MineEvolve: Self-Evolution with Accumulated Knowledge for Long-Horizon Embodied Minecraft Agents

Zhengwei Xie[1,*], **Zhisheng Chen**[2,*], **Ziyan Weng**[3,*], **Jinhan Li**[1], **Chenglong Li**[4], **Zikai Xiao**[5], **Guibin Zhang**, **Jinhao Jing**[7], **Jingwei Song**[6,†], **Kun Wang**[2,†]

[1]University of Science and Technology of China, [2]Nanyang Technological University, [3]City University of Hong Kong (Dongguan), [4]Tsinghua University, [5]Zhejiang University, [6]Shanghai Jiaotong University, [7]The Chinese University of Hong Kong, Shenzhen

[*]Equal contribution. [†]Corresponding author.

xiezhengwei0307@gmail.com, zhisheng.researcher@gmail.com.

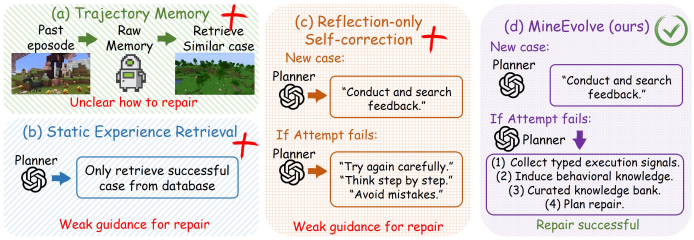


Figure 1: Prior methods retrieve raw trajectories, reuse static successful cases, or rely on generic text reflections, which rarely yield actionable guidance for repairing failures. MineEvolve instead collects typed execution signals, induces behavioral knowledge, curates it in a knowledge bank, and uses it to repair the plan, turning past experience into explicit guidance for future decisions.

Abstract

Long-horizon embodied intelligence requires agents to improve through interaction, not merely to execute plans generated from static goals. A central challenge is therefore to transform past executions into knowledge that can shape future decisions. Minecraft provides a representative testbed for this problem, where tasks such as crafting tools, building redstone components, and obtaining diamond equipment involve long prerequisite chains and are frequently disrupted by missing tools, blocked paths, GUI failures, or stagnant execution. To this end, we propose **MineEvolve**, a knowledge-driven self-evolution framework that converts execution feedback into actionable behavioral knowledge. MineEvolve first uses ❶***Monitor*** to convert each subgoal execution into typed feedback, including state changes, inventory changes, failure types, progress signals, and stagnation indicators. ❷***Inducer*** then derives reusable skills from successful executions and remedies from failed or stagnant executions. ❸***Curator*** validates, merges, filters, and retrieves these knowledge entries, while ❹***Adaptor*** uses them to repair the unfinished part of the plan under repeated failures or stagnation. Experiments on the Minecraft MCU long-horizon task suite show that MineEvolve improves performance over the strongest baseline on all five planner backbones, with paired gains whose confidence intervals exclude zero on three of them and with larger gains on high-dependency task groups. Ablation and knowledge-accumulation studies further demonstrate that converting execution signals into structured behavioral knowledge is an effective path toward self-evolving embodied agents in long-horizon environments. Our code is available at https://github.com/xzw-ustc/MC-MineEvolve.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



Figure 2: Progression of Minecraft from wooden to diamond tiers, illustrating the hierarchy of tools, materials, and equipment used to construct long-horizon tasks for embodied agent evaluation.

1 Introduction

Embodied agents are increasingly expected to perform long-horizon tasks in open and interactive environments, rather than merely follow short-horizon instructions [Gupta et al., 2021, Liu et al., 2025, Luo et al., 2025]. In such settings, an agent's ability depends not only on generating an initial plan from a goal, but also on converting past successes and failures into knowledge that can guide future behavior. Minecraft provides a representative test platform for this problem [Johnson et al., 2016, Guss et al., 2019, Fan et al., 2022, Zheng et al., 2025]. Tasks such as crafting iron tools, building redstone components, and obtaining diamond equipment require long chains of prerequisites, including wood collection, basic tool crafting, ore search, smelting, crafting, and navigation. Failures in these tasks often arise not from a language model's misunderstanding of the goal, but from fine-grained execution errors, such as missing tools, blocked paths, abnormal GUI states, or omitted prerequisites. Although recent large language models have improved agents' ability to understand goals, decompose tasks, and use tools, a key question remains in long-dependency environments [Brown et al., 2020, Achiam et al., 2023, Huang et al., 2022, Yao et al., 2022, Wang et al., 2026]:

> *How can an agent transform execution feedback into behavioral knowledge that is attributable, verifiable, and directly actionable for future planning?*

We address this question with **MineEvolve**, a knowledge-driven self-evolution framework for long-horizon embodied agents in Minecraft, which is achieved by continually constructing and using actionable knowledge from the agent's own executions. The framework follows four sequential steps: ❶***Monitor*** converts subgoal executions into typed execution feedback; ❷***Inducer*** derives skills from successful executions and remedies from failed or stagnant executions; ❸***Curator*** validates, merges, filters, and retrieves the generated knowledge; and ❹***Adaptor*** conditions planning on the retrieved knowledge and repairs the remaining plan when repeated failures or stagnation occur.

MineEvolve centers on converting *execution feedback* into *actionable knowledge* [Shinn et al., 2023, Zhao et al., 2024, Madaan et al., 2023]. During execution, state transitions, inventory changes, failure types, progress signals, and stagnation indicators provide structured information about the consequences of the agent's actions. MineEvolve organizes such feedback into two complementary forms of knowledge. Successful feedback is abstracted into skills, which encode preconditions, action sequences, and verification rules. Failed or stagnant feedback is transformed into remedies, which specify triggers, risk patterns, and repair actions. Skills support the reuse of effective behaviors, while remedies guide plan revision after local failures.

This design goes beyond trajectory memory and static experience retrieval. Rather than only retrieving similar past cases, MineEvolve identifies how past executions should modify the current plan [Park

et al., 2023, Packer et al., 2023, Lewis et al., 2020, Kagaya et al., 2024]. For example, repeated movement near a target block without inventory gain can be converted into a remedy that suggests clearing blocking blocks or selecting an alternative route. A crafting failure caused by a missing crafting table can be converted into a prerequisite repair step before retrying the recipe. Thus, failures are not treated as passive records of unsuccessful attempts, but are converted into actionable knowledge that guides future plan revision.

As shown in Figure 2, we evaluate MineEvolve on the MCU long-horizon task suite [Zheng et al., 2025]. Across five language-model planners, MineEvolve outperforms static experience retrieval baselines and exceeds the strongest system-level baseline by a margin whose paired confidence interval excludes zero on three of them, with the largest improvements on high-dependency task groups such as Iron, Redstone, Diamond, and Armor. Our main contributions are summarized as follows:

- **A knowledge-driven self-evolution framework for long-horizon Minecraft agents.** We introduce MineEvolve, which improves an agent through the continual accumulation and use of external behavioral knowledge. By grounding self-evolution in execution feedback, MineEvolve enables agents to convert past successes and failures into task knowledge that can guide future planning and local repair.
- **Feedback-conditioned generation of skills and remedies.** MineEvolve converts each execution into typed feedback through *Monitor*, capturing state changes, inventory changes, failure types, progress signals, and stagnation indicators. Based on this feedback, *Inducer* derives reusable skills from successful executions and remedies from failed or stagnant executions, producing knowledge with explicit triggers, verification rules, and repair actions.
- **Knowledge-guided repair of local failures.** MineEvolve maintains generated skills and remedies through *Curator*, which validates, merges, filters, and retrieves knowledge relevant to the current context. The retrieved knowledge is then used by *Adaptor* to revise only the remaining subgoals when repeated failures or stagnation occur, reducing ineffective retries and improving the stability of long-horizon task execution.

2 Related Work

2.1 Minecraft Embodied Agents and Open-World Benchmarks

Early research on Minecraft embodied agents mainly focused on reinforcement learning, imitation learning, and human demonstrations [Johnson et al., 2016, Guss et al., 2019], building on general reinforcement-learning foundations [Sutton et al., 1998]. MineRL provides large-scale human demonstration data for studying sample-efficient learning in complex Minecraft tasks [Guss et al., 2019]. MineDojo further extends Minecraft into an open-ended embodied agent platform by integrating language-specified tasks, environment interaction, and internet-scale knowledge from videos, Wiki pages, and community content [Fan et al., 2022]. For low-level behavior modeling, VPT learns visual-to-keyboard-and-mouse control policies from large-scale video pretraining [Baker et al., 2022], while STEVE-1 instruction-tunes this paradigm to enable agents to follow short-horizon text or visual instructions [Lifshitz et al., 2023]. These works provide important datasets, environments, and low-level control foundations for Minecraft agents, but they mainly focus on demonstration learning, behavior generation, or short-horizon instruction following, leaving long-dependency planning, failure recovery, and cross-task experience transformation less explored.

2.2 LLM Planning and Embodied Control in Minecraft

With the progress of large language models in task understanding and reasoning [Brown et al., 2020, Achiam et al., 2023, Huang et al., 2022, Yao et al., 2022], Minecraft agents have increasingly adopted a hierarchical architecture with a high-level LLM planner and a low-level controller [Ahn et al., 2022, Liang et al., 2023, Huang et al., 2022]. DEPS improves subgoal generation and selection in open-world tasks through a Describe–Explain–Plan–Select pipeline, and further uses execution descriptions and failure self-explanations to iteratively revise plans [Wang et al., 2023b]. JARVIS-1 integrates visual observations, language instructions, task planning, and embodied control into a unified framework for multimodal Minecraft task execution, and is memory-augmented: it maintains a multimodal memory of real gameplay experience that conditions later planning [Wang et al., 2024].

