode (spanning thousands of steps) and identify where memory decisions went wrong, much as a code reviewer would read a full execution log. This makes it possible to **automate** both axes of memory improvement. We introduce AUTOMEM (Figure 3), which does so through two sequential outer loops that operate on a shared inner-loop agent using a file system as its memory.

In the first loop (***structure***), a meta-LLM reads complete episode traces, diagnoses failure patterns in the agent's memory use, and iteratively revises the agent scaffold: the code, prompts, and memory file schema that shape how the agent interacts with memory and acts on the world.

In the second loop (***proficiency***), the agent's own memory decisions from many episodes are reviewed and the ones worth reinforcing are selected—again by a meta-LLM—as supervised training data for a dedicated memory model. The same meta-LLM also orchestrates the finetuning configuration that absorbs this data. Because we treat memory as a separable skill, we finetune only a dedicated memory model (*memory specialist*) while the model that commits world actions remains unmodified, thus sharpening memory proficiency without risking the agent's existing task competence.

A common principle connects both loops: **long-horizon task improvement can decompose into trajectory-level review and targeted revision**—a workflow that a strong meta-LLM can execute autonomously where human review of complete traces (up to $10^5$ steps) is impractical. The first loop applies this principle to revise code; the second applies it to curate training data and orchestrate the training itself.

We evaluate on procedurally generated long-horizon games, which are well suited for studying memory skill: episodes are long enough that context-window management alone cannot sustain performance; worlds are regenerated each episode, so pretraining knowledge transfers poorly [Paglieri

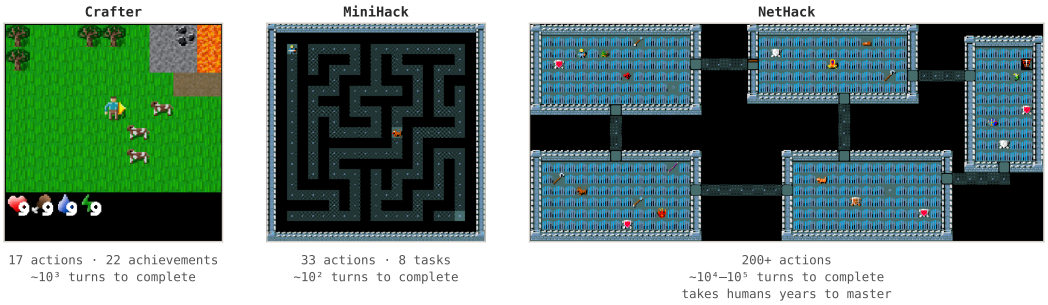


Figure 2: **Long-horizon game environments for evaluating memory skills.** All three environments are stochastic worlds, making each episode unique and minimizing the influence of pretraining knowledge. **Crafter** is an open-world survival game with crafting, combat, and resource management. **MiniHack** presents focused puzzle, navigation and combat tasks within the NetHack engine. **NetHack** is among the most complex games: episodes span $10^4$–$10^5$ turns with a vast exploration space, taking human players typically years to master.

et al., 2024]; and success demands the kind of records humans naturally keep—such as maps, inventories, encounter logs, strategy notes. We choose three environments spanning span a range of complexity (Figure 2): **Crafter**, an open-world survival game with crafting and resource management [Hafner, 2021]; **MiniHack**, a suite of focused puzzle, navigation and combat tasks [Samvelyan et al., 2021]; and **NetHack**, a roguelike whose $10^4$–$10^5$ step episodes take human players years to master [Küttler et al., 2020].

Using `Qwen2.5-32B-Instruct` as the base model, optimizing memory alone—without modifying the model's task-action weights—the full AUTOMEM framework yields $\sim$2×–4× gains over the base agent (Table 1, Figure 1). The optimized 32B agent outperforms `Qwen2.5-72B-Instruct` on all three games by a wide margin, indicating that **well-managed memory is higher-leverage than model scale** on these tasks. It also reaches the performance level of frontier proprietary systems such as Claude Opus 4.5 and Gemini 3.1 Pro Thinking—showing that the gap between open-weight and frontier models on long-horizon tasks can be substantially closed by targeting memory as the optimization objective.

**Contributions.** *(i)* We formulate memory management as an independently learnable skill for LLM agents, instantiated through file-system operations that sit in the same action space as task actions, giving the model full, observable control over what to encode, when to retrieve, and how to organize its memory. *(ii)* We introduce AUTOMEM, a framework that automates memory skill improvement along two complementary axes: scaffold revision that iterates on the agent's memory structure, and targeted training of the model's memory proficiency on its own experience. Both loops are driven by meta-LLMs that analyze complete episode traces, making long-horizon optimization feasible where human review of full trajectories is not. *(iii)* Across three procedurally generated long-horizon games, targeting memory yields $\sim$2×–4× progression gains for a 32B open-weight model, significantly closing the gap with frontier proprietary systems, and demonstrating that memory is a high-leverage objective for long-horizon tasks.

## 2 AUTOMEM

Memory skill, as formulated above, improves along two axes—*structure* and *proficiency*—both of which require automation to optimize on long-horizon tasks. AUTOMEM provides this automation through two sequential outer loops that operate on a shared inner-loop agent (Figure 3). The first loop pushes the agent scaffold as far as code revision can take it; the second trains the model's memory ability beyond the ceiling of any fixed scaffold.

### 2.1 Inner-loop agent: memory as file system

The inner loop is a single LLM agent executing one episode of a long-horizon task, equipped with a directory of files on disk that serves as its external memory (gray-shaded area in the middle of

