PRO-LONG: PROGRAMMATIC MEMORY ENABLES LONG-HORIZON REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon tasks require sustained perception, reasoning, and exploration, and are a persistent challenge for large language model (LLM) agents. This gap is reflected in their limited performance on continual learning benchmarks such as ARC-AGI-3 and NetHack, especially when evaluated without specialized harnesses. Various agent harnesses have been proposed to close the gap, and each commits to a strategy for handling long sequences of observations, i.e., what information to save from the environment and how to load it into model context, a choice we argue is particularly consequential. Existing methods for context management have typically been designed around a tradeoff, where preserving more information makes relevant details harder to retrieve. Recent progress in coding agents, however, makes new approaches to memory viable. We propose **PRO-LONG**, a minimal context management framework built around *programmatic memory* for LLM agents in long-horizon, exploratory settings. PRO-LONG addresses the tradeoff by keeping a complete, structured interaction log and using programmatic tools to search this history efficiently. On NetHack, we find increasing gains from using PRO-LONG for cross-episodic memory, with scores reaching up to $\mathbf{2.2}\times$ those of a base coding agent. On the full ARC-AGI-3 public game set, PRO-LONG improves over this baseline agent by an average of $\mathbf{18.0}$ percentage points across frontier models, and matches or exceeds state-of-the-art specialized harnesses (up to $\mathbf{97.4}\%$ best@2) while using $\mathbf{4.2}$–$\mathbf{5.8}\times$ fewer tokens.

1 INTRODUCTION

Large language model (LLM) agents are being increasingly deployed on more challenging, longer-horizon tasks. Continual learning benchmarks such as ARC-AGI-3 (ARC Prize Foundation, 2026) and NetHack (Küttler et al., 2020) have subsequently been proposed to evaluate agentic progress, testing whether these agents can infer unknown environment dynamics through exploration and continuously improve over thousands of actions. Out-of-the-box frontier LLMs, evaluated without harnesses or tool access, perform poorly on these settings, achieving limited progression on both ARC-AGI-3 and NetHack (ARC Prize Foundation, 2026; Paglieri et al., 2025).

To close this gap, a range of agent harnesses has been proposed for exploratory tasks, wrapping a base model with memory, tools, and other scaffolding, which have shown considerable promise. In parallel, and of particular relevance, agents with access to programmatic tools, such as Claude Code and Codex, have advanced rapidly. Programmatic access enables forms of memory and reasoning that were previously impractical, such as structured search over very large logs. Recent work externalizes long-context processing in a similar spirit (Cao et al., 2026), for example through recursive language models that query slices of context through a REPL (Zhang et al., 2025a). These trends point toward a broader role for coding agents as context processing systems.

For these agents to maintain long-horizon competence and coherence, how to properly engineer context becomes a central concern. A natural way to frame agent context management is to distinguish between the *accessed* information, through the active context window of the Transformer network, and the *accessible* information, namely memory (i.e., retrievable states such as a log file, saved notes, or an embedding store). Although modern LLMs can process context windows of a million tokens or more, performance remains expensive and vulnerable to degradation as trajectories grow, a

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.

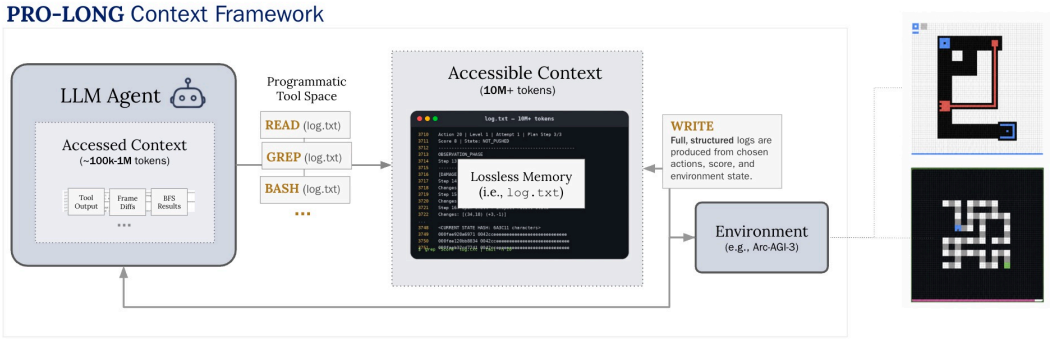


Figure 1: Overview of programmatic memory (PRO-LONG) integrated into a basic coding agent setup. The agent has access to a tool space of increasingly programmatic capabilities (`read`, `grep`, `bash`/`python`), and full, structured logs of environment state are written to `log.txt`. Note the accessed context (limited to ∼100K–1M tokens) versus the far larger accessible context (10M+ tokens). Two example ARC-AGI-3 games are shown on the right: g50t, which requires controlling a "ghost clone," and tu93, a maze game.

phenomenon often referred to as context rot (Hong et al., 2025). Agents therefore require mechanisms that decide what history remains accessible and how relevant pieces are moved into context.

Proposed agent memory systems can be understood as various implementations of READ and WRITE operations over externally stored information (Zhang et al., 2025b). WRITE is defined by heuristics for what information should be accessible or persisted, whether agent-authored, human-designed, learned, or task-specific. Examples include scratchpad notes, summarized observations, embedded memory entries, and distilled skills or strategies (Park et al., 2023; Packer et al., 2023; Chhikara et al., 2025; Ouyang et al., 2026; Xia et al., 2026). READ then determines what is ultimately accessed in context through analogous operations, such as embedding-based retrieval, loading selected memory chunks, keyword search over stored files, or restoring earlier context snapshots (Lewis et al., 2020; Wu et al., 2025). This creates a fidelity–tractability tradeoff. Reducing what is saved at WRITE time makes READ easier, but can discard details whose relevance is evident only in hindsight. Over long horizons, the cost of these decisions accumulates, as a lossy summary can matter many turns later. A timeliness question is also raised as to whether these systems are fully capitalizing on, or are maximally compatible with, modern coding agent capabilities.

We propose **PRO-LONG**, a context management framework for eliciting **pro**grammatic, **long**-horizon reasoning. PRO-LONG centers on our notion of programmatic memory for agents, which we define the WRITE operation as the harness appending every observation, action, and outcome into a log, and the READ as programmatic search over that log, primarily through code. We emphasize three motivating design principles. First, *simplicity*; the WRITE needs no learned or heuristic decisions about what information to store, since everything is added as the environment outputs it. READ requires no specialized retrieval, e.g., accessing a vector database or index. Second, *losslessness*; nothing is compressed or summarized when writing, so the log is a faithful, ground-truth record of environment state. Third, the approach is *uniquely enabled by, and especially compatible with, coding agents*. Programmatic search, via regular expressions or code, is native to coding agents (Cao et al., 2026), and this capability keeps retrieval tractable even over very long logs; in practice, we find PRO-LONG retrieval to be tractable for logs over 100K+ lines.

Contributions. PRO-LONG achieves state-of-the-art results on ARC-AGI-3 and NetHack; we attribute its performance to, and organize our contributions around, the three design principles above.

- **Framework.** We introduce PRO-LONG, a context framework optimized for long-horizon coherence, which introduces programmatic memory for agents as an append-all WRITE to a log and a code-based READ over that log. Our design is motivated by simplicity, losslessness of information, and maximal compatibility with coding agent capabilities.

- **Benchmark Performance.** Across model families and under matched scoring setups for ARC-AGI-3, PRO-LONG achieves performance comparably to or better than specialized harnesses while using 4.2–$5.8\times$ fewer billed tokens. With Opus 4.6, it scores 42.4% pass@1, the best public result for that model. With GPT-5.5, PRO-LONG reaches 41.2% pass@1, compared with the prior best of 45.1%, while using $5.8\times$ fewer tokens. With Fable 5 at a full action budget, we achieve $\mathbf{97.4}\%$ best@2 at less than one-third the cost of the strongest prior harness. On NetHack, PRO-LONG's best episode reaches $\mathbf{25.7}\%$ in a single run and $\mathbf{40.8}\%$ within a lifetime, above the best published episode (20.6%). Despite being a minimal addition to standard coding agents, PRO-LONG improves over the base coding agent without a log by 18.0 points on ARC-AGI-3 and by up to $2.2\times$ on NetHack.
- **Drivers of Performance.** We show that adding PRO-LONG to coding agents with even minimal tooling performs strongly, supporting the simplicity principle. The gains are driven by full log access. Performance improves steadily with increasingly powerful programmatic access (Table 2), while further abstractions, such as persistent workspaces and tools for writing notes, add little (Table 3). Improvements are clearest in games where long-horizon reasoning is most useful, i.e., the current board cannot determine the game dynamics (Figure 6).