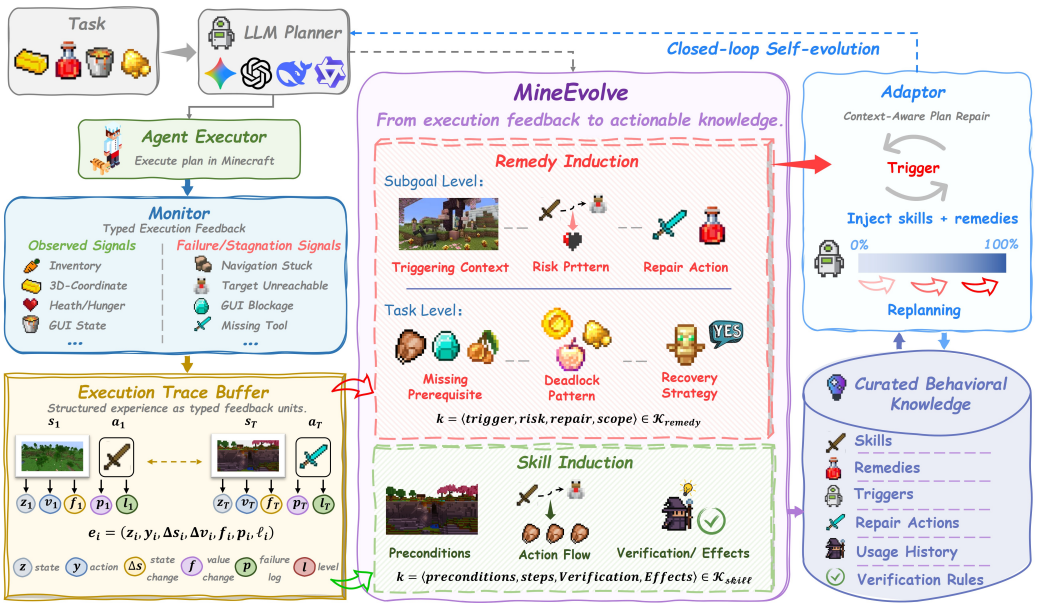


Figure 3: MineEvolve models self-evolution as the continual update and use of an external behavioral knowledge system, and consists of four sequential operations. First, ***Monitor*** converts each subgoal execution into typed execution feedback. Second, ***Inducer*** generates skills and remedies from this feedback. Third, ***Curator*** validates, merges, filters, and retrieves candidate knowledge. Finally, ***Adaptor*** injects the retrieved knowledge into the planner and repairs the unfinished part of the plan when repeated failures or stagnation occur.

Recent work further strengthens the alignment between goals, observations, and actions. Optimus-2 combines multimodal LLM planning with a goal–observation–action conditioned policy, and uses an action-guided behavior encoder to improve embodied execution [Li et al., 2025b]. These methods mainly address how to generate executable behaviors from language goals and visual observations, and substantially improve task decomposition, multimodal perception, and action execution [Radford et al., 2021, Liu et al., 2023]. However, long-horizon tasks also require agents to revise future plans based on execution feedback collected during interaction, which motivates research on experience memory and self-improvement.

2.3 Experience Memory and Self-Improvement

In Minecraft, Voyager uses an automatic curriculum, environment feedback, and a growing code skill library to accumulate transferable skills [Wang et al., 2023a]. Optimus-1 introduces hybrid multimodal memory, organizing world knowledge as a hierarchical knowledge graph and storing historical experience summaries in a multimodal experience pool to support long-horizon planning and reflection [Li et al., 2025a]. More recently, XENON treats long-horizon planning errors as external knowledge bias, and uses successful and failed experiences to revise dependency knowledge and action knowledge [Lee et al., 2026]. These methods show that experience use in Minecraft agents is moving from trajectory retrieval and experience summarization [Shinn et al., 2023, Zhao et al., 2024, Madaan et al., 2023, Park et al., 2023, Packer et al., 2023] toward knowledge organization and knowledge revision [Lewis et al., 2020, Kagaya et al., 2024, Parisi et al., 2019]. However, existing methods mostly focus on successful skill reuse, historical summaries, or dependency correction, while the conversion of fine-grained execution feedback into behavioral knowledge for plan repair remains underexplored. In contrast, MineEvolve focuses on transforming execution feedback into planning-time knowledge: each subgoal execution is first represented as typed feedback, then induced into skills or remedies, and finally used for knowledge-conditioned plan repair.

3 Method

3.1 MineEvolve Overview

As shown in Figure 3, MineEvolve targets long-horizon embodied tasks in Minecraft. Its goal is to help an agent accumulate and use external behavioral knowledge through interaction, without updating the parameters of the language model [Packer et al., 2023, Lewis et al., 2020, Parisi et al., 2019]. Given a task goal g and the current state s_t, the language-model planner generates a sequence of high-level subgoals, and a low-level executor carries them out in the environment. Instead of treating each attempt only as a success or failure, MineEvolve extracts typed execution feedback from each subgoal execution. It then turns successful feedback into *skills* and failed or stagnant feedback into *remedies*. Skills reuse verified successful procedures, while remedies describe risk patterns and repair actions.

3.2 Feedback-Conditioned Skill and Remedy Generation

For the i-th subgoal z_i, ***Monitor*** compresses its execution into typed execution feedback:

$$e_i = \big(z_i,\ y_i,\ \Delta s_i,\ \Delta v_i,\ f_i,\ p_i,\ \ell_i\big), \tag{1}$$

where $y_i \in \{0, 1\}$ denotes whether the subgoal succeeds, Δs_i denotes the state change before and after execution, Δv_i denotes the inventory change, f_i denotes the failure type, p_i denotes execution progress, and ℓ_i indicates stagnation or looping. Unlike free-form reflection [Shinn et al., 2023, Madaan et al., 2023], each field in this representation comes from environment states, inventory changes, or execution monitors, and can therefore be directly used by the knowledge generation process.

To capture failures of low-progress executions, Monitor combines movement, inventory change, and task-specific progress. We define the normalized progress of the i-th execution as

$$p_i = \lambda_x \frac{\mathrm{Var}(x_{i,1:T})}{\epsilon_x + \mathrm{Var}(x_{i,1:T})} + \lambda_v \frac{\|\Delta v_i\|_1}{\epsilon_v + \|\Delta v_i\|_1} + \lambda_g p_i^{\mathrm{goal}}, \tag{2}$$

where $x_{i,1:T}$ is the coordinate sequence during the execution of z_i, Δv_i is the inventory change, and p_i^{goal} is task-specific progress, such as crafting progress, GUI-state changes, or smelting progress. The coefficients $\lambda_x, \lambda_v, \lambda_g$ are weights. If progress is below a threshold and the subgoal is not completed, the execution is marked as stagnant:

$$\ell_i = \mathbb{I}\left[p_i < \epsilon_p \ \wedge \ y_i = 0\right]. \tag{3}$$

This rule allows the agent to detect low-progress behaviors such as repeated movement without effective approach, unchanged inventory, or GUI operations that produce no result.

Given typed execution feedback, ***Inducer*** generates two forms of behavioral knowledge. A *skill* is induced from successful feedback segments and describes the trigger context, preconditions, steps, verification rule, and observed effects for a task. A *remedy* is induced from failed or stagnant feedback segments and describes the trigger context, failure type, risk pattern, repair action, and scope. Both are represented as a unified knowledge entry:

$$k = \big(\mathrm{type},\ c,\ u,\ \phi,\ \rho,\ E\big), \qquad \mathrm{type} \in \{\mathrm{skill}, \mathrm{remedy}\}. \tag{4}$$

Here, c is the trigger context, u is the knowledge content, ϕ is a verification or applicability condition, ρ is the confidence score, and E is the supporting set of execution feedback. For a skill, u denotes reusable subgoal steps. For a remedy, u denotes a risk pattern and the corresponding repair action. *The detailed pseudocode is shown in Appendix C.2.*

When a feedback segment contains consecutive successful subgoals and its final state satisfies the corresponding verification rule, Inducer generates a skill. For example, if the agent successfully completes the sequence of collecting logs, crafting planks, and crafting sticks, and the target item appears in the inventory, the sequence can be induced as a reusable skill. In contrast, when failures repeat in a recent window or the current subgoal is stagnant, Inducer generates a remedy. The trigger is defined as

$$\frac{1}{h} \sum_{j=i-h+1}^{i} \mathbb{I}[y_j = 0] \geq \eta_{\mathrm{fail}} \quad \text{or} \quad \ell_i = 1, \tag{5}$$

where h is the recent execution window and η_{fail} is the failure-rate threshold. This condition prevents a single accidental failure from being immediately stored as a remedy, while still allowing repeated failure patterns or clear stagnation to be captured.

To prevent vague, conflicting, or non-executable text from entering the knowledge store, ***Curator*** validates each candidate knowledge entry k. Validation includes schema completeness, context matchability, executability, specificity, and conflict filtering. We define the acceptance rule as

$$\text{Accept}(k, \mathcal{K}) = \mathbb{I}\left[V_{\text{schema}}(k) \wedge V_{\text{match}}(k) \wedge V_{\text{exec}}(k) \wedge V_{\text{spec}}(k) \wedge \neg C_{\text{conflict}}(k, \mathcal{K})\right], \quad (6)$$

where V_{schema} checks whether all required fields are present, V_{match} checks whether the trigger can be matched to environment-state fields, V_{exec} checks whether the skill steps or repair action can be executed by the executor, V_{spec} rejects overly generic suggestions such as "try again" or "be more careful", and C_{conflict} checks whether the candidate conflicts with existing high-confidence knowledge. Only accepted candidates are written into the external knowledge store; otherwise, they remain only as feedback records for the current episode. We audited this layer in practice: across 400 retrieved-remedy events, 18 entries (4.5%) did not match the complete current state, and the applicability check in Adaptor removed 15 of them, leaving 3 (0.75%) that entered repair constraints. We therefore treat 0.75% as the effective false-positive rate of retrieved knowledge at the scale evaluated here.

3.3 Knowledge-guided repair of local failures

Curator maintains an external knowledge store $\mathcal{K}$. Each entry contains a trigger context, knowledge content, supporting feedback, confidence, support count, and usage history. Before planning, Curator retrieves relevant skills and remedies according to the current task goal, state, subgoal, inventory condition, spatial context, and recent failure types. To prevent the planning context from growing without bound, retrieval is constrained by a fixed prompt budget:

$$\mathcal{K}_i^R = \arg\max_{\mathcal{S} \subseteq \mathcal{K}} \sum_{k \in \mathcal{S}} r(c_i, k), \qquad \text{s.t.} \quad \sum_{k \in \mathcal{S}} \text{tokens}(k) \leq B, \quad (7)$$

where c_i is the planning context at the i-th subgoal, $r(c_i, k)$ measures the relevance between knowledge entry k and the current context, and B is the prompt budget. Retrieved skills provide positive guidance for the planner, while remedies provide risk information and repair actions.