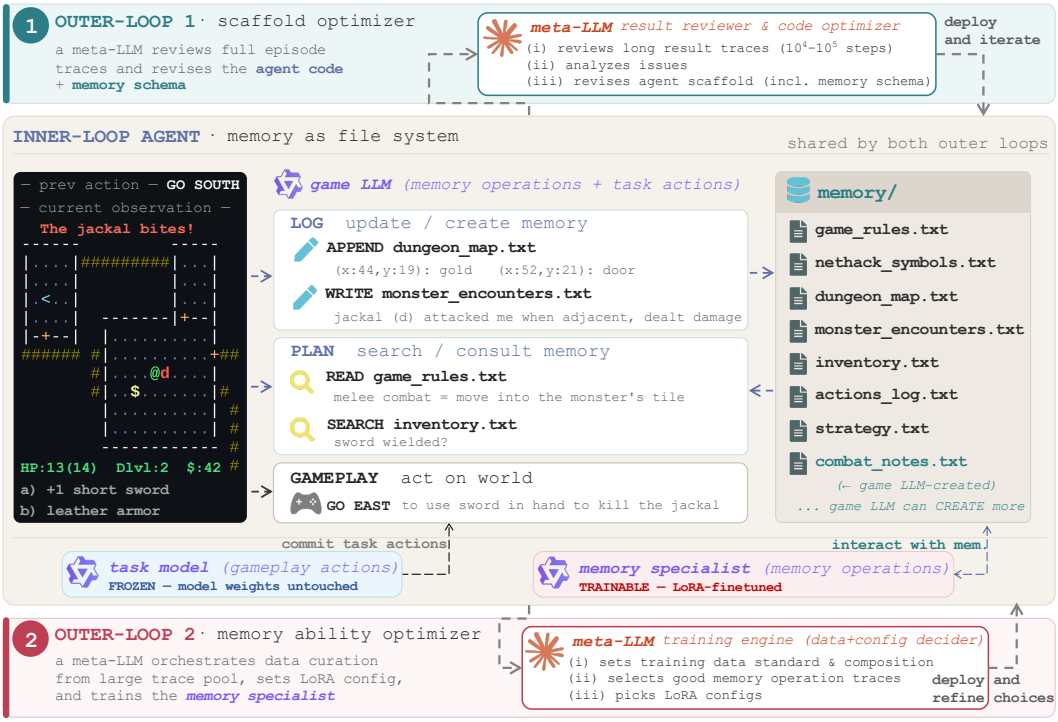


Figure 3: **Overview of AUTOMEM.** Two automated outer loops optimize a shared inner-loop agent that uses the file system as its memory. **Outer-loop #1** (top): a meta-LLM reviews full episode traces and iteratively revises the agent scaffold. **Outer-loop #2** (bottom): a meta-LLM *training engine* jointly orchestrates data curation and finetuning configuration to train a dedicated *memory specialist* that handles memory operations, while the *task model* (frozen, unmodified) commits task actions. The two loops are complementary: loop #1 produces an optimized scaffold within which loop #2 trains the model to interact with its memory more effectively.

Figure 3). At each step the agent runs two routines, each targeting one side of memory management. The LOG routine asks "***what is worth recording about what just happened***": the agent decides whether and how to record the environment's response to the previous action, *e.g.,* appending to an existing file, creating a new one, or rewriting an entry. The PLAN routine asks "***what do I need to recall to act now***": the agent searches across files, reads specific entries or their tails, and commits the next world action.

This unified action space is what makes memory a learnable skill rather than a fixed mechanism. The file-system substrate gives the model a wide decision space—which files to keep, what to record in each, when to consult them, how to organize what is known—and because every memory decision is a traceable action in the trajectory, the outer loops can observe, evaluate, and optimize it.

As a result of this shared space, borne out in our experiments (Section 3.2), **optimizing the memory structure also improves task behavior** (*e.g.,* gameplay actions): better-organized memory reduces redundant exploration and directionless action, even though the optimizer targets the memory scaffold rather than the task strategy directly.

### 2.2 Outer-loop 1: optimizing the memory scaffold

The first outer loop optimizes the *structure* that supports the memory skill. The agent scaffold, *i.e.,* the code, prompts, file schema, and action vocabulary that shape how the agent manages memory and acts on the world, is iteratively revised by the meta-LLM.

The optimization signal must be trajectory-level, because the consequences of memory decisions are often delayed in long-horizon tasks. A memory mistake at step 50—failing to record a map coordinate, or writing a duplicate entry that buries useful information—may not surface until much

later (*e.g.,* step 800), when the agent gets lost or wastes time re-exploring. Final-return metrics alone discard the trajectory structure that reveals *where* memory went wrong.

The meta-LLM is therefore given full episode traces (per-step logs, the resulting memory directories, and the agent code itself) and identifies points where the scaffold caused failures. It functions as a code reviewer with a complete execution log in hand, not as a scalar reward signal. For example, reviewing NetHack traces, the meta-LLM identified that an unbounded map file was accumulating thousands of duplicate coordinate entries, burying useful information; it responded by introducing a clean coordinate-keyed map deduplication format (Figure 5) that sharply shrank the map the agent must carry.

Each iteration is gated on measured improvement: the rewritten agent plays the same fixed seeds as the previous version, and the revision is kept only if average progression improves. Details about the gate and retry mechanics are in Appendix A.2. At rough "convergence" (in practice, after *e.g.,* 2–5 iterations as shown in Figure 1), the scaffold has absorbed what code revision can express. Concrete revisions the optimizer produced across iterations are discussed in Section 3.2 and listed in Appendix B. Figure 5 illustrates how the memory file schema evolved. Given an agent scaffold that is already lean and well-structured, the model's parametric ability to navigate its memory becomes the remaining bottleneck blocking optimal memory decisions.

### 2.3 Outer-loop 2: training memory proficiency

Once the scaffold is optimized, the remaining gap lies in the model's parametric ability to make good memory decisions—the *proficiency* axis. Where the first loop revised code with the meta-LLM as reviewer, this loop updates model weights with the meta-LLM as a *training engine*—a meta-level optimizer that orchestrates the supervised training process end-to-end.

The memory ability is trained on the model's own experience: the meta-LLM reads the inner-loop agent code together with a pool of episode traces from many random episodes, derives selection criteria from what the agent code requires of the model, and produces supervised training data. Every example in the training set is verbatim text the inner-loop model produced during an episode; the meta-LLM's role is to select which responses to reinforce—it thus acts as a ***filter*** on the model's own behavior rather than as a ***teacher*** generating new responses.

High-quality data curation is essential, but the training must be **configured** to absorb it. There are many valid ways to curate good memory traces from the same trace pool (*e.g.,* different selection criteria, sample sizes). Under our LoRA setup—a small adapter finetuned on a modest dataset—a configuration that suits one dataset can underfit or overfit under another, so each data curation calls for its own matching LoRA finetuning configuration to learn effectively. The meta-LLM *training engine* therefore orchestrates the data selection logic, the data composition, and the LoRA training configuration as a **joint decision**, refining all three across iterative trials with concrete eval-trajectory feedback. The point is to find a properly matched (data, configuration) combination so that the finetuning contributes a clean lift of memory capability rather than unwanted disruption.