2 Setup & Methods

2.1 Benchmarks and Evaluation

We evaluate on two challenging continual learning benchmarks, ARC-AGI-3 and NetHack, which stress different aspects of long-horizon reasoning. ARC-AGI-3 has 25 public games of six to ten levels each; the rules are not disclosed, so the agent must infer environment dynamics through exploration. NetHack is a classic dungeon crawler whose conventions an agent is likely to know, but a single game can take over $10{,}000$ actions and ends permanently on death. Therefore, it demands long-horizon planning and, across repeated episodes, the ability to learn from much earlier failures. We detail each environment and its evaluation protocol below.

ARC-AGI-3. To provide intuition for the games in this benchmark, Figure 1 shows example board states from two: `tu93`, a maze game in which the agent must reach a goal tile while disarming turrets, and `g50t`, where "rewinding" creates a ghost that repeats the agent's previous path, so runs must be planned such that ghosts hold switches open. For all games, boards are 64×64 grids of pixels, each taking one of 16 colors. The action space consists of four directional moves, a click targeting a specific (x, y) cell, undo, and reset. Available actions vary by game, from click-only to the full set, and most games have an action budget or timer that forces a reset when exceeded.

We use the benchmark's standard metric, *Relative Human Action Efficiency* (RHAE) (ARC Prize Foundation, 2026). Each level is scored by the agent's action count $a_{l,e}$ against an upper-median human baseline $h_{l,e}$, $S_{l,e} = \min\big((h_{l,e}/a_{l,e})^2,\, 1.15\big)$, so beating the human baseline earns up to 115%. Later levels are weighted more heavily, each environment's score is capped at its weighted fraction of levels completed, and the underlying weighted average over the $|D|$ environments is:

$$T = \frac{1}{|D|} \sum_{e \in D} \frac{\sum_{l=1}^{n} l\, S_{l,e}}{n(n+1)/2}. \tag{1}$$

NetHack. We also evaluate on the NetHack Learning Environment (NLE) (Küttler et al., 2020) using the BALROG evaluation harness (Paglieri et al., 2025). NetHack is a dungeon exploration game where winning requires descending more than 50 dungeon levels. A successful NetHack run thus often requires tens of thousands of actions. Each run procedurally generates a dungeon, whose map is revealed as the agent explores. The agent must manage food and equipment, as well as fight or avoid monsters; death permanently ends the episode. The BALROG progression score maps the deepest dungeon level and the highest experience level an agent reaches to the empirical probability that a human run at that point goes on to win. The resulting scores range from 0% to 100%. In our cross-episodic setting, following prior work on MiniHack (a simplified derivation of NetHack) (Nottingham et al., 2024; Ahmed et al., 2026) and described further in §3.2, each lifetime consists of five sequential episodes played by the same agent on a fixed seed (seed 0; Appendix **??** reports a second seed); PRO-LONG retains its log across episodes.

Table 1: **Summary of metrics used in our evaluations.** ARC-AGI-3 repeats are independent runs over each game (25 total), while NetHack repeats are sequential episodes, with up to 5 in a lifetime.

	Averaged	**Best of**
ARC-AGI-3	*pass@1:* mean score over independent runs	*best@k:* max over k independent runs
NetHack	*mean best@k:* best score of episodes $1..k$ for each lifetime, averaged over all lifetimes	*max best@k:* best of episodes $1..k$, any lifetime

Evaluation protocol. Unless otherwise specified, all evaluated agents run at high model reasoning effort. On ARC-AGI-3, each game allows at most 500 actions, with at most 20 per turn; for all agents that do not use vision, including ours, boards are represented as 64×64 text grids with one character per pixel and the 16 colors mapped to ASCII symbols. In NetHack, each episode allows at most 20,000 environment actions, submitted in batches of at most 30. The agent receives BALROG's text observation, including the map, status, inventory, and game messages. GPT models use the Codex agent framework and Claude models use Claude Code, each with its standard tool space (e.g., `read`, `write`, `bash`, `grep`) in a sandboxed container that provides `python3` with only the standard library. We also evaluate PRO-LONG with Kimi K3 on Mini SWE Agent (Lieret & Jimenez, 2025), a minimal harness (around 200 lines of code) with a single `bash` tool.

Evaluating on multiple continual learning benchmarks at once, with multiple frontier models, has little precedent. We see significant run-to-run variation for most models (§3.3), so Table 1 summarizes the averaged and best-of metrics we report on both benchmarks whenever compute permits. On NetHack, we evaluate lifetimes, which constitute five dependent runs as sequential episodes. On ARC-AGI-3, repeated runs are independent, with a fresh session and workspace for each game, so no information is transferred across runs or games; we report 95% bootstrap confidence intervals. Because prior agents were evaluated under different action budgets or scoring versions, we rescore their publicly available runs for a fair comparison.[1]

2.2 PRO-LONG & Agent Architectures

We build on coding agents, i.e., LLMs operated in a loop with access to a file system and a shell, able to read and write files and execute code as part of their reasoning. Following §1, we distinguish the agent's *accessed* state, the information in its active context window at a model call (e.g., system prompt, the current board, and recent tool outputs), from its *accessible* state, the external memory it can reach only through tools. Figure 1 visualizes how our context framework, PRO-LONG, integrates into a basic coding agent. Below, we define each agent architecture evaluated, including other public baselines. Full prompts for our agents are provided in Appendix C.

Base coding agent. We define a simple coding agent baseline to measure the effectiveness of adding PRO-LONG to a minimal setup. At each turn, the agent reasons over its accessed context of up to $\sim$250K tokens, may call the tools described in §2.1, and outputs a short sequence of actions. The environment executes these actions and then calls the agent again, with the resulting board added to the user prompt; we ablate this choice in Figure 7. Earlier boards may remain in the active context window, but the baseline provides no external memory beyond what the agent writes itself, e.g., notes or saved boards in workspace files; in practice, it relies heavily on such workspace writing (Table 4). We denote this baseline *no-log* and use it as our primary point of comparison. This baseline is stronger than the standard agents in continual learning benchmarks, which typically are only given the current state and a short history, without coding tools. For example, ARC-AGI-3 reference agents are given the last 3–5 board states through a 10-message window (ARC Prize Foundation, 2026), and BALROG agents receive the last 16 observations (Paglieri et al., 2025).

PRO-LONG. We add PRO-LONG, our minimal context framework, to the base coding agent above. Figure 1 shows the corresponding WRITE and READ operations for this harness. For each action, the WRITE appends a structured entry `logs.txt` containing a header (the action number, level, attempt, and score), the agent's summarized plan, the selected action, and the resulting board.

[1]For example, Arcgentica's publicly reported 36.1% was calculated under an earlier ARC scoring version and rises to 39.0% under the current scorer (v0.97) used for all agents here.

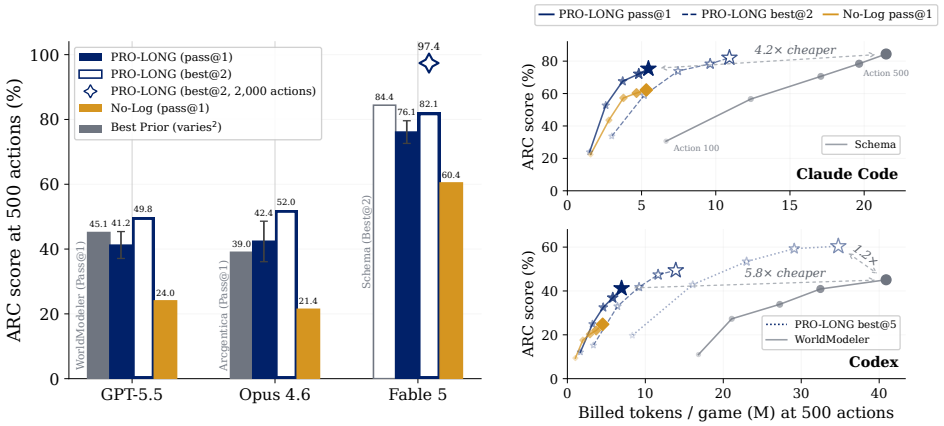


Figure 2: **PRO-LONG matches or exceeds state-of-the-art ARC-AGI-3 results at** 4.2–5.8× **lower token cost.** *Left:* ARC-AGI-3 results on the public game set with a 500 action limit, grouped by model and harness. Filled bars show pass@1, with bootstrap confidence intervals where multiple runs are available (five for Codex, two for Claude Code). Outlined bars show best@k. We rescore the released runs of WorldModeler, Arcgentica, and Schema for a consistent comparison (see footnote 3). Schema reports best@2; the others report pass@1. *Right:* Performance versus billed tokens per game for PRO-LONG and the strongest prior harness on Codex and Claude Code, from 100 to 500 actions. PRO-LONG stays within 2–4 points of the strongest prior harness at 4.2–5.8× lower cost.