When repeated failures or stagnation occur, ***Adaptor*** does not ask the language model to regenerate the entire plan. Instead, it keeps the completed prefix and repairs the unfinished part. Let the current plan be $\mathbf{z}_{1:N}$ and let the prefix $\mathbf{z}_{1:i}$ be completed or retained. Adaptor produces the repaired plan as

$$\mathbf{z}'_{1:N} = \left[\mathbf{z}_{1:i},\ \text{Repair}\left(g,\ s_i,\ \mathbf{z}_{1:i},\ \mathcal{K}_i^R,\ \mathcal{A}_i\right)\right], \quad (8)$$

where $\mathcal{A}_i$ is the active set of remedies, including relevant remedies retrieved from the knowledge store and a temporary remedy induced from the current failure feedback. This design preserves useful progress while allowing new execution feedback to affect future planning. *The detailed pseudocode is shown in Appendix C.3.*

When the agent repeatedly moves near a target block without any inventory change, MineEvolve identifies the pattern as a low-progress navigation failure and generates a remedy that asks the planner to avoid repeating the same path, clear blocking blocks, or choose an alternative route. When the agent fails to craft a target tool because a crafting table is missing, Adaptor inserts the prerequisite step of obtaining or placing a crafting table into the unfinished plan. Thus, MineEvolve does not simply ask the language model to "try again"; it repairs the unfinished plan under explicit remedies derived from feedback, reducing ineffective retries and improving long-horizon execution stability.

4 Experiments

4.1 Experimental Setup

Environment and tasks. We evaluate MineEvolve on a 70-task subset of the Minecraft MCU tech-tree benchmark [Zheng et al., 2025], which contains 3,452 compositional atomic tasks. The subset was fixed once, before MineEvolve was implemented and before any results were inspected,

Table 1: Compact main results on the MCU task suite. Values denote success rate SR (%). Easy Avg. is the task-count-weighted average over Wooden, Stone, and Gold tasks; Hard Avg. is the task-count-weighted average over Iron, Redstone, Diamond, and Armor tasks; Overall is the task-count-weighted average over all 70 tasks. Best scores are highlighted in orange, and second-best scores are highlighted in teal.

Method	Qwen3.5-Flash	Qwen3.5-Plus	GLM-4.7	Gemini-3-Flash	GPT-5.5
Weighted Easy Task Average					
DEPS	57.69	58.65	56.60	59.07	61.34
JARVIS-1	70.47	71.04	69.30	70.62	72.41
Optimus-1	73.76	73.66	72.55	73.26	75.29
Optimus-2 (GOAP)	73.93	74.50	**73.28**	74.86	75.60
MineEvolve	**74.14**	**75.61**	72.97	**75.47**	**76.98**
Weighted Hard Task Average					
DEPS	9.71	9.66	8.82	10.06	11.92
JARVIS-1	22.60	23.62	21.79	22.98	25.07
Optimus-1	27.21	29.93	27.01	29.04	31.35
Optimus-2 (GOAP)	32.11	33.25	31.25	34.19	35.70
MineEvolve	**34.06**	**37.13**	**32.06**	**36.42**	**40.23**
Weighted Overall					
DEPS	28.90	29.25	27.93	29.66	31.69
JARVIS-1	41.75	42.59	40.79	42.04	44.01
Optimus-1	45.83	47.42	45.23	46.73	48.92
Optimus-2 (GOAP)	48.84	49.75	48.06	50.45	51.66
MineEvolve	**50.09**	**52.52**	**48.43**	**52.04**	**54.93**

using five rule-based criteria: survival tech-tree relevance, item/state-checker verifiability, use of the shared observation and action interface, no creative mode, oracle inventory, or method-specific privileged actions, and coverage from basic resources to Diamond and Armor. We therefore treat it as a rule-based technology-tree subset rather than a representative sample of the full benchmark. The tasks span seven technology-tree groups, from early-stage Wooden, Stone, and Gold tasks to more difficult Iron, Redstone, Diamond, and Armor tasks with longer prerequisite chains. Agents operate in a survival-style environment with first-person RGB observations, structured state inputs, and an action space covering navigation, mining, placing, item usage, inventory control, and crafting. All episodes start with an empty inventory unless otherwise specified. We report Overall success rate (SR) over all task groups and additionally focus on the hard groups, where prerequisite acquisition, recovery, and failure propagation are more prominent. *Detailed task-group statistics, representative tasks, and averaging rules are provided in Appendix D.1.*

Baselines and protocol control. We compare MineEvolve with DEPS, JARVIS-1, Optimus-1, and Optimus-2 [Wang et al., 2023b, 2024, Li et al., 2025a,b] under the same task set, task–seed split, episode horizon, and success criterion. DEPS, JARVIS-1, Optimus-1, and MineEvolve use STEVE-1 as the low-level execution policy [Lifshitz et al., 2023], while Optimus-2 uses its native GOAP-based policy. For STEVE-1-based methods, we additionally control the observation/action interface, action primitives, structured state fields, retrieval top-K, memory-token budget, and evaluation-time LLM-call budget.

4.2 Main Experimental Results

Table 1 reports the main results on the MCU task suite. To highlight the benefits on long dependency chains, the main table reports three metrics, Easy Avg., Hard Avg., and Overall. All aggregate metrics are computed at the task level rather than by assigning equal weight to task groups. Specifically, Easy Avg., Hard Avg., and Overall weight each group by the number of tasks it contains. *The complete results for all seven task groups are provided in Appendix Table 5.*

Under the unified MCU task protocol, MineEvolve achieves the highest Overall and Hard Avg. across five planner backbones. Table 2 reports the paired gains over Optimus-2, whose 95% intervals exclude zero on three of the five backbones, so the gains arise mainly from transforming execution

feedback into behavioral knowledge for conditioned local repair rather than from a single language model. The improvement is more pronounced on difficult task groups, indicating that remedies are particularly effective in repairing missing prerequisites, navigation stagnation, and crafting failures in long dependency chains.

All methods and backbones share 10 fixed world seeds; within each seed every Easy task is repeated with 10 and every Hard task with 14 distinct agent seeds, giving $R_t = 100$ and $R_t = 140$ and 8,680 matched identifiers per method. We aggregate trials to task-level SRs and report paired task-clustered bootstrap intervals (10,000 resamples) with paired sign-flip tests (200,000 draws). Since the task is the cluster and repeats are averaged first, these intervals capture cross-task uncertainty after fixed-seed averaging rather than seed-to-seed variation. Task-composition sensitivity confirms this: with the seven task groups equally weighted, MineEvolve remains ahead of Optimus-2 on all five backbones by 0.36–3.36 points, and across 35 leave-one-group-out settings (five backbones $\times$ seven removed groups), 34 are positive and one ties at two decimals.

Table 2: Paired gains of MineEvolve over the strongest system-level baseline (Optimus-2), with 95% paired task-clustered bootstrap intervals.

Backbone	Hard gain [95% CI]	Overall gain [95% CI]
Qwen3.5-Flash	$+1.95\ [-0.18, 4.06]$	$+1.25\ [-0.18, 2.63]$
Qwen3.5-Plus	$+3.88\ [2.18, 5.73]$	$+2.77\ [1.66, 3.91]$
GLM-4.7	$+0.81\ [-1.20, 2.77]$	$+0.37\ [-0.91, 1.62]$
Gemini-3-Flash	$+2.23\ [0.09, 4.27]$	$+1.59\ [0.30, 2.88]$
GPT-5.5	$+4.53\ [2.84, 6.23]$	$+3.27\ [2.22, 4.34]$

4.3 Controlled Ablation Study

To verify the source of the gains, we conduct targeted ablations using the Qwen3.5-Plus planner and the same STEVE-1 low-level execution policy. Each variant corresponds to one key design in MineEvolve: the Monitor provides typed execution feedback, the Inducer generates skills and remedies, the Curator verifies and maintains knowledge, and the Adaptor performs local repair. All variants use the same planner, low-level execution policy, interaction horizon, retrieval top-K, and evaluation-time LLM-call budget.

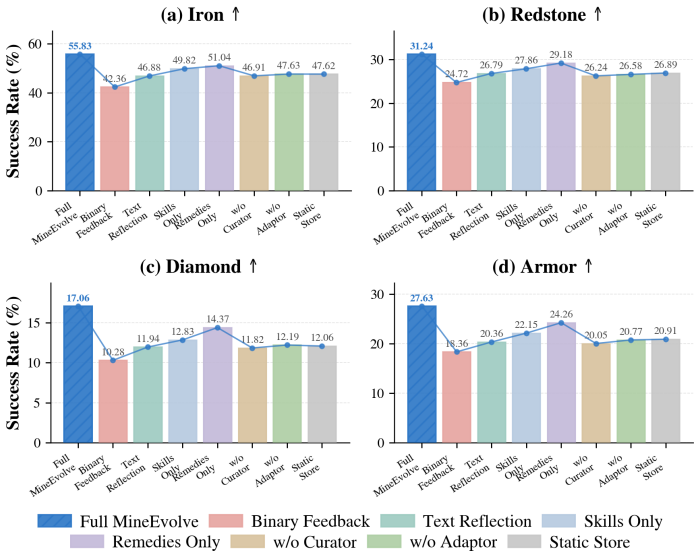


Figure 4: Targeted ablation results on the four difficult task groups. Bars denote SR (%). MineEvolve consistently outperforms ablated variants across Iron, Redstone, Diamond, and Armor tasks.

Figure 4 shows the targeted ablation results on the four difficult task groups. Binary Feedback keeps only binary success/failure signals and is used to verify the necessity of typed execution feedback.

Text Reflection replaces structured skills and remedies with free-form textual reflection. Skills Only and Remedies Only isolate the effects of reusing successful experience and failure-recovery knowledge, respectively. w/o Curator removes verification, merging, and conflict filtering. w/o Adaptor injects retrieved knowledge only during initial planning and does not perform local repair. Static Store freezes the knowledge base and is used to examine whether continuous knowledge updates truly bring self-evolutionary gains.

The ablation results show that each component contributes to performance on difficult tasks. Removing typed execution feedback causes the largest degradation across all four task groups, with Binary Feedback consistently performing worst. Replacing structured knowledge with free-form reflection improves over binary feedback but still remains substantially below the full system, suggesting that natural-language summaries alone are insufficient for reliable knowledge reuse. Skills Only and Remedies Only both improve over reflection, while the full system performs best, indicating that successful-experience reuse and failure-conditioned recovery knowledge are complementary. Remedies Only consistently outperforms Skills Only, suggesting that failure-conditioned recovery knowledge is especially important for difficult long-horizon tasks. Removing the Curator, the Adaptor, or continuous knowledge updates also leads to clear drops, showing that MineEvolve benefits not only from storing experience but also from verifying, updating, and applying it through local repair.