Because memory is a separable skill, we train it as a distinct target rather than finetuning on full episodes that mix memory operations with action-format outputs. This separation carries through to inference deployment. The inner loop runs two model instances that share a single conversation history (lower panel of Figure 3). The *memory specialist* is the LoRA-finetuned [Hu et al., 2022] copy and handles the LOG routine together with the memory-consultation portion of the PLAN routine. The *gameplay model* is the unmodified base and commits the world action. After the specialist's last memory operation, a short handoff passes the conversation to the gameplay model, which can still issue further memory reads before committing.

Key benefits follow from this split. First, **the training signal stays focused**: the supervised loss targets memory-operation behavior without being diluted by action-format examples, so each gradient step addresses memory proficiency directly. Second, **the base model's competence at producing well-formatted world actions is fully preserved**, since the gameplay model's weights are never touched by finetuning. In practice, this means the memory-proficiency gain stacks cleanly on top of the scaffold gain (a further $\sim$9–18% relative gain, see Section 3.2) rather than trading off against it.

**Two loops, one objective.** Combining both loops together, AUTOMEM instantiates a coherent process for **learning memory as a skill**, one that mirrors the familiar structure of machine learning.

Table 1: **Performance on long-horizon games from BALROG.** AUTOMEM is built on `Qwen2.5-32B-Instruct`. Frontier numbers are taken from the BALROG leaderboard [Paglieri et al., 2024]; baseline rows for `Qwen2.5-32B-Instruct` use the same BALROG harness with the listed context management method. *Scaffold opt.* refers to the agent produced by Outer-loop #1 at convergence (Crafter at `v5`, MiniHack at `v4`, NetHack at `v2`). + *memory training* adds Outer-loop #2's training engine on top. All values are progression rate (%), reported as mean ± standard error.

| Agent | Crafter (%) | MiniHack (%) | NetHack (%) |
|---|---|---|---|
| *Frontier proprietary (BALROG leaderboard)* | | | |
| Gemini-3-Pro | $57.3 \pm 4.4$ | $40.0 \pm 7.7$ | $6.8 \pm 3.2$ |
| Gemini-3.1-Pro-Thinking | $55.0 \pm 6.4$ | $27.5 \pm 7.1$ | $2.6 \pm 0.3$ |
| Claude-Opus-4.5 | $49.5 \pm 3.1$ | $27.5 \pm 7.1$ | $2.0 \pm 0.5$ |
| Gemini-2.5-Pro | $55.0 \pm 6.0$ | $17.5 \pm 6.0$ | $1.7 \pm 0.2$ |
| *Open-weight (BALROG leaderboard)* | | | |
| DeepSeek-R1 | $36.4 \pm 3.8$ | $25.0 \pm 6.8$ | $1.4 \pm 0.5$ |
| Qwen2.5-72B-Instruct | $27.3 \pm 3.6$ | $5.0 \pm 3.4$ | $0.3 \pm 0.3$ |
| Qwen2.5-7B-Instruct | $16.4 \pm 3.0$ | $0.0 \pm 0.0$ | $0.0 \pm 0.0$ |
| `Qwen2.5-32B-Instruct` *with basic context-management baselines* | | | |
| sliding window | $19.55 \pm 3.46$ | $2.50 \pm 2.47$ | $0.00 \pm 0.00$ |
| + chain-of-thought | $17.27 \pm 2.71$ | $10.00 \pm 4.74$ | $0.00 \pm 0.00$ |
| `Qwen2.5-32B-Instruct` *with* AUTOMEM *(ours)* | | | |
| memory-as-file-system, `v0` | $25.00 \pm 5.50$ | $7.50 \pm 4.16$ | $0.42 \pm 0.37$ |
| + scaffold opt. (loop #1) | $47.27 \pm 2.05$ | $27.50 \pm 7.06$ | $1.57 \pm 0.35$ |
| + memory training (loop #2) | **51.36** $\pm 3.81$ | **30.00** $\pm 7.25$ | **1.85** $\pm 0.44$ |

Each loop has *parameters* $\theta$ to optimize and an *update signal* $\nabla L$ derived from the meta-LLM's trajectory analysis—but in the first loop $\theta$ is the agent scaffold and $\nabla L$ is a code revision, while in the second, $\theta$ is the dedicated memory model weights and $\nabla L$ is a properly-configured supervised training step (curated data together with a matched LoRA setup) drawn from the agent's own behavior. The scaffold sets the structural ceiling for what memory operations are possible, and proficiency training then pushes the model toward that ceiling.

# 3 Experiments

## 3.1 Setup

**Environments.** We evaluate on three procedurally generated long-horizon games used in BALROG [Paglieri et al., 2024] (Figure 2). **Crafter** [Hafner, 2021] is a 2D survival world with 17 actions and 22 achievements covering exploration, crafting, and combat; episodes run up to $\sim 10^3$ steps. **MiniHack** [Samvelyan et al., 2021] is an 8-task suite (mazes, corridors, Boxoban puzzles, quests) built on the NetHack engine, with 33 actions and $\sim 10^2$ steps per task. **NetHack** [Küttler et al., 2020] is the full NetHack Learning Environment (NLE), with 200+ actions and $10^4$–$10^5$ steps per episode; the dungeon, monsters, and items are regenerated every seed. The BALROG harness is used as released, with minor configuration changes documented in Appendix A.1.

**Metric.** We use game *progression rate* following BALROG, scaled to $[0, 100]$. For Crafter it is the fraction of the 22 achievements obtained in an episode. For MiniHack it is the fraction of the 8 task variants completed (binary per task). For NetHack it is the dungeon-and-experience-level progression metric introduced by Paglieri et al. [2024]. We report mean and standard error over a fixed list of 10 seeds, $[42, 43, \ldots, 51]$, paired with per-environment episode counts: 10 episodes for Crafter, 5 per task for MiniHack ($5 \times 8 = 40$ episodes), and 5 for NetHack, matching the BALROG defaults.

**Base model and meta-LLM.** The inner-loop model is `Qwen2.5-32B-Instruct`, served locally with vLLM. The scaffold optimizer (Outer-loop #1) and the training engine (Outer-loop #2) call Claude Opus 4.6 & Opus 4.7 respectively as the meta-LLM. In outer-loop #2, the *memory specialist* is

trained with LoRA; the gameplay model is the unmodified base. Hyperparameters, prompt templates and more implementation details are in Appendix A.

**Baselines.** Two groups. (i) BALROG leaderboard: frontier proprietary models (Gemini-3-Pro, Gemini-3.1-Pro-Thinking, Claude-Opus-4.5, Gemini-2.5-Pro), frontier open-weight reasoning models (DeepSeek-R1, 671B), and same-family open weights at two scales (Qwen2.5-72B-Instruct, Qwen2.5-7B-Instruct). (ii) `Qwen2.5-32B-Instruct` under the BALROG harness with basic context-management methods: a sliding window of 16-step recent observations with/without chain-of-thought [Wei et al., 2022].