The READ searches over this log with the programmatic tools, i.e., regular expressions or code, of effect we discuss in §3.3. With these tools, the agent can, for instance, `grep` the log for score changes to identify the most consequential actions for a level. An interesting example we see is a PRO-LONG agent that writes a `regress.py` script to replay every action in `log.txt` and verify its coded game model predicts the logged board states. We go over more examples on specific games in §3.3. We also study an intermediate condition for PRO-LONG that truncates the log to the last 25 actions (§3.1). We use no subagents, and our prompt is only around 30 lines (Appendix C.2).

Baselines. For ARC-AGI-3, in addition to our base coding agent, *no-log*, we compare against WorldModeler[2] (Rodionov, 2026) (GPT-5.5), Arcgentica (Symbolica AI, 2026) (Opus 4.6), and the Schema harness (Zeng et al., 2026) (Opus 4.8 and Fable 5). We focus on these three as the strongest public harnesses for the frontier models evaluated, that also follow the standard assumptions of ARC's official leaderboard, including independent runs with no information shared across games.

WorldModeler and Schema both focus on having the agent create coded "world models" of the games. WorldModeler, a harness for Codex, has a prompt of around 600 lines instructing the agent to write and continually update a Python simulator of the game. The harness provides helper scripts and two subagents for tasks like frame preprocessing and animation analysis. WorldModeler is also the only agent evaluated that uses vision. Schema follows a similar approach on Claude Code, replacing the default tools with twelve custom ones. Across turns, the Schema agent updates a `step(grid, action)` function predicting the next board, calls a backtest tool to validate the model, and searches for plans by breadth-first search. Arcgentica is built on a custom agent SDK, and defines a main orchestrator agent that delegates to four subagents (Explorer, Theorist, Tester, and Solver), which all update a shared memory database of hypotheses and findings.

On NetHack, our references are BALROG's official agent (the leaderboard configuration, given the last 16 observations and no tools) run with the same models as ours, and Glyphbox (Vaguely Aligned, 2026), a NetHack-specific harness exposing a high-level game API for pathfinding and auto-exploration, alongside context compression and explicit note-keeping.

[2]The released agent of Rodionov (2026) is named `baseline1`; we refer to it as WorldModeler for clarity, given the executable world model emphasis in their implementation.

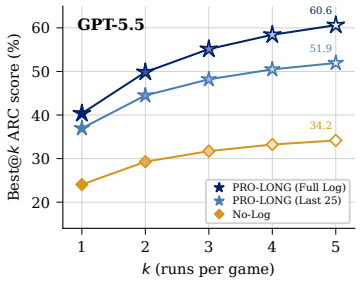


Figure 3: **Best@k comparison** for PRO-LONG, a truncated version (log keeps only the last 25 actions), and no-log. The gap widens with k.

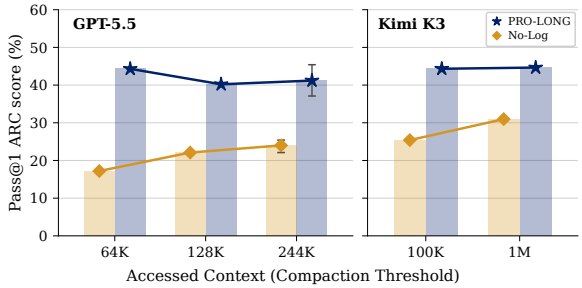


Figure 4: **Accessed versus Accessible Context.** Decreasing accessed context does not impair PRO-LONG, while no-log degrades. *Left:* GPT 5.5 + Codex. *Right* Kimi + Mini-SWE. ($n = 5$ for 244K, $n = 1$ otherwise).

3 Main Results

We evaluate PRO-LONG on the full ARC-AGI-3 public game set and on NetHack. Despite being a minimal addition to a basic coding agent, PRO-LONG improves significantly over the no-log baseline on both, and on ARC-AGI-3 matches or outperforms specialized harnesses at much lower token cost. We present benchmark performance (§3.1), study cross-episodic learning on NetHack (§3.2), and ablate which components drive performance and analyze agent behavior qualitatively (§3.3).

3.1 ARC-AGI-3 Benchmark Performance

Figure 2 compares benchmark performance for PRO-LONG and the relevant baselines discussed in §2.2. Across all frontier models: GPT 5.5, Opus 4.6, and Fable 5, PRO-LONG consistently improves by 15.7–21.0 percentage points over the coding agent baseline (no-log, pass@1). Further intuition for this result, and how PRO-LONG incentivizes long-horizon reasoning, are in §3.3.

Under matched scoring setups,[3] PRO-LONG either surpasses specialized harnesses (Opus 4.6, +3pp at pass@1) or is within comparable range (2–4pp on GPT-5.5 and Fable 5). We emphasize that PRO-LONG's simple design is model-agnostic in implementation, whereas prior harnesses are typically tuned to a specific model. This simplicity also translates into substantial token savings.

In Figure 2, we also compare the billed token cost of our method against the best prior harness on each coding agent, Claude Code and Codex. On Codex, PRO-LONG reaches 41.2 pass@1 versus WorldModeler's 45.1 while using $\mathbf{5.8}\times$ fewer tokens. At best@5, it reaches 60.1 while remaining $\mathbf{1.2}\times$ cheaper. On Claude Code, PRO-LONG reaches 82.1 best@2, close to Schema's 84.4 best@2 while using $\mathbf{4.2}\times$ fewer tokens. Across the full game set, this corresponds to savings of roughly **400 million tokens**.

Beyond the matched 500-action comparison, we also run Fable 5 at a 2,000-action budget, where PRO-LONG scores 94.6 pass@1 at a cost of $1,500 (150M billed tokens). PRO-LONG reaches $\mathbf{97.4}\%$ best@2 at a total cost of $1,750; this is a lower bound on best@2, as several games were not rerun. A few difficult games account for most of the cost; `bp35` alone costs $298. Schema releases logs only for its retained best runs (up to 3,000 actions),[4] whose reported cost totals $6,447 for 99.0 best@2; even this partial total is more than $\mathbf{3}\times$ higher than ours.

[3] For prior agents, the selection procedure is not always public and replicates are not available. WorldModeler evaluates a single playthrough per game, and Arcgentica releases one run per game with no stated selection procedure; we label both pass@1. Schema uses a best-of-two heuristic over Opus 4.8 and Fable 5, which we compare against our best@2.

[4] Schema releases logs only for each game's retained run, i.e., the better of its up to two attempts, so our cost comparisons for Claude use retained run costs. Even comparing the full cost of our best@2 (both runs) against Schema's single retained runs, PRO-LONG remains 2.0× cheaper. For Codex, we instead report the full best@5 cost (all five runs) to emphasize the savings relative to WorldModeler's pass@1.