Table 3: Ablation on feedback granularity. All variants share the same low-level execution policy, knowledge-base size, retrieval budget, replanning trigger conditions, and evaluation-time LLM-call budget. Values denote SR (%).

Source of Remedies	Iron	Redstone	Diamond	Armor	Hard Avg.
Binary Feedback	42.36	24.72	10.28	18.36	27.06
Trajectory-level Reflection	46.88	26.79	11.94	20.36	29.98
Typed Execution Feedback	**55.83**	**31.24**	**17.06**	**27.63**	**37.13**

Table 3 further examines how remedy effectiveness depends on the granularity of execution feedback. Typed Execution Feedback substantially outperforms both binary feedback and trajectory-level reflection across all difficult task groups. This indicates that the effectiveness of remedies does not merely come from asking the LLM to summarize past trajectories, but from execution-feedback fields that are matchable, verifiable, and directly actionable during future planning and local repair.

4.4 Knowledge Accumulation and Analysis

To examine the self-evolving behavior of MineEvolve, we freeze the external knowledge base at different experience scales and measure same-task knowledge reuse on unseen world instances: construction and evaluation share the same 42 hard tasks but draw from non-overlapping world-seed pools, so these curves reflect accumulation and reuse rather than transfer to unseen tasks. Task-disjoint transfer is evaluated separately in Appendix D.5. The agent first interacts with the environment for M episodes, and the knowledge base is fixed at checkpoints with $M \in \{0, 50, 100, 200, 400\}$. During evaluation, no further knowledge is updated or generated. This setting disentangles the gains from knowledge accumulation from online learning during evaluation. Figure 5 shows the knowledge accumulation curves on the four difficult task groups, i.e., Iron, Redstone, Diamond, and Armor, and *Appendix Table 7 reports the corresponding numerical results.*

As shown in Figure 5, all three methods improve to some extent on the four difficult task groups as the number of accumulated episodes increases, but the magnitude of improvement differs substantially. Static Store exhibits only limited growth, suggesting that relying solely on static retrieval or weakly updated knowledge is insufficient to fully exploit subsequent interaction experience. Text Reflection brings certain gains across task groups, but its improvement remains limited because free-form textual reflection lacks explicit triggering conditions, verification rules, and repair actions. In contrast, MineEvolve achieves more pronounced gains across all task groups, indicating that abstracting execution feedback into structured skills and remedies is better suited for sustained capability growth in long-horizon tasks.

The gains also vary meaningfully across task groups. The Iron group starts and ends at relatively high performance, because its dependency chain is shorter than those of Diamond and Armor, and its

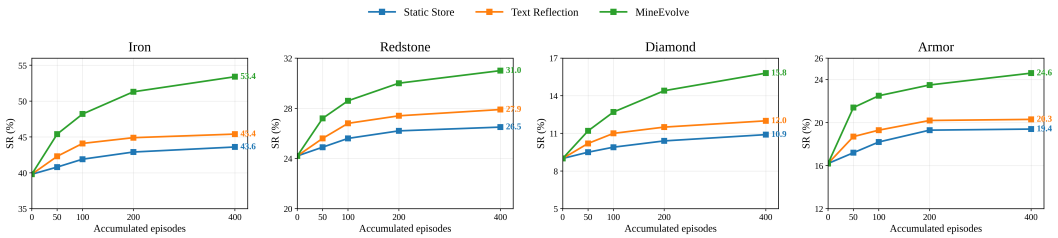


Figure 5: Knowledge accumulation curves on the four difficult task groups. At each checkpoint, the external knowledge base is frozen before evaluation. Compared with Static Store and Text Reflection, MineEvolve shows more pronounced and sustained improvement on Iron, Redstone, Diamond, and Armor, indicating that structured skills and remedies can be accumulated and reused within the same task groups on unseen worlds.

tool-making and smelting procedures are easier to reuse. Diamond remains the most difficult task group throughout, but MineEvolve still improves steadily with accumulated experience, suggesting that remedies provide meaningful repair for sparse resources, tool prerequisites, and failures in underground search. The Armor group shows substantial improvement under MineEvolve, mainly benefiting from prerequisite completion, reuse of smelting procedures, and repair of crafting failures. The Redstone group has intermediate difficulty, and structured knowledge helps handle component crafting and multi-step dependencies.

In addition, *Appendix Table 8* reports the knowledge-base size, retrieval tokens, retrieval latency, and the number of evaluation-time LLM calls, to verify that the budget control of the Curator can prevent unbounded context growth. *Appendix Section D.5* further analyzes the curriculum-style knowledge construction strategy on Diamond tasks, and *Appendix Section D.6* provides qualitative case studies to illustrate how remedies transform failure feedback into plan repair.

5 Conclusion

We presented MineEvolve, a knowledge-driven self-evolution framework for long-horizon embodied agents in Minecraft. MineEvolve improves agents not by updating language-model parameters, but by continually converting execution feedback into actionable behavioral knowledge. Through monitor, inducer, curator, and adaptor, successful executions are distilled into reusable skills, while failed or stagnant executions are transformed into remedies for local plan repair. Experiments on the MCU long-horizon task suite show that MineEvolve improves performance over the strongest baseline on all five planner backbones, and significantly so on three, especially on high-dependency task groups such as Iron, Redstone, Diamond, and Armor. Ablations and knowledge-accumulation studies further confirm that typed feedback, skill/remedy generation, knowledge curation, and local repair are all important for sustained improvement. These results suggest that structured external behavioral knowledge provides an effective path toward self-evolving embodied agents in long-horizon environments.

Limitations. The main evaluation covers a fixed 70-task subset of the 3,452-task Minecraft benchmark. Accumulation curves assess reuse on seen tasks and may depend on curriculum order, while cross-domain generalization remains unvalidated. Performance depends on the planner and controller, with computational overhead from knowledge construction. Unseen failures require reactive repair, and safety-critical deployment would require independent safeguards. We did not run a direct empirical comparison with XENON, the closest prior system: it uses binary success/failure feedback without failure typing, repair-action content, or a validation layer, so the comparison is mechanistic and left to future work.

Ethics and broader impact. MineEvolve is a planning framework for a sandbox game rather than a deployed application. Improved long-horizon recovery reduces the human supervision embodied agents require, but the same capability could transfer to settings where autonomous failure recovery has safety implications. We release only code and schemas, without model weights.

Acknowledgments and Disclosure of Funding

This work was supported by computational resources from Nanyang Technological University. We gratefully acknowledge financial assistance from NeurIPS for conference attendance.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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

Agrim Gupta, Silvio Savarese, Surya Ganguli, and Li Fei-Fei. Embodied intelligence via learning and evolution. *Nature communications*, 12(1):5721, 2021.

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

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pages 9118–9147. PMLR, 2022.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *Ijcai*, volume 16, pages 4246–4247, 2016.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. *arXiv preprint arXiv:2402.03610*, 2024.

Seungjoon Lee, Suhwan Kim, Minhyeon Oh, Youngsik Yoon, and Jungseul Ok. Experience-based knowledge correction for robust planning in minecraft. In *International Conference on Learning Representations*, volume 2026, pages 137856–137908, 2026.

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.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks. *Advances in neural information processing systems*, 37:49881–49913, 2025a.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-2: Multimodal minecraft agent with goal-observation-action conditioned policy. In *Proceedings of the computer vision and pattern recognition conference*, pages 9039–9049, 2025b.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023.

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

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023.

Huaping Liu, Di Guo, and Angelo Cangelosi. Embodied intelligence: A synergy of morphology, action, perception and learning. *ACM Computing Surveys*, 57(7):1–36, 2025.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, et al. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback, 2023. *URL https://arxiv. org/abs/2303.17651*, 2303, 2023.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

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.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 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.

Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

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.

Zehong Wang, Fang Wu, Hongru Wang, Xiangru Tang, Bolian Li, Zhenfei Yin, Yijun Ma, Yiyang Li, Weixiang Sun, Xiusi Chen, et al. Why reasoning fails to plan: A planning-centric analysis of long-horizon decision making in llm agents. *arXiv preprint arXiv:2601.22311*, 2026.

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.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

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.

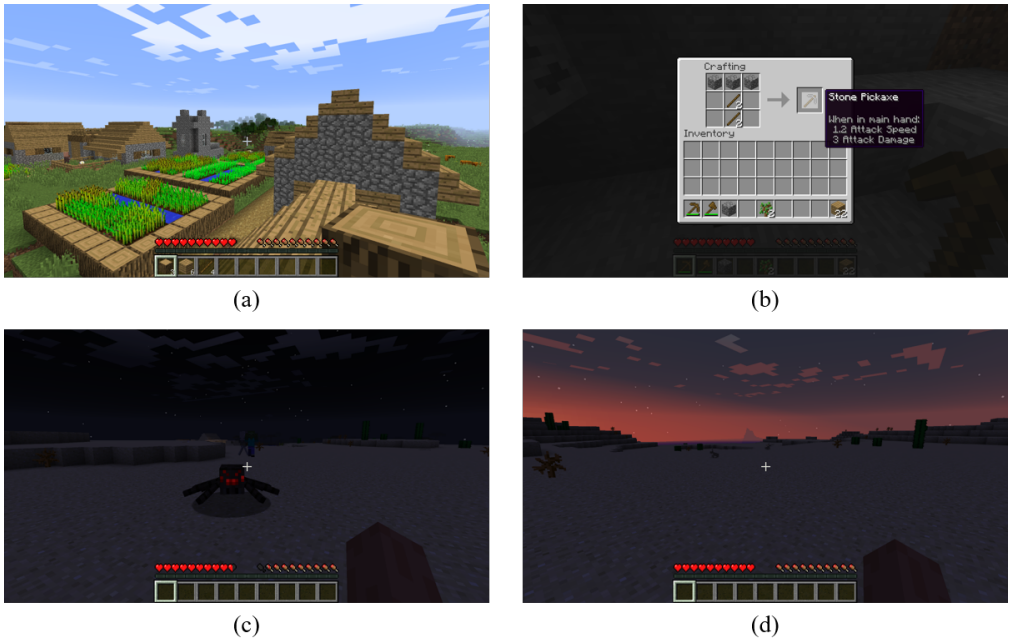
Xinyue Zheng, Haowei Lin, Kaichen He, Zihao Wang, Qiang Fu, Haobo Fu, Zilong Zheng, and Yitao Liang. Mcu: An evaluation framework for open-ended game agents. In *Forty-second International Conference on Machine Learning*, 2025.