### 3.2 Results

**Memory management is a high-leverage axis on its own.** The model weights are untouched throughout scaffold optimization—only the agent code, prompts, and file schema change—yet progression **roughly doubles or more than triples on every environment**: Crafter 25.0%→47.27% (×1.89), MiniHack 7.5%→27.5% (×3.67), NetHack 0.42%→1.57% (×3.74) (Table 1, Figure 1).

To calibrate this: the scaffolded 32B agent outperforms Qwen2.5-72B-Instruct (under the BALROG harness) by a wide margin, and surpasses the same base model under basic sliding-window context strategy by an even wider one, indicating that well-structured external memory management is a **higher-leverage axis than model scale** or context management on these long-horizon tasks.

**Training the memory specialist adds complementary lift on top of the optimized scaffold.** On top of the optimized scaffold (`v5` for Crafter, `v4` for MiniHack, `v2` for NetHack), the memory-proficiency training loop lifts progression to 51.36% on Crafter (+4.09), 30.0% on MiniHack (+2.5), and 1.85% on NetHack (+0.28), comparable in magnitude to one or two scaffold iterations (Table 1).

The two optimization axes thus work in sync: structure-revision (outer-loop 1) pushes the ceiling of prompts and agent schema, and proficiency-training (outer-loop 2) addresses the remaining model-capacity gap. Together they bring the 32B open-weight model to the performance level of frontier proprietary systems on these tasks, comparable to Claude-Opus-4.5 (Crafter/MiniHack/NetHack: 49.5/27.5/2.0) and within a few points of Gemini-3.1-Pro-Thinking (55.0/27.5/2.6).

**Task decisions and memory operations are simultaneously improved in the shared action space.** Because memory operations and task actions share the same action space, both are observable in full episode traces, and the meta-LLM's trajectory-level review surfaces failures on both sides. We measure four behavioral indicators to quantify the effect of scaffold optimization (Figure 4).

On the task action (*i.e.,* gameplay) side, we track the ***unproductive action rate***: the fraction of steps where the agent is either **stuck** (observation unchanged from the previous step—the action had no effect) or **oscillating** (≥3 direction reversals in the last 10 movements—pacing back and forth without spatial memory of where it has been). This combined rate drops 32–65% across all three

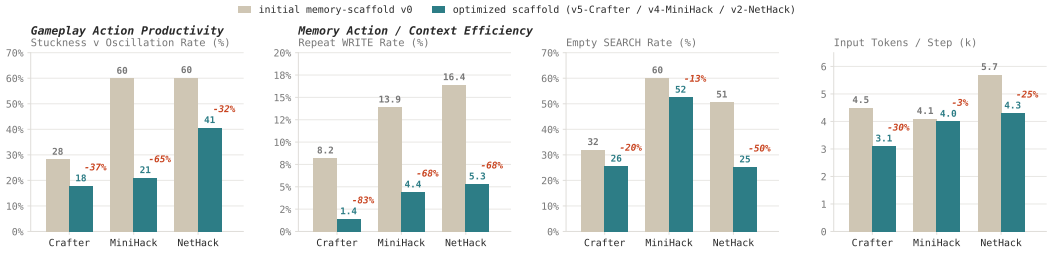


Figure 4: **Effect of scaffold optimization on gameplay and memory behavior** (v0 → v5 for Crafter, v0 → v4 for MiniHack, v0 → v2 for NetHack). *Left:* the unproductive game action rate (fraction of steps that are either stuck or oscillating) drops 32–65% across all three environments. *Right three panels:* **memory operations become more efficient and better targeted**—redundant writes drop sharply (−68 to −83%), the empty-search rate (memory `SEARCH`es returning nothing) falls (−13 to −50%), and per-step input context shrinks (−3 to −30%) as leaner memory compresses what the model must attend to. All values are v0 vs. final scaffold version; lower is better in every panel.

environments. This reduction recovers steps that the base agent spent repeating ineffective actions; each recovered step is an opportunity to explore, gather, or craft that was previously wasted.

On the memory side, the optimized scaffold makes memory operations substantially more efficient and better targeted. *Redundant memory-file writes* drop sharply ($-68$ to $-83\%$) and the *empty-search rate*—memory `SEARCH`es that return nothing—falls ($-13$ to $-50\%$), so the agent both **writes less duplicate content** and **retrieves more precisely**. Especially on Crafter and NetHack, the leaner memory also compresses the *input context* per step ($-25$ to $-30\%$ tokens). This precisely echoes the purpose of external memory, *i.e.,* **compressing the information** the model needs to attend to. The optimized scaffold leads to the emergence of such effective compression because the meta-LLM's trajectory review surfaces the consequences of verbose or redundant memory on the actual task environments—a key benefit of memory operations and task decisions sitting in the shared action space.

Figure 5 illustrates some concrete memory schema changes that drive these numbers, using NetHack as an example. More details on per-iteration changes for all three environments are listed in Appendix B. The base scaffold (v0) uses an append-only `dungeon_map.txt` that grows without bound, frequently accumulating duplicate coordinate entries (the same tile logged multiple times as the agent revisits it). The optimizer introduces a dedicated `<|UPSERT_MAP|>` operation that replaces append-based map logging with coordinate-keyed deduplication, so any new observation of tile $(x, y)$ overwrites the previous entry rather than appending alongside it. The evolved schema also adds auto-synced `inventory` and `status` files that the scaffold updates programmatically from the observation, removing the need for the model to manually `READ` and reconcile its own records, and a pre-populated `strategy` reference that encodes the game's primary objective (find stairs and descend) so the model does not waste early-episode operations rediscovering the goal. Together these changes sharply shrink the memory the agent must carry: its per-step growth falls from 138 to 6 characters, a 95% reduction.