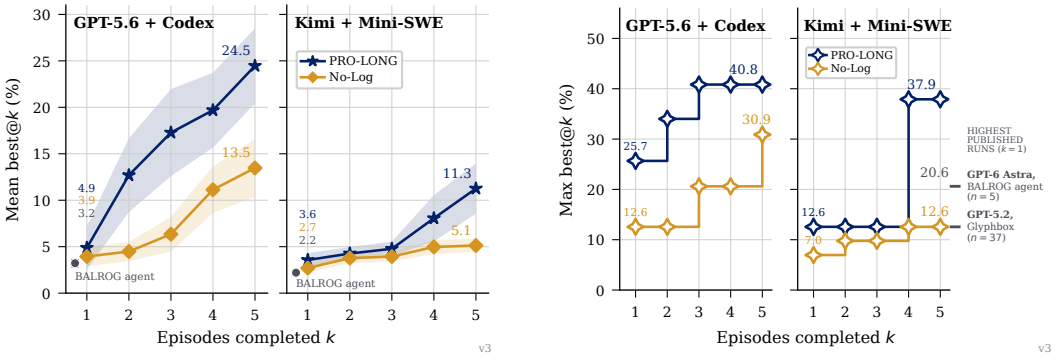


Figure 5: **PRO-LONG shows increasing gains on NetHack with cross-episodic memory.** BALROG progression over the first k episodes of a lifetime for GPT-5.6 + Codex (n=10) and Kimi + Mini-SWE (n=15 each). *Left:* Mean of the best score within the first k episodes, $\pm$ SE across lifetimes, with the baseline BALROG agent shown in gray at k=1. *Right:* Best episode observed within the first k episodes of any run, with the highest published single episode runs marked on the right.

In Figure 3, we do further best@k analysis, looking at how score improves as k increases and how variants of PRO-LONG compare. Note, first, there is significant variance in game outcomes across independent runs, and thus it is important to report best@k results where possible. For example, GPT-5.5 reaches 41.2 at pass@1, while best@5 rises by **18.9** percentage points to 60.1. Second, increasing the "fullness" of memory leads to a large improvement in score, and this gap widens with k. The widening gap suggests that fuller memory covers more games, i.e., expands the set of games solvable at all, and that games requiring long-horizon reasoning are harder in general, with higher variance that may reflect interesting exploration failure modes.

3.2 Cross-Episodic Learning on NetHack

We complement our ARC-AGI-3 results by evaluating on NetHack, as scored by BALROG progression (Paglieri et al., 2025). NetHack episodes can be far longer—runs exceed 10,000 environment actions in our tests—and are therefore a stress-test of long-horizon competency. A single episode has significant variance due to the permanent death mechanic, which makes NetHack a suitable setting for testing explicit *cross-episodic memory*. This setting retains information from one episode into later ones, and in particular can test whether a weak early episode is recoverable (which we find it can; Figure 10). Similar cross-episode tests have been previously studied on simplified MiniHack (NetHack-derived) tasks (Nottingham et al., 2024; Ahmed et al., 2026).

To test cross-episodic memory, we define *lifetimes* which consists of five sequential episodes played by the same agent. PRO-LONG persists its log across episodes, such that the accessible history spans a full lifetime, which is empirically over 24,000 cumulative actions. At $k = 1$, before any cross episode history is available, all agents are weak (including the base BALROG agent), with PRO-LONG only showing a modest improvement over no-log. After five episodes, the average best@k progression reaches $\mathbf{24.5}\%$ versus 13.5% with GPT 5.6 and $\mathbf{11.3}\%$ versus 5.1% with Kimi, approximately twofold higher in both settings (Figure 5). We find that PRO-LONG with GPT 5.6 improves increasingly and steadily; with Kimi, we see that performance is uniformly weak on early episodes across all harnesses, while PRO-LONG jumps later in score (max best@k $12.6\% \rightarrow 37.9\%$) in episode four and five.

Released BALROG leaderboard episodes reach at most 20.61% (GPT-6 Astra, best of n=5 independent episodes), and Glyphbox's best GPT-5.2 episode reaches 12.56% (n=37) (Vaguely Aligned, 2026). Even without cross-episode memory, PRO-LONG's strongest first episode (GPT-5.6 + Codex) reaches $\mathbf{25.66}\%$, above the prior best of 20.61%. With it, the best Codex lifetime reaches $\mathbf{40.8}\%$, and even a Kimi lifetime reaches 37.9%, above every public episode.

Table 2: Tool ladder ablation along the hierarchy of programmatic access (GPT 5.5). Increasing level of programmatic control over logs and programmatic reasoning lead to improved performance, whereas other types of tooling such as write do not as much.

Tools enabled	Score (%)
Read only	23.1
+ Grep / Regex Search	27.2
+ Python	38.3
+ Write, Edit	41.2

Table 3: Workspace persistence ablation. pass@1 (%) for GPT 5.5 with bootstrap confidence intervals ($n = 5$ persistent, $n = 2$ cleared). Clearing the workspace after every call has little effect on PRO-LONG but reduces no-log performance by about four points.

Workspace	PRO-LONG	No-Log
Persistent	41.2 ±3.5	24.0 ±2.0
Cleared every call	40.7 ±3.6	19.9 ±2.1

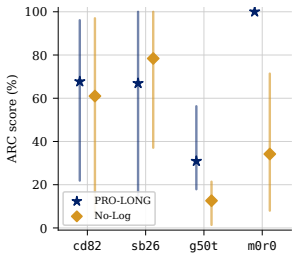
3.3 General Ablations

PRO-LONG works with limited accessed context. We observe that `logs.txt`, PRO-LONG's primary source of accessible context, can grow up to 320K lines long, while any given turn accesses only a small fraction. The accessed context is itself controllable, with direct implications at inference time (Table 6). We ablate it by lowering Codex's compaction threshold from 244K to 64K. PRO-LONG does not degrade, scoring slightly higher at 64K than at the default (44.3% vs. 41.2%, within its confidence interval), while no-log falls monotonically. We find the same trend in a simpler setting, using Kimi and the Mini-SWE harness; reducing the threshold from 1M to 100K, PRO-LONG stays flat, within 0.3 points, while no-log drops more than five.

Programmatic access drives performance. PRO-LONG's READ operation is defined by a "programmatic tool space." We would expect (1) more capable tools to make the log more useful, and (2) relevant tools that capitalize on the log, rather than other forms of memory, to show the largest performance gains. Table 2 tests, and supports, both these hypotheses. Starting from the full standard tool space (41.2%), removing `write` and `edit` tools lowers performance by only 2.9 points, even though these are the agent's primary means of maintaining other forms of memory, like notes. Programmatic tools account for most of the remaining gain. Starting from file reading alone (23.1%), adding regular expression search such as `grep` improves performance by 4.1 points, while adding Python access contributes more than ten additional percentage points.

Note that since PRO-LONG builds on a base coding agent, it can WRITE other forms of memory beyond the log, such as self-authored notes. A core property of how useful such memory is relies on whether it is persistent, i.e., notes must be maintained across model calls so the agent can refer back to them. We ablate how much PRO-LONG is affected by disabling the persistence of these other forms of accessible context, through wiping the workspace every model call. We find that for the no-log agent, which relies heavily on self-authored notes and helper functions (Table 4), performance degrades by roughly four points, while PRO-LONG is essentially unaffected (Table 3).

Table 4: Where programmatic memory helps (GPT 5.5). *Left:* Pass@1 on the four games discussed in §3.3, bars span score range over $n = 5$ runs. *Right:* Tool call distribution across the 25 games. PRO-LONG spends most of its calls on programmatic analysis of the log (40.8% → 60.6%), while No-Log spends them on workspace management (19.1% → 59.2%).



Category	Tools	PRO-LONG	No-Log
Programmatic analysis	`python3`	60.6%	40.8%
Log parsing	on `logs.txt`	20.3%	0.0%
Recent state	`tail`, `head`, `cat`	10.7%	–
Search across turns	`grep`, `sed`, `awk`	9.6%	–
Workspace management		19.1%	59.2%
Navigation	`find`, `ls`, `pwd`	14.6%	29.0%
Writing	`apply_patch`	4.6%	30.3%

Qualitative Analysis. We show individual game score statistics over five runs for PRO-LONG and our base coding agent in Figure 6, which helps explain where programmatic memory provides the greatest benefit. Both agents perform strongly on some games, such as `sb26` and `cd82`, where the current board largely specifies the puzzle dynamics, so access to the full trajectory provides little additional information; most agents we evaluate complete both in under 250 actions.

Programmatic memory is most useful for games that require longer-horizon reasoning, like `m0r0` and `g50t`. For `m0r0`, the agent controls two blocks in a maze-like game. Although our setup does not emphasize "world models," PRO-LONG codes one on its own, defining transition functions that grow with later levels (e.g., by adding a switch state variable) and running breadth-first search over joint block positions. PRO-LONG scores 100% on all five runs, compared with 34.2% for the no-log baseline. On `g50t`, where individual run logs exceeded 320,000 lines, it still reaches 56.3%. Tool use across the full set of games (Table 4) supports a similar attribution, as PRO-LONG spends most of its calls on programmatic analysis of the log, while no-log spends them on writing files.