A Game Introduction

Minecraft (MC) is an open-ended 3D sandbox game where players interact with a procedurally generated world composed of discrete blocks. From the perspective of agent research, MC naturally induces long-horizon sequential decision making: under **partial observability**, an agent must explore, gather resources, craft tools, and plan multi-step actions to accomplish goals. Its highly combinatorial state space and flexible objectives make Minecraft a widely used testbed for studying embodied control, planning, and tool-use behaviors in complex environments.

A.1 Basic Rules

A Minecraft world is composed of discrete blocks and populated with resources (e.g., wood, stone, ores), passive animals, and hostile mobs depending on the selected mode and difficulty. Many tasks follow a characteristic progression of **collecting** materials, **crafting** intermediate items, **upgrading** equipment, and **unlocking** access to more advanced resources or regions. Figure 6 provides representative in-game screenshots that illustrate these mechanics and the typical interaction context.



(a) (b)

(c) (d)

Figure 6: (a) Diverse block types form an explorable, harvestable, and modifiable world. (b) The crafting interface illustrates recipe-based item composition and tool progression. (c) Survival mode introduces continuous survival pressure from health, hunger, and environmental hazards. (d) The day–night cycle changes visibility and threats, making exploration and shelter building more strategic.

- **Block interaction**: Agents can mine blocks to obtain items and place blocks to construct structures or modify the terrain; tool choice affects mining speed and obtainable drops.
- **Resource acquisition and crafting**: Collected items can be combined via crafting recipes to produce tools, weapons, and utilities; the crafting graph induces clear multi-step dependencies.
- **Survival dynamics (optional)**: In Survival mode, health and hunger constrain behavior; damage from falls, lava, or hostile mobs introduces risk and requires reactive control.
- **Time and environment**: Day–night cycles and biomes change visibility and threats, making navigation and shelter building strategic.

B Compute Reporting

B.1 Experimental Configuration

All experiments are conducted on a server equipped with 8 NVIDIA A40 GPUs. The GPUs are primarily used for parallel Minecraft environment interaction, visual observation processing, and inference of the STEVE-1 low-level execution policy. The high-level planner, Monitor, Inducer, Curator, and Adaptor are executed under the unified protocol described in the main text, with the same evaluation-time LLM-call budget across all compared methods and ablation variants.

Unless otherwise stated, we use $\lambda_x = 0.4$, $\lambda_v = 0.4$, $\lambda_g = 0.2$, $\epsilon_x = \epsilon_v = 1.0$, $\epsilon_p = 0.10$, $\eta_{\text{fail}} = 0.5$, $h = 4$, retrieval top-$K = 16$, a per-query prompt budget $B_{\text{tok}} = 512$ tokens, and a maximum plan length of 12 subgoals. The feedback buffer $\mathcal{B}$ is reset at the start of each episode and therefore has no capacity parameter; B_{tok} bounds a single retrieval query, whereas the token counts reported in Table 8 are summed over all retrieval calls in an episode. Monitor, the Curator validation layer, and retrieval are deterministic functions and add no LLM call; the planner, Inducer, and Adaptor are the only LLM components.

To ensure a fair comparison, all methods share the same task set, episode horizon, task–seed split, success criterion, and evaluation metric. For methods that use STEVE-1 as the low-level execution policy, we further fix the observation/action interface, action primitives, structured state fields, retrieval top-K, memory-token budget, and knowledge retrieval budget.

The main computational cost comes from environment rollouts, low-level policy inference, knowledge retrieval, and high-level LLM calls. Since this work focuses on the accumulation, validation, and reuse of external behavioral knowledge rather than parameter updates of the language model or the low-level policy, we do not train the planner backbone or STEVE-1. This setup ensures that the reported performance differences mainly reflect differences in planning, feedback utilization, and local repair mechanisms, rather than differences in hardware resources or training compute.

B.2 Environment, Tasks, and Protocol Control

We evaluate MineEvolve on a 70-task evaluation suite selected from the Minecraft MCU tech-tree benchmark. Agents operate in a survival-style environment and perceive the world through first-person RGB observations together with structured state inputs. The structured state includes the agent's position, health, hunger, GUI status, inventory contents, and observable task progress. The action space covers navigation, camera control, mining, attacking, placing blocks, using items, switching inventory slots, and crafting. Each episode starts with an empty inventory, and all task-required resources must be obtained through interaction with the environment.

The selected 70 tasks are grouped according to the Minecraft tech-tree progression into seven categories: Wooden, Stone, Iron, Gold, Redstone, Diamond, and Armor. The early-stage groups mainly involve basic resource collection and simple crafting, while the later-stage groups require longer dependency chains, including tool upgrades, ore search, smelting, redstone-component construction, diamond-equipment crafting, and armor crafting. To report the overall success rate across all task groups, we separately analyze the Iron, Redstone, Diamond, and Armor groups, as these tasks more clearly expose missing prerequisites, local failure propagation, and the need for recovery planning.

We define the **low-level execution policy** as the model that maps observations and high-level subgoals to executable Minecraft actions. Our main comparison follows a system-level evaluation protocol: all methods are evaluated on the same task set, task–seed split, episode horizon, success criterion, and success-rate metric, while each method retains its native high-level planning and execution pipeline. DEPS, JARVIS-1, Optimus-1, and MineEvolve use STEVE-1 as the low-level execution policy under our interface. Optimus-2, in contrast, uses its native GOAP-based execution policy and is included as a system-level baseline with a different low-level execution design.

For STEVE-1-based methods, we further control the action primitives, observation/action interface, structured state fields, memory-token budget, retrieval top-K, and evaluation-time LLM-call budget. This ensures that the same-policy comparisons are not confounded by differences in the low-level controller, input representation, retrieval budget, or LLM-call budget. Typed execution feedback is constructed from the same environment states and execution-monitoring signals across all relevant

methods. However, only MineEvolve further abstracts this feedback into reusable skills and remedies, which are then used for knowledge-conditioned local repair during planning and execution.

C Algorithmic Details

The main text describes the core design of MineEvolve: the agent performs self-evolution through an external behavioral knowledge system rather than by updating language-model parameters. Specifically, MineEvolve converts each subgoal execution into typed execution feedback, induces skills and remedies from the feedback, validates and maintains the generated knowledge, and performs knowledge-conditioned local replanning when repeated failures or stagnation occur. This appendix provides full pseudocode for the procedure.

C.1 Overall MineEvolve Loop

Algorithm 1 shows the overall loop of MineEvolve. The system maintains an episode-level feedback buffer $\mathcal{B}$ and an external knowledge store $\mathcal{K}$ containing validated skills and remedies. After execution, Monitor–Inducer generates candidate knowledge from the process, and Curator–Adaptor validates, maintains, retrieves, and uses the knowledge to repair the unfinished plan when needed.

Algorithm 1 Overall MineEvolve Loop

Input: task goal g, initial state s_0, language-model planner π_θ, low-level executor $\mathcal{E}$, knowledge store $\mathcal{K}$
Output: task outcome and updated knowledge store $\mathcal{K}$
Initialize feedback buffer $\mathcal{B} \leftarrow \emptyset$
Set current state $s \leftarrow s_0$
Curator retrieves skills and remedies from $\mathcal{K}$ relevant to (g, s)
π_θ generates an initial subgoal plan $\mathbf{z}_{1:N}$ using the goal, state, and retrieved knowledge
for $i = 1$ to N **do**
    Executor $\mathcal{E}$ executes the current subgoal z_i and returns trace τ_i
    Monitor–Inducer generates candidate skills or remedies from τ_i and $\mathcal{B}$
    Curator validates, merges, filters, and updates $\mathcal{K}$ with candidate knowledge
    if repeated failure or stagnation is detected **then**
        Curator retrieves skills and remedies related to the current failure context
        Adaptor freezes the completed prefix and repairs the unfinished plan using retrieved knowledge and active remedies
    end if
    Update current state s
    if task goal g is completed **then**
        return success, $\mathcal{K}$
    end if
end for
return task outcome, $\mathcal{K}$

C.2 Monitor–Inducer: Feedback-Conditioned Skill and Remedy Generation

Monitor–Inducer corresponds to the feedback-conditioned generation process in the main text. Monitor first compresses a subgoal execution trace into typed execution feedback,

$$e_i = (z_i, y_i, \Delta s_i, \Delta v_i, f_i, p_i, \ell_i),$$

where z_i is the current subgoal, y_i is the success label, Δs_i and Δv_i are the state and inventory changes, f_i is the failure type, p_i is the execution progress, and ℓ_i is the stagnation flag. Inducer then generates a skill from successful feedback and a remedy from failed or stagnant segments.

Algorithm 2 Monitor–Inducer: Feedback-conditioned generation of skills and remedies.