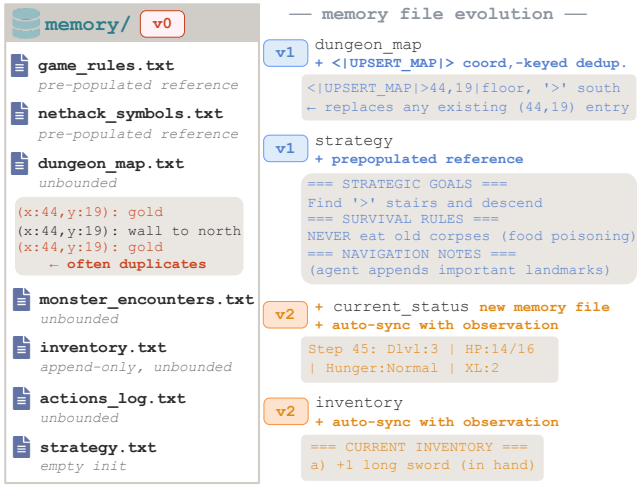


Figure 5: **Example memory file evolution (NetHack).** The base schema (v0) logs to an unbounded, append-only `dungeon_map.txt` that accumulates duplicate coordinate entries. Scaffold optimization replaces this with a coordinate-keyed `<|UPSERT_MAP|>` deduplication format, adds auto-synced `inventory` and `status` files maintained from observation, and pre-populates a `strategy` reference.

**Training internalizes a consult-before-write memory discipline.** Beyond the aggregate progression lift, the trained memory specialist exhibits a consistent shift toward **consulting memory before modifying it** (Table 2). In the LOG phase, the ratio of memory `writes` to `SEARCH`es falls in every environment: Crafter $0.84 \rightarrow 0.39$ ($-54\%$), MiniHack $2.89 \rightarrow 0.82$ ($-72\%$), and NetHack $4.66 \rightarrow 1.31$ ($-72\%$). That is, the specialist searches existing files more before appending new content, rather than writing blindly. This is precisely the consult-before-write pattern the optimized scaffold encourages via prompting, now **internalized** into the dedicated memory model weights.

Table 2: LOG-phase memory `writes` per `SEARCH` on the *evolved* scaffold with base model vs. **+ trained** memory specialist (lower = more retrieval before writing). The trained specialist checks memory before writing in every environment.

| | Base | + Trained |
|---|---|---|
| Crafter | 0.84 | 0.39 ($-54\%$) |
| MiniHack | 2.89 | 0.82 ($-72\%$) |
| NetHack | 4.66 | 1.31 ($-72\%$) |

**Qualitative examples.** Figure 6 traces, for each environment, a representative episode at each optimization stage, illustrating memory's effect on behavior. In Crafter, the base `v0` agent loops on

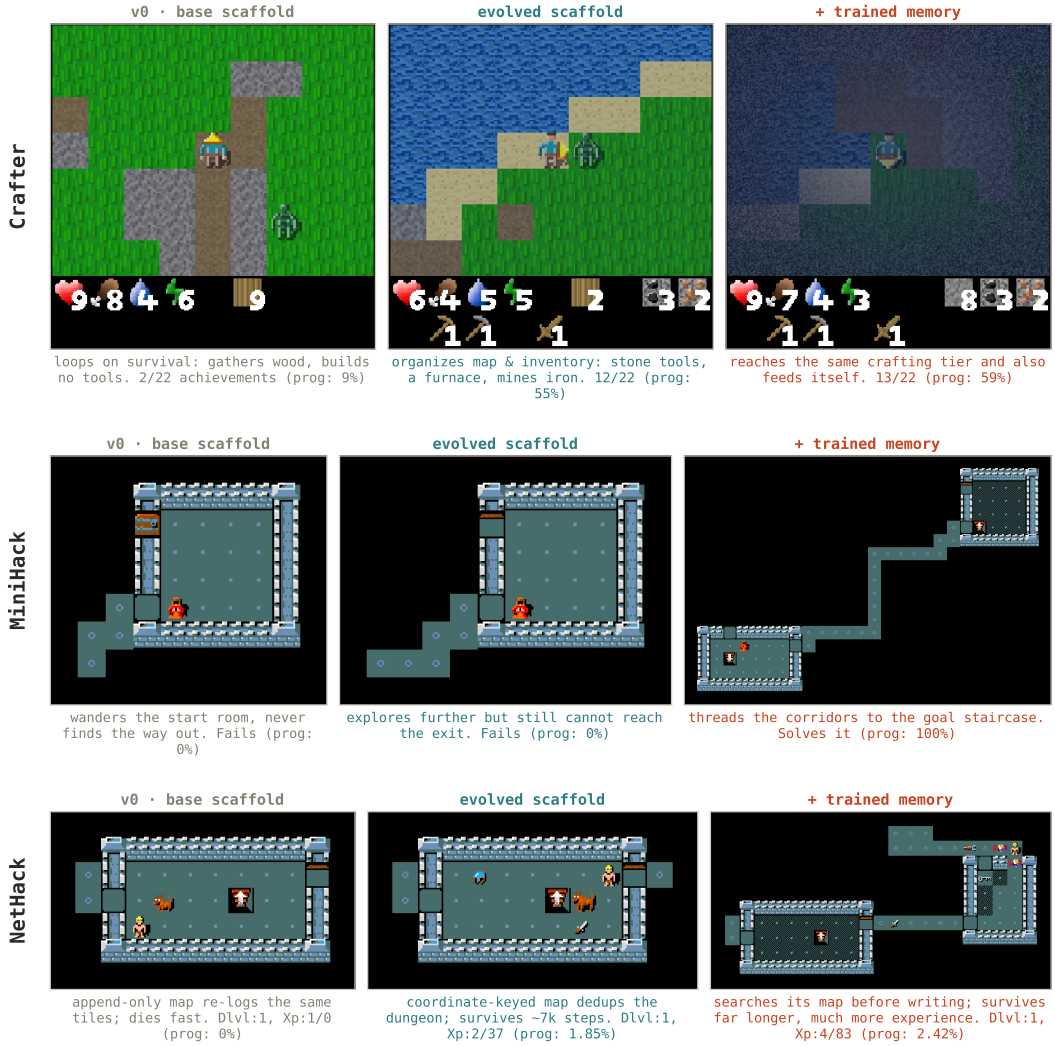


Figure 6: **Qualitative behavior across the two optimization stages.** Each row is an environment; its three cells show the base scaffold `v0`, the evolved scaffold, and the + trained specialist on a representative evaluation episode. The note under each cell reports the agent's behavior and its progression rate (*prog.*); for NetHack it also lists the dungeon level (`Dlvl`, the agent's depth) and the experience level and points (`Xp`). **Crafter**: the base agent only gathers wood; the evolved scaffold crafts stone tools, builds a furnace, and mines iron; and the trained specialist reaches that same crafting tier while also feeding itself ($9 \rightarrow 55 \rightarrow 59\%$ progression). **MiniHack** Corridor-R3, a task that requires navigating branching corridors to reach a goal staircase: neither the base nor the evolved scaffold reaches the staircase, whereas the trained specialist solves the task ($0 \rightarrow 0 \rightarrow 100\%$). **NetHack**: the base agent dies at experience level 1 within a few hundred steps; the evolved scaffold survives thousands of steps and reaches experience level 2; and the trained specialist survives far longer and reaches experience level 4 ($0 \rightarrow 1.85 \rightarrow 2.42\%$ progression).