4 Related Work

Context Management. LLMs have finite context windows, and their performance often declines as context gets longer due to context rot (Liu et al., 2024; Hong et al., 2025). Recent work on context engineering addresses this problem by storing information outside the model's active context and retrieving it when needed (Mei et al., 2025). Externalized memory systems can be defined by how they write, read, and synthesize information. Nye et al. (2021) let the agent write summaries into a scratchpad. MemGPT (Packer et al., 2023) uses a tiered memory system that moves information in and out of a fixed context window. Other systems, including Mem0 (Chhikara et al., 2025) and ReasoningBank (Ouyang et al., 2026), turn past experience into entries that can later be retrieved. Skill-library methods such as SkillRL (Xia et al., 2026) convert trajectories into reusable skills.

Programmatic Harnesses. An alternative is to keep information intact and decide what is relevant only at query time. Recursive language models (Zhang et al., 2025a), for example, use code to query parts of a long input through a REPL, and coding agents have been used for broader long-context processing (Cao et al., 2026). PRO-LONG applies the same idea to the agent's own trajectory: it keeps a complete, append-only log that the agent queries with code as needed. RLM-inspired harnesses represent game states programmatically to form, test, and revise hypotheses before selecting actions (Symbolica AI, 2026; Karten et al., 2026; Furgale et al., 2026), while world-model methods build an executable Python model of each game and plan with it before acting (Rodionov, 2026; Zeng et al., 2026; Courtis et al., 2026; Sarafian et al., 2026).

5 Conclusion

We introduce **pro**grammatic memory for **long**-horizon reasoning with our framework PRO-LONG. We claim that (1) programmatic perception and reasoning, such as using code to parse board grids, model game transitions, and plan with breadth-first search, are broadly effective for continual learning tasks, and (2) simple harnesses can elicit these behaviors, with PRO-LONG a minimal but highly effective instance. We emphasize simplicity, alongside lossless memory and compatibility with coding agents. Even with a minimal prompt and agent architecture, PRO-LONG frequently creates sophisticated models of game dynamics and long-horizon action plans. As such, our model-agnostic method is competitive across all evaluation settings: on ARC-AGI-3, it achieves up to 97.4% on the public benchmark while using significantly fewer tokens than the strongest prior harness on each model; on NetHack, it reaches 25.7% in a single episode, above the best prior published run.

We evaluate across various frontier models, two of which were released after ARC-AGI-3. Although this may advantage agents using those models, including the baselines, ARC-AGI-3 remains useful for relative comparison of long-horizon reasoning and context management methods. We also note the significant variance we see across game runs for both benchmarks, and thus report best@k where possible. Given the high cost of evaluating on continual learning benchmarks, an important direction is to see if reinforcement learning and improved harnesses can close the gap between pass@1 and best@k. Other exciting directions include co-training models with the harness, and studying how programmatic memory interacts with other memory mechanisms.

AI USE STATEMENT

We used language models to assist in draft edits and code design, e.g., debugging, run log analysis, and refining this manuscript. Any edits were done so under author supervision, and any LLM-generated code was verified and tested by the authors. The authors take full responsibility for methods, results, and prose.

REPRODUCIBILITY STATEMENT

We will publicize the code to reproduce all of the experiments once the paper is released. One can refer to Section 2.2 for the coding agent setup. For benchmark and evaluation setup, we have detailed them in Section 2.1. Appendix C contains the full prompts we used for the agent.

REFERENCES

Zergham Ahmed, Kazuki Irie, Joshua B. Tenenbaum, Christopher J. Bates, and Samuel J. Gershman. Learning abstractions for hierarchical planning in program-synthesis agents. *arXiv preprint arXiv:2602.00929*, 2026. URL https://arxiv.org/abs/2602.00929.

ARC Prize Foundation. ARC-AGI-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026. URL https://arxiv.org/abs/2603.24621.

Weili Cao, Xunjian Yin, Bhuwan Dhingra, and Shuyan Zhou. Coding agents are effective long-context processors, 2026. URL https://arxiv.org/abs/2603.20432.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. In *European Conference on Artificial Intelligence (ECAI)*, 2025. URL https://doi.org/10.3233/FAIA251160.

David Courtis, Wenhao Li, and Scott Sanner. Opine-world: Programmatic world modeling with ontology-error-prioritized interactive exploration for arc-agi-3, 2026. URL https://arxiv.org/abs/2607.01531.

Paul Furgale, Severin Klingler, James Nolan, Matt Staats, Gaia Di Lorenzo, Elisa Martinez Abad, Christian Schüller, Razvan Dinu, Alessio Devoto, Pascal Berard, Gal Kaplun, Elad Sarafian, Riccardo Roveri, Leon Derczynski, and Ricardo Silveira Cabral. NVIDIA-labs OO Agents: Native python object-oriented agents, 2026. URL https://arxiv.org/abs/2607.20709.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. https://www.trychroma.com/research/context-rot, 2025.

Seth Karten, Alex L. Zhang, Kevin Thomas, Sebastian Müller, Elie Bakouch, Daniel Auras, Mika Senghaas, Fares Obeid, Konstantin Dunas, Johannes Hagemann, and Sami Jaghouar. Prime agent: A self-improving rlm harness. *arXiv preprint arXiv:2608.23552*, 2026.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack learning environment. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL https://papers.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Kilian Lieret and Carlos E. Jimenez. mini-SWE-agent. https://github.com/SWE-agent/mini-swe-agent, 2025. Version 2.4.6.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models, 2025. URL https://arxiv.org/abs/2507.13334.

Kolby Nottingham, Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Sameer Singh, Peter Clark, and Roy Fox. Skill set optimization: Reinforcing language model behavior via transferable skills. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 38409–38425, 2024. URL https://proceedings.mlr.press/v235/nottingham24a.html.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. URL https://arxiv.org/abs/2112.00114.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking agentic LLM and VLM reasoning on games. In *International Conference on Learning Representations (ICLR)*, 2025.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *User Interface Software and Technology (UIST)*, 2023. URL https://doi.org/10.1145/3586183.3606763.

Sergey Rodionov. Executable world models for ARC-AGI-3 in the era of coding agents, 2026. URL https://arxiv.org/abs/2605.05138.

Elad Sarafian, Gal Kaplun, Ron Banner, Daniel Soudry, and Boris Ginsburg. Workspace optimization: How to train your agent, 2026. URL https://arxiv.org/abs/2605.09650.

Symbolica AI. From 0% to 36% on day 1 of ARC-AGI-3. https://www.symbolica.ai/blog/arc-agi-3, 2026.

Vaguely Aligned. It's 2026. Can LLMs Play NetHack Yet? https://kenforthewin.github.io/blog/posts/nethack-agent/, 2026.

Yaxiong Wu, Sheng Liang, Chen Zhang, Yichao Wang, Yongyue Zhang, Huifeng Guo, Ruiming Tang, and Yong Liu. From human memory to AI memory: A survey on memory mechanisms in the era of LLMs, 2025. URL https://arxiv.org/abs/2504.15965.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08234.

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/.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025a. URL https://arxiv.org/abs/2512.24601.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025b. URL https://doi.org/10.1145/3748302.

A Scoring and Token Cost Details

Games are run and scored with the official `arc_agi` engine, with scores computed according to Equation 1 and game versions pinned to those listed in Table 5.

Table 5: Game versions used for all runs in our experiments.

ar25-0c556536	bp35-0a0ad940	cd82-fb555c5d	cn04-2fe56bfb	dc22-fdcac232
ft09-0d8bbf25	g50t-5849a774	ka59-38d34dbb	lf52-271a04aa	lp85-305b61c3
ls20-9607627b	m0r0-492f87ba	r11l-495a7899	re86-8af5384d	s5i5-18d95033
sb26-7fbdac44	sc25-635fd71a	sk48-d8078629	sp80-589a99af	su15-1944f8ab
tn36-ef4dde99	tr87-cd924810	tu93-0768757b	vc33-5430563c	wa30-ee6fef47

We report cost in billed tokens. Each token class is weighted by its list price relative to one uncached input token of a reference model. Multiplying a run's billed token count by the reference input price therefore recovers its dollar cost. Token counts are taken from the usage reported by each API call and summed over the run.