Input: current subgoal z_i, execution trace τ_i, feedback buffer $\mathcal{B}$, recent window size h
Output: updated feedback buffer $\mathcal{B}$ and candidate knowledge set $\mathcal{C}$
$\mathcal{C} \leftarrow \emptyset$ ▷ Initialize candidate skills and remedies
Stage 1: Monitor extracts typed execution feedback
$(s_i^{\text{pre}}, s_i^{\text{post}}, o_{i,1:T}) \leftarrow \text{ParseTrace}(\tau_i)$ ▷ Parse pre-state, post-state, and observations
$\Delta s_i \leftarrow \text{StateDiff}(s_i^{\text{pre}}, s_i^{\text{post}})$ ▷ Compute environment-state changes
$\Delta v_i \leftarrow \text{InventoryDiff}(s_i^{\text{pre}}, s_i^{\text{post}})$ ▷ Compute inventory changes
$y_i \leftarrow \text{CheckSuccess}(z_i, s_i^{\text{post}})$ ▷ Check whether the subgoal succeeds
$f_i \leftarrow \text{DiagnoseFailure}(z_i, \tau_i, \Delta s_i, \Delta v_i)$ ▷ Identify failure type
$p_i \leftarrow \text{ProgressScore}(\tau_i, \Delta s_i, \Delta v_i)$ ▷ Estimate spatial, inventory, and goal progress
$\ell_i \leftarrow \text{DetectStagnation}(p_i, y_i)$ ▷ Detect low-progress or stagnant execution
$e_i \leftarrow (z_i, y_i, \Delta s_i, \Delta v_i, f_i, p_i, \ell_i)$ ▷ Build typed execution feedback
$\mathcal{B} \leftarrow \mathcal{B} \cup \{e_i\}$ ▷ Update episode-level feedback buffer
Stage 2: Inducer identifies feedback segments
$\mathcal{S}^+ \leftarrow \text{FindSuccessfulSegments}(\mathcal{B})$ ▷ Find consecutive and verifiable successful segments
$\mathcal{S}^- \leftarrow \text{FindFailureSegments}(\mathcal{B}, h)$ ▷ Find recent failed or stagnant segments
Stage 3: Inducer builds candidate behavioral knowledge
for all $\mathcal{B}^+ \in \mathcal{S}^+$ **do**
  if PassTargetCheck($\mathcal{B}^+$) = True **then**
    $k^+ \leftarrow \text{BuildSkill}(\mathcal{B}^+)$ ▷ Build skill with trigger, preconditions, steps, verification, and effects
    $E(k^+) \leftarrow \mathcal{B}^+$ ▷ Record supporting feedback
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{k^+\}$
  end if
end for
for all $\mathcal{B}^- \in \mathcal{S}^-$ **do**
  if NeedRemedy($\mathcal{B}^-$) = True **then**
    $k^- \leftarrow \text{BuildRemedy}(\mathcal{B}^-)$ ▷ Build remedy with trigger, failure, risk, repair, and scope
    $E(k^-) \leftarrow \mathcal{B}^-$ ▷ Record failed or stagnant supporting feedback
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{k^-\}$
  end if
end for
return $\mathcal{B}, \mathcal{C}$

Algorithm 2 gives the full Monitor–Inducer procedure. Monitor–Inducer turns raw interaction traces into structured knowledge candidates. Unlike storing complete trajectories or generating free-form reflections, each candidate skill or remedy must include a clear trigger, verification rule, or repair action, which provides a structured basis for Curator validation and retrieval.

Structured outputs are treated as untrusted rather than directly executable. Across 548 structured-generation calls in 400 construction episodes, 522 (95.26%) were directly parseable, 19 (3.47%) were recovered by a deterministic extractor, and 7 (1.28%) were unrecoverable: 4 Inducer outputs were discarded and 3 Adaptor repairs were skipped. None caused a crash or entered the knowledge base, and an episode is terminated after three consecutive subgoal failures.

C.3 Curator–Adaptor: Knowledge Maintenance and Local Replanning

Algorithm 3 gives the full Curator–Adaptor procedure. Curator–Adaptor prevents MineEvolve from simply placing all historical text into the prompt. Curator first filters out non-executable, overly generic, or conflicting candidates, and then retrieves the most relevant knowledge under a fixed prompt budget. Adaptor uses retrieved skills as positive guidance and remedies as repair knowledge for the unfinished plan.

Algorithm 3 Curator–Adaptor: Knowledge-guided repair of local failures.

Input: candidate knowledge set $\mathcal{C}$, knowledge store $\mathcal{K}$, task goal g, current state s_i, current subgoal index i, current plan $\mathbf{z}_{1:N}$, feedback buffer $\mathcal{B}$, planner π_θ, prompt budget B
Output: updated knowledge store $\mathcal{K}$ and repaired plan $\mathbf{z}_{1:N}$
Stage 1: Curator validates and maintains candidate knowledge
for all $k \in \mathcal{C}$ **do**
  if SchemaCheck(k) = False **then**
    continue ▷ Discard entries with incomplete fields
  end if
  if ContextMatch(k, s_i) = False **then**
    continue ▷ Discard entries not matchable to current state
  end if
  if ExecutableCheck(k) = False **then**
    continue ▷ Discard non-executable steps or repair actions
  end if
  if ConflictCheck($k, \mathcal{K}$) = True **then**
    continue ▷ Discard candidates conflicting with high-confidence knowledge
  end if
  if SpecificityCheck(k) = False **then**
    continue ▷ Discard overly generic experience summaries
  end if
  $\mathcal{K} \leftarrow$ MergeOrInsert($\mathcal{K}, k$) ▷ Merge similar entries or insert new knowledge
end for
Stage 2: Curator retrieves planning-relevant knowledge
$c_i \leftarrow (g, s_i, \mathbf{z}_{1:N}, \mathcal{B})$ ▷ Build current planning context
$\mathcal{K}_i^R \leftarrow$ Retrieve($\mathcal{K}, c_i, B$) ▷ Retrieve relevant knowledge under prompt budget B
$\mathcal{S}_i^R \leftarrow \{k \in \mathcal{K}_i^R \mid k.\text{type} = \text{skill}\}$ ▷ Retrieved reusable skills
$\mathcal{M}_i^R \leftarrow \{k \in \mathcal{K}_i^R \mid k.\text{type} = \text{remedy}\}$ ▷ Retrieved remedies
Stage 3: Adaptor performs knowledge-conditioned local replanning
if NeedRepair($\mathcal{B}$) = True **then**
  $k_{\text{temp}}^{\text{remedy}} \leftarrow$ BuildTemporaryRemedy($\mathcal{B}$) ▷ Build a temporary remedy from recent failure feedback
  $\mathcal{A}_i \leftarrow \mathcal{M}_i^R \cup \{k_{\text{temp}}^{\text{remedy}}\}$ ▷ Form active remedy set
  $\mathbf{z}_{1:i} \leftarrow$ FreezePrefix($\mathbf{z}_{1:N}, i$) ▷ Keep completed or still valid plan prefix
  $\mathbf{z}'_{i+1:N} \leftarrow \pi_\theta(g, s_i, \mathbf{z}_{1:i}, \mathcal{S}_i^R, \mathcal{A}_i)$ ▷ Replan the unfinished subgoals under skills and active remedies
  $\mathbf{z}_{1:N} \leftarrow [\mathbf{z}_{1:i}; \mathbf{z}'_{i+1:N}]$ ▷ Replace the original suffix with the repaired suffix
else
  Keep the current plan $\mathbf{z}_{1:N}$ unchanged ▷ Use retrieved knowledge only in later planning calls
end if
return $\mathcal{K}, \mathbf{z}_{1:N}$

For example, if recent feedback shows that the agent repeatedly moves near the same path without inventory changes, Curator maintains a corresponding remedy, and Adaptor asks the planner to avoid the repeated path and generate subgoals such as clearing blocking blocks or taking an alternative route. If the failure is caused by a missing tool or crafting table, Adaptor inserts the corresponding prerequisite into the unfinished plan. Thus, local repair in MineEvolve is a correction of the unfinished plan under both validated external knowledge and current failure feedback.

D Supplementary Experiments and Analysis

D.1 MCU Tech-Tree Task Suite and Metric Definitions

We evaluate MineEvolve on a 70-task subset selected from the Minecraft MCU tech-tree benchmark. The task suite is organized according to the Minecraft technology tree and contains seven task groups: Wooden, Stone, Iron, Gold, Redstone, Diamond, and Armor (Table 4). These groups cover resource collection, tool and weapon crafting, mining, trading, repairing, eating, redstone-component construction, and armor-related tasks. The suite forms a curriculum from short-horizon resource acquisition to long-horizon tasks with deeper prerequisite chains.

Agents operate in a survival-style Minecraft environment. Each episode provides first-person RGB observations and structured state inputs, including the agent's position, health, hunger, GUI status, inventory contents, and observable task progress. The action space covers navigation, camera control, mining, attacking, placing blocks, using items, switching inventory slots, inventory manipulation, and crafting. Unless otherwise specified, each episode starts with an empty inventory, and all task-required resources must be obtained through interaction with the environment.

Table 4: MCU tech-tree evaluation suite with seven task groups.

Group	Task Num.	Representative Task Types
Wooden	11	Wooden tools and weapons; wood, planks, sticks, and saplings gathering
Stone	10	Stone tools and weapons; stone and cobblestone mining; basic tool upgrades
Iron	16	Iron ore mining; smelting; iron tools and weapons; villager trading
Gold	7	Gold-related tools; golden foods; repairing golden equipment
Redstone	6	Redstone components; simple mechanisms; basic circuits and clocks
Diamond	7	Diamond mining; diamond tools; repairing and trading for high-tier items
Armor	13	Leather and iron armor; washing and repairing armor; helmet-related tasks

Task groups. We use Wooden, Stone, and Gold as the easy task groups, since they primarily involve resource acquisition, basic crafting, simple item-level progression, and lower-tier technology-tree objectives in this benchmark subset. We use Iron, Redstone, Diamond, and Armor as the hard task groups, since they involve longer prerequisite chains, more frequent missing-resource failures, tool dependencies, navigation bottlenecks, recovery requirements, and failure propagation across subgoals.

Success rate. For a method m, task t, and evaluation run r, let $s_{m,t,r} \in \{0, 1\}$ denote whether the task is successfully completed within the episode horizon. The task-level success rate is defined as

$$\mathrm{SR}_m(t) = 100 \times \frac{1}{R_t} \sum_{r=1}^{R_t} s_{m,t,r},$$

where R_t is the number of evaluation runs for task t.

For each task group g, let $\mathcal{T}_g$ denote the set of tasks in that group. The group-level success rate is

$$\mathrm{SR}_m(g) = \frac{1}{|\mathcal{T}_g|} \sum_{t \in \mathcal{T}_g} \mathrm{SR}_m(t).$$

Aggregated metrics. The main text reports three aggregate metrics: Easy Avg., Hard Avg., and Overall. All aggregate metrics are computed as task-count-weighted averages, so that each individual task contributes equally to the corresponding score. This avoids assigning the same weight to task groups with different numbers of tasks.