basic survival. The evolved scaffold organizes its map and inventory well enough to craft stone tools and a furnace and to mine iron, and the trained specialist reaches the same crafting tier while also sustaining itself with food. In MiniHack's Corridor-R3, a task that requires navigating branching corridors to reach a goal staircase, both the base and the evolved scaffold exhaust the step budget without reaching it, whereas the trained specialist threads the corridors and solves the task. NetHack is the sharpest illustration of the delayed-consequence problem. With an append-only map, the base agent re-logs redundant observations and never builds on them, dying at experience level 1 within a few hundred steps. The evolved scaffold's coordinate-keyed map (Figure 5) lets it survive thousands of steps and reach experience level 2. The trained specialist, which searches its map before writing rather than logging blindly, survives far longer still and climbs to experience level 4. These traces make concrete what the aggregate metrics summarize: improving memory management alone contributes significantly to reshaping the agent's task behavior.

## 4 Related Work

**External memory for language models.** The fixed-size context window bounds an LLM's working-memory buffer, and motivates a range of external memory designs. Retrieval-augmented generation [Lewis et al., 2020] couples a retriever with a language model, fetching relevant passages from a document store at inference time; MemGPT [Packer et al., 2023] pages information in and out of context using OS-inspired memory management; Generative Agents [Park et al., 2023] maintain a timestamped memory stream from which they retrieve, reflect, and plan; A-MEM [Xu et al., 2025] equips LLMs with active decisions on what to retain and forget; MemoryBank [Zhong et al., 2024] maintains a persistent long-term store across sessions. MemLLM [Modarressi et al., 2024] trains models to interact with a dedicated read-write memory module, and Self-Notes [Lanchantin et al., 2023] trains models to interleave reasoning with memory tokens—both closest to our *proficiency* axis. More recent works promote memory operations to **learnable policy actions** and train models to decide when to store, retrieve, or discard [Yu et al., 2026, Zhou et al., 2025, Zhang et al., 2026b, Yuan et al., 2025, Yan et al., 2025]. MemAct [Zhang et al., 2025b] shares our "memory as action" insight but operates on the context window itself rather than on external files. MemSkill [Zhang et al., 2026a] also frames memory operations as evolvable skills whose repertoire grows by reviewing failure cases. MeMo [Quek et al., 2026] trains a separate, dedicated memory model that a frozen base LLM queries at inference—an architecture close to our trained memory specialist deployed alongside a frozen task model—but its memory model encodes static document knowledge for question answering rather than the memory-management decisions of a long-horizon agent. Our work differs in coverage: we optimize both the memory *structure* (the scaffold) and the model's parametric *proficiency* at using it, whereas prior work typically addresses only one axis.

**Cognitive science of memory management.** Our work draws on *metamemory*—the capability to monitor and regulate one's own memory processes—introduced by Flavell [1979] and formalized by Nelson [1990] as a monitor–control loop between a meta-level and an object-level. The Extended-Mind thesis [Clark and Chalmers, 1998] argues that such aids—notes, diagrams, files—are part of the cognitive system, not mere accessories. These cognitive-science perspectives have recently been applied to LLMs: CoALA [Sumers et al., 2023] explicitly separates working- from long-term-memory in a language-model architecture; and MetaMem [Xin et al., 2026] introduces a meta-memory layer that self-improves its knowledge-retrieval strategies across tasks. AUTOMEM turns the *metamemory* concept into a **concrete optimization target** rather than an interpretive lens or a symbolic rule set: the scaffold loop shapes the memory structure, and the training loop sharpens the model's proficiency at the "monitor–control" decision-making process.

**Automated agent optimization.** A growing line of work automates the optimization of LLM agent systems. The Automated Design of Agentic Systems framework (ADAS) [Hu et al., 2024] searches over agent architectures in code; AFlow [Zhang et al., 2024a] casts agentic workflow generation as Monte Carlo tree search over code-represented workflows; DSPy [Khattab et al., 2023] compiles LM pipelines into optimized prompt chains; PromptBreeder [Fernando et al., 2023] and APE [Zhou et al., 2023] evolve or search over prompts. Closest to our setting, MemEvolve [Zhang et al., 2025a] evolves memory architectures from a modular design space of "encode, store, retrieve, and manage" components; and EvolveMem [Liu et al., 2026] casts memory optimization as an "AutoResearch" loop in which an LLM diagnoses per-question failure logs and proposes retrieval-configuration changes.

Our scaffold loop (Outer-loop #1) shares a similar "diagnose-and-revise" structure but rewrites the agent's code, prompts, and memory-file schema rather than tuning a fixed retrieval configuration, and draws its update signal from **complete long-horizon trajectory analysis** (up to $10^5$ steps) rather than from per-question logs in multi-session QA. These methods are also inference-only, whereas AUTOMEM additionally trains the model's memory proficiency (Outer-loop #2).

**LLMs in games.** Long-horizon game environments have become a standard testbed for LLM agents. Voyager [Wang et al., 2023a] uses LLMs to acquire skills and maintain a reusable skill library in Minecraft; DEPS [Wang et al., 2023b] applies structured "describe–explain–plan–select prompting" to open-world Minecraft tasks; NetPlay [Jeurissen et al., 2024] is the first zero-shot LLM agent applied to full NetHack [Küttler et al., 2020], underscoring the difficulty of dynamic context management at that scale. In the embodied-agent paradigm more broadly, ReAct [Yao et al., 2022] interleaves reasoning with acting; Reflexion [Shinn et al., 2023] draws on verbal self-reflection across episodes; Inner Monologue [Huang et al., 2022] grounds LM planning in environment feedback; and ExpeL [Zhao et al., 2024] extracts reusable insights from past trajectories to guide future decisions without parameter updates. We use game environments for their **stochastic** nature and **long horizons**, but study a different perspective: memory management as the primary lever, rather than reasoning, planning, or retrieval architectures.