Claude models (reference: Fable 5 input, \$10/M):

$$\text{billed} = \text{input} + 5\,\text{output} + 0.1\,\text{cache_read} + 2\,\text{cache_write}, \tag{2}$$

where `input_tokens` already excludes cached tokens. Cache writes use the one-hour rate because Claude Code issues only one-hour-TTL writes; we verified that `ephemeral_5m` $= 0$ across our runs. Opus has the same relative prices, so the same weights apply.

Codex models (reference: GPT-5.5 input, \$5/M):

$$\text{billed} = (\text{input} - \text{cached}) + 6\,\text{output} + 0.1\,\text{cached}, \tag{3}$$

No cache-write term is included because GPT-5.5 does not charge separately for cache writes.

All token comparisons in the paper use the same model family, so relative billed token counts do not depend on the chosen reference model.

B Additional Ablations and Analysis

Decreasing accessed context has varying effects on token cost. To complement our compaction analysis in Figure 4, we report what ablating the accessed context costs in terms of billed tokens. Table 6 reports token usage at each compaction threshold, with $Billed$ computed by Equation 3. Lowering this threshold nearly halves total input, from 46.4M to 24.3M tokens per game, but leaves billed cost roughly unchanged, as the savings are offset by a lower cache hit rate. Cached input decreases from 97.3% of $Input$ at the default to 90.7% at 64K.

Specific game performance for ARC-AGI-3. Figure 6 provides the full game-by-game comparison on the ARC-AGI-3 public set. The shortest consistently completed games were `sb26` and `cd82`. Across five runs per condition, successful runs required median totals of 157 and 213 actions for PRO-LONG, and 174 and 271 actions for no-log, respectively. Consequently, these are games in which PRO-LONG and no-log performed similarly. As discussed in Section 3.3, programmatic memory primarily helps on games requiring long-horizon reasoning. Exemplifying this, individual PRO-LONG logs for the long-horizon game `g50t` (Figure 1) exceeded 320,000 lines; PRO-LONG still reached a score of 56.3%. The primary game mechanic is "rewinding," which creates ghosts that replay the agent's previous paths, with later levels require coordinating multiple ghosts as they move and teleport toward switches.

Ablating no-log format. Our no-log baseline places each board in the user prompt, from which the agent can save information to workspace files, as described in §2.2. We ablate this choice by repeating the accessed-context experiment in Figure 4 with a variant that instead retrieves the current board through a read-only `observe` MCP tool. Thus, the only intended difference is how the current observation is delivered. The two variants follow the same overall trend. MCP scores 3.0 points higher at the default threshold, but degrades more as the compaction threshold decreases, falling below in-prompt at both 128K and 64K.

Table 6: **Token cost across compaction thresholds** (GPT 5.5, PRO-LONG), in millions of tokens per game. Default setting is averaged over $n = 5$. The billed token total adds the uncached input $Input - Cached$ to the weighted output $6Ouput$ and cached input $0.1Cached$.

Context setting	$Input$	$Cached$	$Input - Cached$	$Output$	$Billed$
64K	24.3	22.0	2.3	0.4	**6.9**
128K	30.5	29.1	1.3	0.3	**6.0**
Default (244K)	46.4	45.1	1.3	0.2	**7.1**

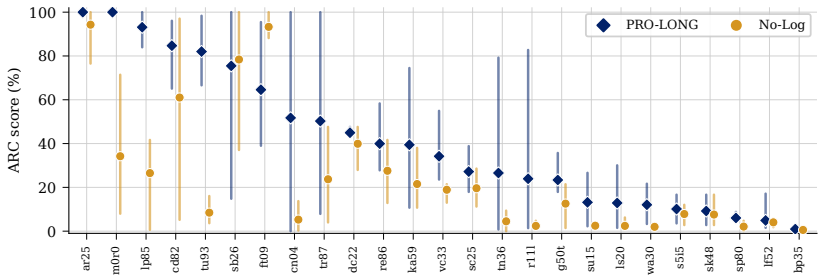


Figure 6: **Score breakdown across the 25 games, PRO-LONG vs. no-log** for GPT 5.5 runs, sorted by PRO-LONG mean. Markers show the pass@1 average per game and bars span the minimum and maximum score over $n = 5$ runs. PRO-LONG matches or exceeds no-log scores on nearly every game.

Individual NetHack lifetimes. Figures 9 and 10 show every GPT-5.6 + Codex and Kimi + Mini-SWE lifetime behind the means in Figure 5. Several PRO-LONG lifetimes score below 5% in their first one or two episodes and then reach 20–40%, whereas No-Log lifetimes mostly stay near the floor.

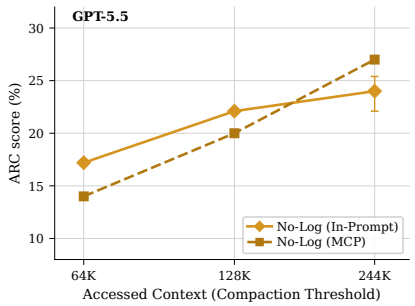


Figure 7: **No-log baseline ablation.** Using the same setup as Figure 4, we compare the default no-log condition, where the board is placed in the user prompt, with a variant that retrieves it through a read-only `observe` MCP tool.

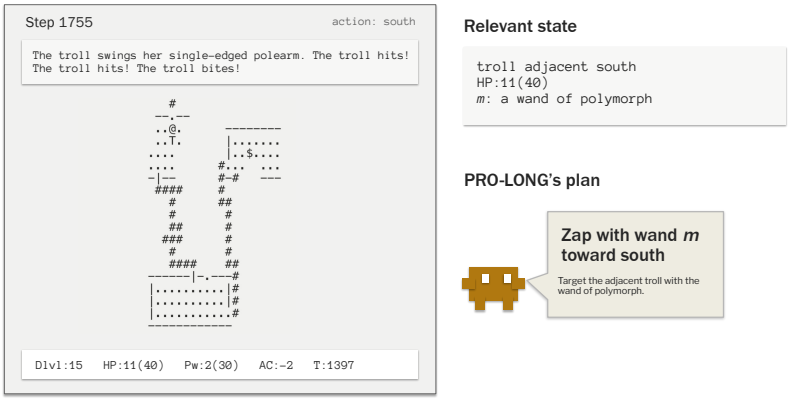


Figure 8: **An example NetHack state from a PRO-LONG episode.** An adjacent troll attacks the agent, leaving it at 11/40 HP (left). The PRO-LONG agent responds by zapping its wand of polymorph south at the troll (right).

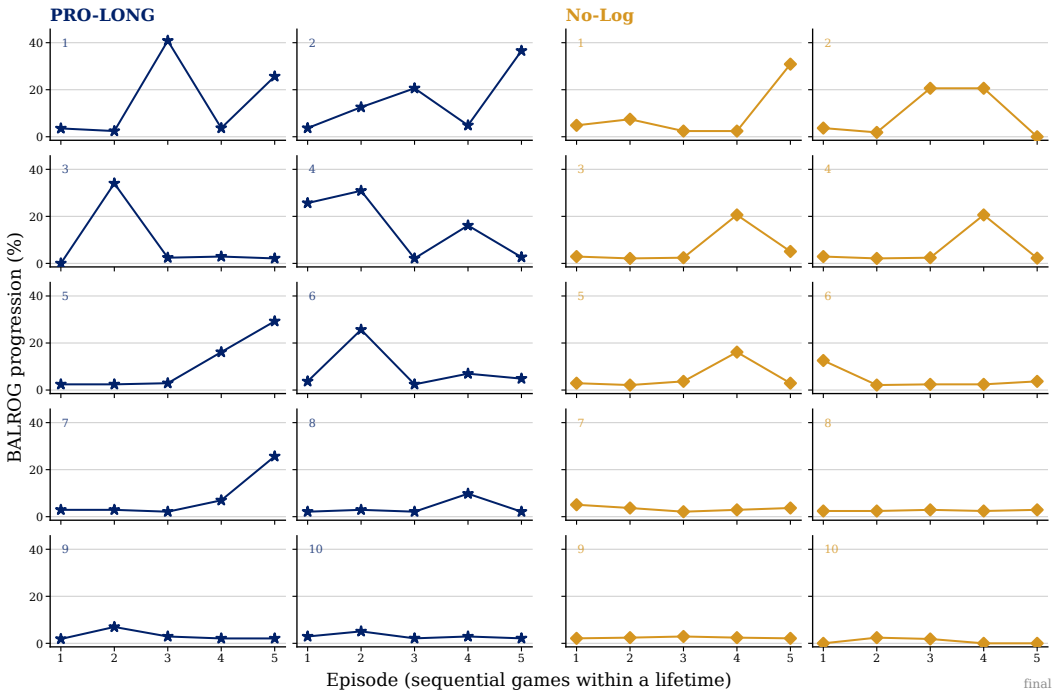


Figure 9: **Every GPT-5.6 + Codex lifetime.** BALROG progression for each of the five sequential episodes in all 10 PRO-LONG (left) and 10 No-Log (right) lifetimes, ordered by best episode.