Let $n_g = |\mathcal{T}_g|$ denote the number of tasks in group g. For the 70-task MCU subset, the group sizes are:

$$n_{\mathrm{Wooden}} = 11, \quad n_{\mathrm{Stone}} = 10, \quad n_{\mathrm{Iron}} = 16, \quad n_{\mathrm{Gold}} = 7,$$
$$n_{\mathrm{Redstone}} = 6, \quad n_{\mathrm{Diamond}} = 7, \quad n_{\mathrm{Armor}} = 13.$$

Easy Avg. is the task-count-weighted average over the easy task groups, namely Wooden, Stone, and Gold:

$$\mathrm{EasyAvg}_m = \frac{11\,\mathrm{SR}_m(\mathrm{Wooden}) + 10\,\mathrm{SR}_m(\mathrm{Stone}) + 7\,\mathrm{SR}_m(\mathrm{Gold})}{28}.$$

Hard Avg. is the task-count-weighted average over the hard task groups, namely Iron, Redstone, Diamond, and Armor:

$$\mathrm{HardAvg}_m = \frac{16\,\mathrm{SR}_m(\mathrm{Iron}) + 6\,\mathrm{SR}_m(\mathrm{Redstone}) + 7\,\mathrm{SR}_m(\mathrm{Diamond}) + 13\,\mathrm{SR}_m(\mathrm{Armor})}{42}.$$

Overall is computed using the same task-count-weighted convention over all seven task groups, i.e., as the average task-level success rate over the full 70-task suite.

Protocol control. All compared methods are evaluated on the same task set, task–seed split, episode horizon, success criterion, and success-rate metric. For STEVE-1-based methods, we additionally fix the low-level execution policy, observation/action interface, action primitives, structured state fields, retrieval top-K, memory-token budget, and evaluation-time LLM-call budget. This ensures that performance differences primarily reflect differences in planning, execution-feedback utilization, knowledge maintenance, and local repair, rather than differences in task selection or evaluation protocol.

D.2 Full Task-Group Results

Table 5 provides the full success rates across all seven MCU task groups. This table complements the compact results in Table 1 by showing how the gains are distributed across different stages of the Minecraft tech tree. We additionally report the STEVE-1 low-level-only baseline, which highlights the difficulty of solving long-horizon dependency tasks without high-level planning and recovery.

Table 5: Full results on the MCU task suite. Values denote success rate SR (%). All system-level methods are evaluated under the same task set, evaluation metric, and interaction horizon. Wo., St., Ir., Go., Re., Di., and Ar. denote Wooden, Stone, Iron, Gold, Redstone, Diamond, and Armor, respectively. Overall is task-weighted over all 70 tasks.

Planner backbone	Method	Wo.	St.	Ir.	Go.	Re.	Di.	Ar.	Overall
Qwen3.5-Flash	DEPS	82.91	68.37	16.48	2.79	5.84	2.11	7.25	28.90
	JARVIS-1	92.37	88.14	34.76	**10.83**	23.41	8.39	14.92	41.75
	Optimus-1	98.30	**92.75**	45.24	8.07	28.17	8.21	14.80	45.83
	Optimus-2 (GOAP)	**98.60**	92.45	51.30	8.70	28.10	12.40	20.95	48.84
	MineEvolve	98.09	92.61	**52.43**	10.11	**30.77**	**14.58**	**23.47**	**50.09**
Qwen3.5-Plus	DEPS	84.05	69.34	17.39	3.47	8.94	2.43	4.36	29.25
	JARVIS-1	93.08	88.73	35.57	11.12	26.18	9.03	15.58	42.59
	Optimus-1	97.91	**93.60**	46.06	7.05	**32.03**	11.42	19.09	47.42
	Optimus-2 (GOAP)	**99.05**	93.20	53.20	9.20	28.70	13.20	21.60	49.75
	MineEvolve	98.71	93.48	**55.83**	**13.79**	31.24	**17.06**	**27.63**	**52.52**
GLM-4.7	DEPS	82.37	66.91	15.92	1.36	7.82	2.18	4.12	27.93
	JARVIS-1	91.79	86.87	33.94	**8.87**	21.82	8.08	14.19	40.79
	Optimus-1	96.63	91.40	44.13	7.79	23.06	7.95	18.04	45.23
	Optimus-2 (GOAP)	**97.90**	**91.65**	50.35	8.35	26.85	**12.05**	20.10	48.06
	MineEvolve	97.36	91.18	**50.55**	8.63	**29.08**	10.92	**22.07**	**48.43**
Gemini-3-Flash	DEPS	84.25	69.86	17.81	4.07	9.61	2.76	4.66	29.66
	JARVIS-1	92.99	88.58	35.21	9.79	22.96	8.92	15.52	42.04
	Optimus-1	98.70	91.76	45.92	6.85	24.48	13.06	18.98	46.73
	Optimus-2 (GOAP)	**99.10**	**93.55**	54.20	10.05	30.25	**13.80**	22.35	50.45
	MineEvolve	98.63	93.37	**55.28**	**13.52**	**33.73**	12.78	**27.18**	**52.04**
GPT-5.5	DEPS	86.42	72.38	20.31	6.16	11.84	3.68	6.07	31.69
	JARVIS-1	94.18	90.31	38.44	12.61	25.96	10.34	16.14	44.01
	Optimus-1	98.62	**94.82**	49.28	10.71	31.47	15.31	17.86	48.92
	Optimus-2 (GOAP)	**99.22**	94.60	56.15	11.35	32.10	14.70	23.50	51.66
	MineEvolve	99.16	94.77	**60.12**	**16.70**	**36.47**	**18.91**	**28.98**	**54.93**
STEVE-1	low-level only	25.61	18.72	2.47	0.00	0.00	0.00	0.00	7.26

D.3 Positioning with Recent Systems

Table 6 summarizes the relationship between MineEvolve and recent Minecraft agents from the perspectives of experimental protocol and mechanism design. The purpose of this table is to clarify the research focus of different systems, rather than to provide an absolute ranking under mismatched evaluation protocols.

Table 6: Mechanistic positioning of recent Minecraft agents.

Method	Low-level policy	Use of failure feedback	Typed feedback	Continual knowledge update	Same-policy comparison
DEPS	STEVE-1	Failure self-explanation; iterative plan revision	No	No	Yes
JARVIS-1	STEVE-1	Multimodal gameplay-memory retrieval	No	No	Yes
Optimus-1	STEVE-1	Retrieved as part of the experience pool	No	Partial	Yes
Optimus-2	GOAP	Not a core mechanism	No	No	No
XENON	—	Binary failure feedback; dependency/action correction	No	Yes	No
MineEvolve	STEVE-1	Remedy generation	Yes	Yes	Yes

D.4 Knowledge Accumulation and Runtime Overhead

Table 7 reports the group-wise results of the knowledge accumulation experiment on the four difficult task groups. At each checkpoint, we freeze the external knowledge base before evaluation. Hard Avg. denotes the average success rate over Iron, Redstone, Diamond, and Armor.

Table 7: Group-wise results for the knowledge accumulation curves. Values denote SR (%) on held-out hard tasks. Hard Avg. is the task-count-weighted average over Iron, Redstone, Diamond, and Armor, using group sizes 16, 6, 7, and 13, respectively. At each checkpoint, the external knowledge base is frozen before evaluation.

Method	Task group	0 eps.	50 eps.	100 eps.	200 eps.	400 eps.
Static Store	Iron	39.8	40.8	41.9	42.9	43.6
	Redstone	24.2	24.9	25.6	26.2	26.5
	Diamond	9.0	9.5	9.9	10.4	10.9
	Armor	16.2	17.2	18.2	19.3	19.4
	Hard Avg.	25.13	26.01	26.90	27.79	28.22
Text Reflection	Iron	39.8	42.3	44.1	44.9	45.4
	Redstone	24.2	25.6	26.8	27.4	27.9
	Diamond	9.0	10.2	11.0	11.5	12.0
	Armor	16.2	18.7	19.3	20.2	20.3
	Hard Avg.	25.13	27.26	28.44	29.19	29.56
MineEvolve	Iron	39.8	45.4	48.2	51.3	53.4
	Redstone	24.2	27.2	28.6	30.0	31.0
	Diamond	9.0	11.2	12.7	14.4	15.8
	Armor	16.2	21.4	22.5	23.5	24.6
	Hard Avg.	25.13	**29.67**	**31.53**	**33.50**	**35.02**

Since MineEvolve relies on an external knowledge base, it is important to examine whether knowledge growth introduces excessive retrieval or prompting overhead. Table 8 reports the number of skills, the number of remedies, the average number of retrieved tokens per episode, retrieval latency, and the number of evaluation-time LLM calls at different accumulation checkpoints. This analysis verifies whether the budget control of the Curator prevents unbounded context growth.

Retrieved tokens are summed over all retrieval calls in an episode rather than counted per prompt; each individual query is capped at $B_{\text{tok}} = 512$ tokens and returns at most $K = 16$ entries, whereas the knowledge base itself is not bounded.

To make the full performance–cost tradeoff explicit, Table 9 varies the per-query budget on the same frozen 400-episode checkpoint used in Table 7, with the Qwen3.5-Plus planner on the hard task groups.

Table 8: Knowledge-base size and runtime overhead. Values are averaged over held-out hard-task episodes.

Checkpoint	#Skills	#Remedies	Retrieved tokens/episode	Retrieval latency	Eval. LLM calls
0 eps.	0	0	0	0.00 s	7.82
50 eps.	31	44	628	0.10 s	7.96
100 eps.	56	78	846	0.14 s	8.07
200 eps.	94	128	1087	0.19 s	8.18
400 eps.	143	191	1216	0.25 s	8.31

Table 9: Sensitivity to the per-query retrieval budget. All rows use the same frozen 400-episode knowledge base and the Qwen3.5-Plus planner on the hard task groups.

Per-query budget	Retrieved tokens/episode	Retrieval latency	Hard SR
128	408	0.14 s	31.47
256	731	0.19 s	34.00
512	1,216	0.25 s	35.02
1,024	1,908	0.34 s	35.10

Performance improves by 3.55 points from 128 to 512 tokens, whereas doubling the budget from 512 to 1,024 changes Hard SR by only 0.08 while increasing retrieved context by 56.9%, so 512 tokens is a favorable accuracy–cost tradeoff rather than an arbitrary truncation.

Table 10 separates construction-time from evaluation-time cost. Monitor and Curator are deterministic and make no LLM call, while Inducer and Adaptor account for 97.1% of the total; knowledge construction adds 3,420 input and 286 output tokens per episode. Frozen-knowledge evaluation uses 8.31 versus 7.82 LLM calls per episode (+0.49, 6.3%).

Table 10: Component-wise overhead over 400 knowledge-construction episodes.