## 5 Conclusion

This work demonstrates that memory management is an independently learnable, high-leverage skill for LLM agents. With AUTOMEM, we factored this skill into two axes—*structure* and *proficiency*—and automated their optimization through meta-LLM-driven outer loops. Targeting memory alone, without modifying the gameplay model's weights, improved performance $\sim 2\times$–$4\times$ across three long-horizon game environments, approaching the level of frontier proprietary systems. These findings also validate the *metamemory* perspective from cognitive science as a productive framework for LLM agent design: skilled memory use is learned through practice and feedback, not designed into fixed architectures. The file-system interface, the scaffold revisions, and the trained memory specialist are all instances of this principle—the model acquires memory expertise by having its memory decisions observed, evaluated, and improved. A further finding is that long-horizon task improvement can decompose into trajectory-level review and targeted revision—a workflow that meta-LLMs can execute autonomously where human review of full episode traces is intractable. Applying this decomposition to other agent capabilities beyond memory is a promising direction.

## 6 Limitations and Future Work

Our current study has several limitations, each suggesting a direction for further work. First, the memory we study is *episodic*: the file system starts fresh at the beginning of each episode, and a natural extension is a *persistent* memory that carries knowledge across episodes. Second, our experiments are on game environments, which are well-suited to studying memory—long horizons, procedural generation, and rich information-management demands—and the approach could also be applied to real-world, memory-intensive tasks. Third, since the three games differ in structure and objectives, we optimize a separate scaffold and memory specialist for each; whether a single scaffold or specialist can be shared across environments remains to be explored.

## 7 Broader Impacts

AUTOMEM shows that automated scaffold optimization and targeted training of memory management as a distinct capability effectively improves performance on long-horizon tasks. This also lowers the model-scale threshold at which long-horizon agents become practical, an accessibility gain for open-weight deployment. The same techniques could be adapted to tasks beyond games. The released artifacts are not directly applicable to high-stakes deployment without further safety review.

## Acknowledgments

We thank Kaiyue Wen, James Burgess, Laura Bravo-Sánchez, and Mark Endo for their insights and helpful discussions.

## References

Andy Clark and David Chalmers. The extended mind. *analysis*, 58(1):7–19, 1998.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

John H Flavell. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American psychologist*, 34(10):906, 1979.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=nZeVKeeFYf9`.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

Dominik Jeurissen, Diego Perez-Liebana, Jeremy Gow, Duygu Cakmak, and James Kwan. Playing nethack with llms: Potential & limitations as zero-shot agents. In *2024 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

Jack Lanchantin, Shubham Toshniwal, Jason Weston, Sainbayar Sukhbaatar, et al. Learning to reason and memorize with self-notes. *Advances in Neural Information Processing Systems*, 36: 11891–11911, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Jiaqi Liu, Xinyu Ye, Peng Xia, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Evolvemem: Self-evolving memory architecture via autoresearch for llm agents. *arXiv preprint arXiv:2605.13941*, 2026.

Ali Modarressi, Abdullatif Köksal, Ayyoob Imani, Mohsen Fayyaz, and Hinrich Schütze. Memllm: Finetuning llms to use an explicit read-write memory. *arXiv preprint arXiv:2404.11672*, 2024.

Thomas O Nelson. Metamemory: A theoretical framework and new findings. In *Psychology of learning and motivation*, volume 26, pages 125–173. Elsevier, 1990.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, et al. Balrog: Benchmarking agentic llm and vlm reasoning on games. *arXiv preprint arXiv:2411.13543*, 2024.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

Ryan Wei Heng Quek, Sanghyuk Lee, Alfred Wei Lun Leong, Arun Verma, Alok Prakash, Nancy F Chen, Bryan Kian Hsiang Low, Daniela Rus, and Armando Solar-Lezama. Memo: Memory as a model. *arXiv preprint arXiv:2605.15156*, 2026.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*, 2021.

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

Theodore Sumers, Shunyu Yao, Karthik R Narasimhan, and Thomas L Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023b.

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

Haidong Xin, Xinze Li, Zhenghao Liu, Yukun Yan, Shuo Wang, Cheng Yang, Yu Gu, Ge Yu, and Maosong Sun. Metamem: Evolving meta-memory for knowledge utilization through self-reflective symbolic optimization. *arXiv preprint arXiv:2602.11182*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z Pan, et al. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *arXiv preprint arXiv:2508.19828*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*, 2026.

Qianhao Yuan, Jie Lou, Zichao Li, Jiawei Chen, Yaojie Lu, Hongyu Lin, Le Sun, Debing Zhang, and Xianpei Han. Memsearcher: Training llms to reason, search and manage memory via end-to-end reinforcement learning. *arXiv preprint arXiv:2511.02805*, 2025.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025a.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2024a.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026b.

Yuxiang Zhang, Jiangming Shu, Ye Ma, Xueyuan Lin, Shangxi Wu, and Jitao Sang. Memory as action: Autonomous context curation for long-horizon agentic tasks. *arXiv preprint arXiv:2510.12635*, 2025b.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024b.

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

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=92gvk82DE-`.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. *arXiv preprint arXiv:2506.15841*, 2025.

# A Implementation Details

This appendix gives implementation configuration organized by components. The complete prompt templates for both outer loops, together with all code, are released in our codebase: `https://github.com/autoLearnMem/AutoMem`.

## A.1 Game environment configuration

**BALROG environments.** We use the BALROG harness as released [Paglieri et al., 2024], with minor configuration changes: *(i)* We set `autopickup=True` on MiniHack to align with the NetHack configuration. *(ii)* The default fallback action when the LLM outputs no parseable action string is set to `search` where the task's action space includes it (NetHack and a subset of MiniHack tasks) instead of the BALROG default `esc`, and kept as `Noop` for Crafter. All baseline runs with `Qwen2.5-32B-Instruct` use the same updated settings as our AUTOMEM method.

Crafter is configured with a $64\times64$ world area, a $9\times9$ agent view, dense reward, `unique_items=True`, and `max_episode_steps=2000`. MiniHack covers eight tasks: Boxoban-Hard, Boxoban-Medium, MazeWalk-$9\times9$, MazeWalk-$15\times15$, Corridor-R3, CorridorBattle-Dark, Quest-Easy, and Quest-Medium, with `max_episode_steps=100`, `penalty_step=`$-0.01$. NetHack uses `max_episode_steps=100,000` and `no_progress_timeout=150`. For all three environments, `skip_more=True` suppresses the `-More-` prompt.