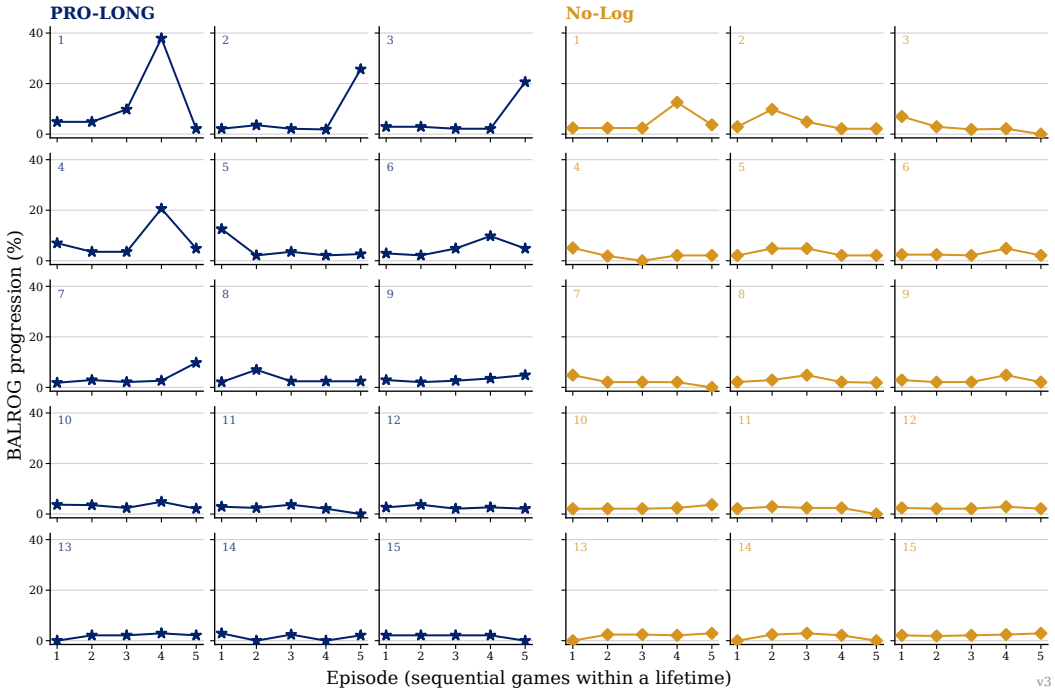


Figure 10: **Every Kimi + Mini-SWE lifetime.** BALROG progression for each of the five sequential episodes in all 15 PRO-LONG (left) and 15 No-Log (right) lifetimes, ordered by best episode.

C FULL PROMPTS

Values in braces are filled at runtime. Codex takes agent instructions from `AGENTS.md`, while Claude Code loads them from `CLAUDE.md`. Mini-SWE receives its adapted instructions as a system prompt. We refer to these collectively as agent instructions below. Each backend also receives the user prompts shown for its condition.

C.1 SHARED COMPONENTS

C.1.1 ACTION SPACE

Available Actions

```
- ACTION1 --- Up - ACTION2 --- Down - ACTION3 --- Left - ACTION4 ---
Right - ACTION5 --- Spacebar / interact - ACTION6(x,y) --- Click at
column x (0-63), row y (0-63) - ACTION7 --- Undo - RESET --- Reset
level (actions still count)
```

C.1.2 COLOR MAP

Color Map (ASCII Mode)

```
**Color Map (ASCII character -> color):** COLOR_MAP = { '$':
'White', '8': 'Off-White', '#': 'Light Gray', 'h': 'Gray', 'q':
'Off-Black', 'O': 'Black', 'C': 'Magenta', 'z': 'Light Magenta',
'n': 'Red', 'f': 'Blue', '(': 'Light Blue', 'G': 'Yellow', '-':
'Orange', '>': 'Maroon', 'I': 'Green', '"': 'Purple', }
```

C.2 PRO-LONG

C.2.1 AGENT INSTRUCTIONS

Agent Instructions (PRO-LONG)

```
You are a coding agent playing a grid-based puzzle game by writing
Python action plans.

Your primary objective is to solve all levels in the game. Your
secondary objective is to minimize total cumulative actions used.

`./logs.txt` is the game log: action headers, tool calls, board
states, and your own prior analyses. log_window_desc Parse it
**programmatically**, as reading full 64x64 board states from prompt
can introduce precision errors.cross_turn_hint

**Tools**: Read, Write, Edit, Bash, Grep, Glob.

**Workspace**: `./` persists across calls. `actions.json` is cleared
each call; other files accumulate. Feel free to save notes, state, or
helper functions.

**Log markers**: [INITIAL BOARD STATE] --- the grid at the start
(after Action 0 header) [POST-ACTION BOARD STATE] --- the grid after
each action (when the log includes action history) [frame 1/N]
... [settled] --- animation frames if the log includes them; the
grid following [settled] (or the only grid, if there are no [frame]
markers) is the committed state

**Game structure and strategy**: - Score increase means that a level
was cleared. - Most games have a step budget or timer mechanism,
which will cause a level reset if exceeded. - For parsing the boards,
programmatic options include identifying connected components (color,
position, size, shape) and forming testable hypotheses about what each
represents (player, walls, goals, UI, etc.).
```

```
**Response format**:  a strategic briefing, then [PLAN] <2-3 sentence
action plan>
**Write `./actions.json`** with a JSON object {"actions":
["ACTION6(30,40)", "ACTION1", "RESET"]} --- a list of 1--action_cap
actions to execute in order.  Entries beyond action_cap are discarded.
Prefer short lists (1--2 actions) when testing a new hypothesis so you
see the result before committing further; scale up toward action_cap
for proven sequences.
**Actions available in this game**:  actions_section
The runner executes the list in order, then calls you again with the
updated log.
```

C.2.2 User Prompts

Initial User Prompt

```
Read the full game log at ./logs.txt
This is the first analysis.  Analyze the board state and write
./actions.json with your first set of actions.
```

Resume User Prompt

```
Read the full game log at ./logs.txt
Recent actions and boards are at the end of the log; what changed
since the last call (new moves, score transitions, plan adherence)
can be informative.  Check ./ for anything you saved previously, then
write a new ./actions.json.
```

C.2.3 Mini-SWE Setup

For the Kimi experiments, Mini-SWE receives the PRO-LONG instructions above as its system prompt. Mini-SWE exposes the Bash tool, replaces the tool line below, omits the response format block, and appends the command instructions below. The initial and resume user prompts are unchanged.