**Evaluation seeds.** We evaluate AUTOMEM using a fixed list of 10 seeds, $[42, 43, \ldots, 51]$, paired with per-environment episode counts: 10 episodes for Crafter, 5 episodes per task for MiniHack ($5 \times 8 = 40$ episodes), and 5 episodes for NetHack. Episode index $i$ always uses seed $42 + i$, so per-episode results are directly comparable across scaffold & memory-trained versions.

### A.2 Outer-loops and training pipeline

**Outer-loop #1.** The scaffold optimization is driven by Claude Opus 4.6 with `-effort max`. A revision is accepted only if its average progression on the same fixed eval seeds strictly exceeds the previous iteration's. When a fresh revision fails the gate, up to 1 retry is run within the same session where the meta-LLM is given the failed evaluation log and asked to revise again. If that also fails, the meta-loop is restarted from a clean session.

**Outer-loop #2.** The pipeline runs in three stages: (a) training-data collection, (b) data-engine selection, and (c) LoRA finetuning followed by two-model inference deployment. Stages (a) and (b) operate on the final scaffold from Outer-loop #1 (V5 for Crafter, V4 for MiniHack, V2 for NetHack); the trained adapter is then deployed back into that same scaffold under the two-model inference architecture.

**Stage (a) training-data collection**: The base model plays 100 episodes for Crafter, 50 episodes per task for MiniHack ($50 \times 8 = 400$ episodes total), and 50 episodes for NetHack under the final scaffold. Seeds are drawn randomly and explicitly disjoint from the 10 evaluation seeds $[42, \ldots, 51]$, so no train/eval seed contamination is possible.

**Stage (b) training engine:** The training engine is driven by Claude Opus 4.7 with `-effort max`. Each iteration it (i) sets the dataset composition (*e.g.,* which episodes to draw from, which standards to apply, minimum dataset size); (ii) selects training examples from the trace pool; and (iii) picks a LoRA training configuration. The trained model is then evaluated and the meta-LLM uses the eval trajectories to refine its choices on the next pass. Advisory per-environment configurations are given at the beginning of the loop as a starting prior. The training engine is free to deviate from these based on its own analysis.

After the meta-LLM produces the selection, a deterministic postprocessing step is applied: format artifacts (code-block wrappers) are cleaned from assistant messages; examples containing only gameplay action commitment with no memory operations are filtered out, since they carry no memory-op training signal for the specialist; and for examples that contain both memory operations and an action commitment, the gameplay action part is trimmed, retaining only the memory-operation reasoning. The final training sets selected by the training engine contain 1597 examples for Crafter, 444 for MiniHack, and 800 for NetHack.

**Stage (c) LoRA training.** We use `cutoff_len=16384`, bf16 precision, AdamW, and a cosine learning-rate schedule with `warmup_ratio=0.05`. We run training on two GPUs using DeepSpeed ZeRO-3. The LoRA hyperparameters differ across environments, and we report the per-environment configuration used to produce the deployed adapter:

- **Crafter:** `lora_rank=256`, `lora_alpha=512`, `lora_dropout=0.0`, `effective_batch_size=32`, `learning_rate=5e-5`, `num_train_epochs=4`, `target_modules=attention-only`.
- **MiniHack:** `lora_rank=128`, `lora_alpha=256`, `lora_dropout=0.0`, `effective_batch_size=16`, `learning_rate=5e-5`, `num_train_epochs=3`.
- **NetHack:** `lora_rank=256`, `lora_alpha=512`, `lora_dropout=0.0`, `effective_batch_size=32`, `learning_rate=5e-5`, `num_train_epochs=1`.

## B Memory-scaffold evolution by iteration

This appendix lists the main changes Outer-loop #1 made to the agent scaffold at each iteration, for all three environments. Every change edits the agent's prompts, its memory-file layout, or the automatic hints the scaffold computes from the observation and shows the agent; the two-phase LOG/PLAN loop and the memory-operation interface are otherwise unchanged.

**Crafter (v0 → v5).**

- **v1:** Pre-load `game_knowledge.txt` with the Crafter crafting tree and survival/placement rules, and replace the empty `strategy.txt` with a `goals.txt` template that the agent keeps up to date in every planning prompt. Warn automatically when health, food, or drink runs low, or when an action keeps failing.
- **v2:** State the inventory change since the previous step in the LOG prompt (*e.g.,* "gained +1 wood"). Add a 22-item achievement checklist (`progress.txt`), and detect back-and-forth movement loops.
- **v3:** For crafting near a table: escalate the warning as a craft keeps failing; after a table or furnace is placed, list what can now be built; when a table is in view, list the specific items the current inventory has the materials to make.
- **v4:** From the inventory, estimate the next crafting step and indicate which resource to gather next. Warn when health drops or a hostile is nearby, and list untried one-step achievements.
- **v5:** Before a craft or place action is committed, verify the agent has the required materials and otherwise block the action, so it stops attempting impossible crafts. Also block a sleep action next to a monster, reject a `Noop`-action once it has repeated several times, and warn when the inventory has not grown for many steps.

**MiniHack (v0 → v4).**

- **v1:** Track the agent's recent actions and warn it when it paces back and forth or repeats the same move. Provide per-task rules (*e.g.,* a dedicated rule sheet for the Boxoban push puzzles).
- **v2:** Record which map cells the agent has already visited and report which neighbouring directions are still unexplored. When the game asks a follow-up question (*e.g.,* "In what direction?"), prepend a directive to reply with a single direction word. Also auto-load the agent's most recent action-log entries into its prompt.
- **v3:** When the goal staircase is visible on the map, add a planning-prompt directive toward it. When the agent revisits the same cell, add a directive toward an unexplored passable direction. When the game asks a direction question, replace the long planning prompt with a short direction-only one.
- **v4:** Fix the map-reading the hints above depend on: parse passable directions correctly, read tiles per task (*e.g.,* the # symbol is an impassable bar in Boxoban), switch to another direction when a directed move leaves the position unchanged (*e.g.,* hits a wall), route around lava, and flag nearby doors.

**NetHack (v0 → v2).**

- **v1:** Add the `<|UPSERT_MAP|>` operation, which keeps one entry per map coordinate so a new sighting of a tile overwrites the old entry instead of piling up duplicates. Automatically trim the action log, pre-fill `strategy.txt` with the main goal and safety rules, and expand the game rules (*e.g.,* avoiding unsafe corpses).
- **v2:** When the agent is on or next to a downward staircase, issue a high-priority directive to descend. Maintain two memory files from the observation every step, a `current_status.txt` and an up-to-date `inventory.txt`. Warn on back-and-forth movement.