Mini-SWE Prompt Addendum

```
**Tool**:  Bash.  Use shell commands to read, write, edit, and search
files.
## Command execution rules
Use the Bash tool to inspect and edit the workspace.  Each command
starts in a fresh shell.
When the action batch is ready, write the final analysis and optional
[PLAN] to ./last_message.txt.  Then submit with this command alone:
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

C.3 No-Log

C.3.1 Agent Instructions

Agent Instructions (No-Log)

```
You are a coding agent playing a grid-based puzzle game by writing
Python action plans.
Your primary objective is to solve all levels in the game.  Your
secondary objective is to minimize total cumulative actions used.
```

The current 64x64 board is included in the user prompt each turn. The workspace does not contain a game log, but files you create there persist across calls.

Tools: Read, Write, Edit, Bash, Grep, Glob.

Workspace: `./` persists across calls. `actions.json` is cleared each call; other files accumulate. Feel free to save notes, state, or helper functions.

Prompt markers (on the user turn): [CURRENT BOARD STATE] --- the live 64x64 grid

Game structure and strategy: - Score increase means that a level was cleared. - Most games have a step budget or timer mechanism, which will cause a level reset if exceeded. - For parsing the boards, programmatic options include identifying connected components (color, position, size, shape) and forming testable hypotheses about what each represents (player, walls, goals, UI, etc.).

Response format: a strategic briefing, then [PLAN] <2-3 sentence action plan>

Write `./actions.json` with a JSON object {"actions": ["ACTION6(30,40)", "ACTION1", "RESET"]} --- a list of 1--action_cap actions to execute in order. Entries beyond action_cap are discarded. Prefer short lists (1--2 actions) when testing a new hypothesis so you see the result before committing further; scale up toward action_cap for proven sequences.

Actions available in this game: actions_section

The runner executes the list in order, then calls you again with an updated board.

C.3.2 User Prompts

Initial User Prompt

[CURRENT BOARD STATE] board

This is the first analysis. Analyze the board above and write ./actions.json with your first set of actions.

Resume User Prompt

[CURRENT BOARD STATE] Score: score | Action: action_num | Level: level Last actions: last_actions

board

What ran since the last call is listed above; check ./ for anything you saved previously, then write a new ./actions.json.

C.4 MCP

C.4.1 Agent Instructions

Agent Instructions (MCP)

You are a coding agent playing a grid-based puzzle game by writing Python action plans.

Your primary objective is to solve all levels in the game. Your secondary objective is to minimize total cumulative actions used.

The game observation is available through the read-only `observe` MCP tool. The workspace does not contain a game log, but files you create there persist across calls.

Tools: Read, Write, Edit, Bash, Grep, Glob.

```
**Workspace**: `./` persists across calls. `actions.json` is cleared
each call; other files accumulate. Feel free to save notes, state, or
helper functions.
**Observation tool**: `observe` --- returns the current settled board
and, when available, the executed actions and animation frames from
the preceding batch
**Game structure and strategy**: - Score increase means that a level
was cleared. - Most games have a step budget or timer mechanism,
which will cause a level reset if exceeded. - For parsing the boards,
programmatic options include identifying connected components (color,
position, size, shape) and forming testable hypotheses about what each
represents (player, walls, goals, UI, etc.).
**Response format**: a strategic briefing, then [PLAN] <2-3 sentence
action plan>
**Write `./actions.json`** with a JSON object {"actions":
["ACTION6(30,40)", "ACTION1", "RESET"]} --- a list of 1--action_cap
actions to execute in order. Entries beyond action_cap are discarded.
Prefer short lists (1--2 actions) when testing a new hypothesis so you
see the result before committing further; scale up toward action_cap
for proven sequences.
**Actions available in this game**: actions_section
The runner executes the list in order, then calls you again with an
updated board.
```

C.4.2 User Prompts

Initial User Prompt

```
Call `observe` to inspect the game.
This is the first analysis. Analyze the board returned by the tool
and write ./actions.json with your first set of actions.
```

Resume User Prompt

```
Score: score | Action: action_num | Level: level Last actions:
last_actions
Call `observe` to inspect the current boards. Check ./ for anything
you saved previously, then write a new ./actions.json.
```

C.4.3 MCP Interface

MCP Server Instruction

```
Inspect the game observation state.
```

'Observe' Tool

```
Name: observe
Arguments: none
Description: Return the current board and preceding action frames.
The first call returns:
{ "board": "<current settled board>" }
Later calls return:
{ "board": "<current settled board>", "executed_actions": ["ACTION1",
"ACTION2"], "frames": [ { "action_index": 1, "frame_index": 1,
"settled": false, "board": "<frame>" } ] }
```

C.5 NETHACK PROMPTS

The NetHack coding agent reads the following instructions from `AGENTS.md`. The environment provides its action list and rules separately in `instructions.txt`.

Shared Agent Instructions

```
{state_access} {workspace_access}
Status:  Dlvl is dungeon depth, $ is gold, HP is current/maximum
health, Pw is spell power, AC is armor class (lower is better), Xp
is experience level/points, and T is game turn.
NetHack reminders:  - NetHack is permadeath:  reaching 0 HP ends the
run.  Never take the up staircase on Dlvl 1; it exits the dungeon and
also ends the run.
- Monsters can act after each turn.  They may pursue, attack at range,
inflict conditions, or act faster than you.  Avoid trading attacks at
low HP when escape or recovery is available.
- Do not rest or search repeatedly while a hostile is nearby.
- A pickaxe is primarily a digging tool.  Digging may take several
turns while nearby monsters continue acting, so avoid digging near
hostiles and re-wield your combat weapon afterward.
- Hunger progresses from Hungry to Weak to Fainting; eat before Weak.
Corpses can rot, so only eat newly killed or known-safe corpses.
Burdened characters move more slowly.
- Prayer can resolve emergencies but has a long cooldown.  Engraving
Elbereth can deter many melee attackers.
- Shop merchandise becomes unpaid when picked up.  Pay for it or
return it before leaving, or the shopkeeper may become hostile.
- The Gnomish Mines and Sokoban are side branches, so progress can
require returning to the main dungeon.  Cave levels with gnomes and
dwarves indicate the Mines; a town of shops and a temple is Minetown;
boulder puzzles indicate Sokoban.
When ready, import submit from submit.py and call it once with the
chosen actions.  Submit at most 30 actions.  They execute in order
without intermediate model feedback.  Use 1-2 actions when testing
something uncertain; scale toward 30 for a safe, proven sequence.  The
runner stops the batch and asks again if you take damage or change
dungeon level.  It also stops if a new monster appears or one action
advances multiple game turns.
```

PRO-LONG: State and Workspace Access

```
Read instructions.txt for the environment rules and valid actions.
logs.txt contains the complete game history, including the current
observation at the end.  Read and search it rather than relying on
memory.

Use Python and shell tools to analyze the game state and available
history programmatically when useful.  You may save notes, state, or
helper functions in the workspace across calls.
```

No-Log: State and Workspace Access

```
Read instructions.txt for the environment rules and valid actions.
The current observation is included in each user message.

You may save notes and helper functions in the workspace across calls.
```

For multi-attempt runs, the next user prompt also includes:

Attempt Restart

```
A new attempt has started from the same seed.  Your goal remains to
maximize your chance of eventually ascending.  Use what you learned
```

```
from previous attempts to avoid repeating mistakes and improve your
strategy.
```

PRO-LONG: User Prompt

```
The game state has been appended to logs.txt.  Use Python or shell
tools when useful to analyze the current state or relevant history,
for example by parsing board coordinates, tracking explored routes and
stairs, comparing status over time, or solving spatial puzzles (e.g.
via BFS). When ready, submit the next action batch.
```

No-Log: User Prompt

```
[CURRENT OBSERVATION] {observation}
Inspect the current game state and submit the next action batch.
```

NetHack Environment Instructions

```
You are an agent playing NetHack.  The following are the common named
actions you can take in the game, followed by a short description of
each action:
{action_strings}.
Tips:  - When the message asks for a completion, such as:  "What
do you want to eat?  [d or ?*]", you should respond with a single
character corresponding to the item you want to eat/use.
- For example, "What do you want to eat?  [dgh or ?*]" -> Possible
answers are "d", "g", or "h" to eat the associated food.
- NetHack reads one key per action.  Enter counts one digit at a
time.  To type free text, submit one action per character followed
by "enter".  For example, engraving Elbereth is ["engrave", "minus",
"E", "l", "b", "e", "r", "e", "t", "h", "enter"].
- When the message asks for a direction, such as:  "In what
direction?" you should respond with a direction.
- When the message has --More-- at the end, your next action should be
"more" to see the rest of the message.
- Explore the environment to find the stairs down to the next level.
- Always carefully read the last message to understand the current
state of the game and decide your next action accordingly.
- If you keep moving in the same direction, you will eventually hit
a wall and stop moving.  Your message might be:  "It's solid stone",
or "It's a wall".  Change your action to move in another direction to
continue exploring the environment.
- Read the language observation carefully and look at ascii map or
image observation provided to decide the next action to take and where
to move next.
- You can attack monsters by moving into them.
Your goal is to maximize your chance of eventually ascending.  Make
sustained progress by surviving, gaining experience and useful
equipment, and descending when prepared.  Depth matters, but so do
survival, experience, and equipment.
```