Component	Invocation frequency	Added latency
Monitor	deterministic, per episode	0.004 s/episode
Curator validation, conflict checking, merging	candidate-triggered	0.081 s/episode
Inducer	384 calls (0.96/episode)	1.84 s/episode
Adaptor	164 calls (0.41/episode)	0.97 s/episode
Total non-environment construction/repair	—	2.90 s/episode

D.5 Curriculum-style Knowledge Construction on Diamond Tasks

To further analyze how the source of external behavioral knowledge affects difficult tasks, we conduct a curriculum-style knowledge construction experiment on the Diamond task group. This experiment differs from the unified MCU evaluation protocol used in the main experiments; instead, it is designed to isolate how different knowledge initialization and update strategies influence continual learning efficiency on Diamond tasks. All settings share the same planner backbone, STEVE-1 low-level execution policy, environment interface, interaction horizon, retrieval budget, and evaluation-time LLM-call budget. The only difference lies in how the external knowledge bases K_{skill} and K_{remedy} are initialized and updated.

Table 11 summarizes the four knowledge construction strategies. Cold Start keeps the knowledge base empty throughout and serves as a lower bound without cross-episode behavioral knowledge. Diamond-only Self-learning generates skills and remedies only from Diamond tasks. Curriculum Pretrain→Freeze first builds the knowledge base on lower-tier tasks from Wooden to Redstone, and then freezes this knowledge when transferring to Diamond tasks. Mixed Sampling (1:1) mixes lower-tier tasks and Diamond tasks at a 1:1 ratio and maintains a unified $K_{\text{skill}}/K_{\text{remedy}}$ online.

Table 11: Curriculum-style knowledge construction strategies for Diamond tasks. All settings share the same planner, low-level execution policy, interaction horizon, and retrieval budget.

Strategy	Knowledge initialization	Training-stage update	Purpose
Cold Start (Empty KB)	Empty $K_{\text{skill}}/K_{\text{remedy}}$	No cross-episode knowledge writing	Estimate the lower bound without external behavioral knowledge
Diamond-only Self-learning	Empty $K_{\text{skill}}/K_{\text{remedy}}$	Write Diamond-only experience	Test whether self-learning only on the target hard tasks is sufficient
Curriculum Pretrain→Freeze	Knowledge pretrained on lower-tier tasks	Freeze lower-tier knowledge during the Diamond stage	Test whether lower-tier knowledge transfers to Diamond tasks
Mixed Sampling (1:1)	Unified online knowledge base	Jointly update with lower-tier and Diamond experience	Test whether mixed updates yield stronger continual improvement

The training budget is controlled by the total number of episodes, and evaluation is performed at five checkpoints: 20%, 40%, 60%, 80%, and 100%. For Diamond-only Self-learning and Mixed Sampling (1:1), each checkpoint corresponds to the completed online training progress under the respective strategy. For Curriculum Pretrain→Freeze, each checkpoint corresponds to the construction progress of the lower-tier pretrained knowledge base. At each checkpoint, we freeze the current knowledge base before evaluating on Diamond tasks. Each evaluation is repeated across multiple world seeds, and the success rate is averaged over the seven Diamond tasks. Table 12 reports the success rates on Diamond tasks. This experiment is intended to analyze the effect of knowledge construction pathways on Diamond tasks and does not replace the unified MCU evaluation results reported in the main table. As shown in Table 12, different knowledge construction strategies lead to clear performance

Table 12: Success rates of different curriculum-style knowledge construction strategies on Diamond tasks. Values denote SR (%), averaged over seven Diamond tasks and repeated across multiple world seeds.

Strategy	SR@20%	SR@40%	SR@60%	SR@80%	SR@100%
Cold Start (Empty KB)	2.6	2.8	2.7	2.8	2.9
Diamond-only Self-learning	3.0	6.1	9.6	12.2	13.9
Curriculum Pretrain→Freeze	5.7	8.9	11.8	14.3	16.4
Mixed Sampling (1:1)	**6.2**	**9.8**	**13.1**	**15.8**	**17.6**

differences on Diamond tasks. Cold Start (Empty KB) remains below 3% across all checkpoints, with the success rate only changing slightly from 2.6% to 2.9%. This suggests that, without cross-episode skills and remedies, the agent struggles to achieve sustained improvement on long-dependency tasks such as Diamond. Diamond-only Self-learning improves steadily as training progresses, increasing from 3.0% to 13.9%. This indicates that task-specific knowledge can be gradually induced from Diamond trajectories alone. However, its low early-stage success rate also suggests that accumulating experience from scratch on high-tier tasks is sample-inefficient.

In contrast, Curriculum Pretrain→Freeze consistently outperforms Diamond-only Self-learning at all checkpoints and reaches a final success rate of 16.4%. This result shows that knowledge induced from lower-tier tasks, including tool crafting, resource collection, smelting, crafting prerequisites, and common failure-recovery patterns, can transfer to Diamond tasks. Although this strategy does not continue updating the lower-tier knowledge base during the Diamond stage, its performance on Diamond tasks improves as the lower-tier pretrained knowledge base becomes more complete across checkpoints.

Mixed Sampling (1:1) achieves the highest success rate at all checkpoints, improving from 6.2% to 17.6%. Compared with Curriculum Pretrain→Freeze, Mixed Sampling not only preserves transferable foundational knowledge from lower-tier tasks, but also continuously incorporates Diamond-specific skills and remedies during training. As a result, it maintains a stable advantage in the middle and later stages, suggesting that for high-tier tasks, freezing lower-tier knowledge provides a strong initialization, while continually mixing lower-tier and target-task experience is more effective for capturing high-tier-specific failure modes and repair actions.

To more directly compare the overall learning efficiency of different strategies, Table 13 summarizes the average success rate over the five checkpoints and the final success rate. Mixed Sampling (1:1) achieves the highest checkpoint-averaged success rate, indicating that its advantage is not limited to the final stage but remains stable throughout training. Curriculum Pretrain→Freeze ranks second in checkpoint average, further validating the effectiveness of lower-tier knowledge transfer. Diamond-only Self-learning eventually achieves a substantial improvement, but its overall learning efficiency is lower. Cold Start remains close to the performance floor throughout.

Table 13: Summary of gains from curriculum-style knowledge construction on Diamond tasks. Checkpoint Avg. denotes the average success rate over the five checkpoints. Final gain is computed relative to Diamond-only Self-learning.

Strategy	Checkpoint Avg.	Final SR	Final gain vs. Diamond-only
Cold Start (Empty KB)	2.76	2.9	-11.0
Diamond-only Self-learning	8.96	13.9	0.0
Curriculum Pretrain→Freeze	11.42	16.4	+2.5
Mixed Sampling (1:1)	**12.50**	**17.6**	**+3.7**

Overall, this experiment shows that performance on Diamond tasks depends not only on the number of training episodes, but also on the pathway through which the knowledge base is constructed. External behavioral knowledge acquired from lower-tier tasks substantially improves cold-start performance on high-tier tasks, while continual mixed updates further improve the final success rate. This finding is consistent with the design goal of MineEvolve: the agent's capability growth does not come from updating the parameters of the language model, but from continually accumulating, validating, and reusing external skills and remedies.

D.6 Qualitative Case Studies

In this section, we present two qualitative case studies to illustrate how MineEvolve transforms successful and failed execution experiences into external behavioral knowledge that can affect subsequent planning. The first case demonstrates failure-conditioned local replanning: when the agent becomes stuck near a terrain obstacle, the system converts the failure into a remedy that can be immediately applied within the current episode, and the Adaptor repairs the remaining plan accordingly. The second case illustrates cross-episode knowledge accumulation: a failed crafting attempt is distilled into a reusable remedy, while a subsequent successful execution is stored in the external knowledge base as a skill.

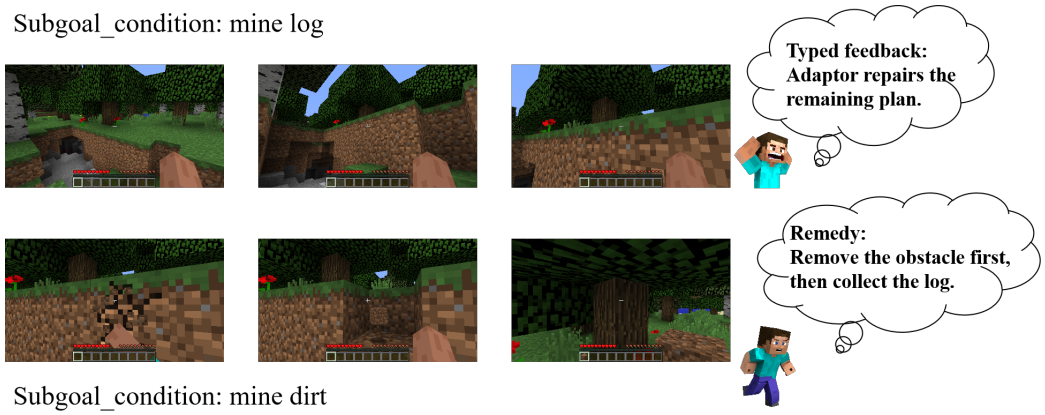


Figure 7: Failure-conditioned local replanning. When the agent repeatedly moves near a terrain obstacle without making effective progress and without any inventory change, the Monitor identifies the behavior as a low-progress navigation failure. The resulting remedy prompts the Adaptor to repair the remaining plan by inserting a corrective subgoal: first break the dirt block blocking the path, and then continue collecting the target wood.

Figure 7 illustrates how typed execution feedback supports local plan repair. The agent initially attempts to approach the target wood, but repeatedly moves near a terrain boundary with only limited change in position, while the inventory shows no increase in the target resource. The Monitor compresses this execution trace into typed feedback, including the failed subgoal, low-progress signals, no inventory change, and a navigation-related failure type. Because the failure is not a one-off mistake but a repeated stagnation pattern, the Inducer further generates an active remedy that specifies

both triggering conditions and a repair action: instead of repeatedly following the same approach path, the agent should first remove the blocking terrain or choose an alternative route.

The Curator then retrieves or maintains this remedy according to the current spatial context, and the Adaptor preserves the valid plan prefix while repairing only the unfinished downstream subgoals. As a result, the original plan no longer continues to approach the wood directly, but is revised to first break the dirt block blocking the path and then resume the subgoal of collecting the target wood. This case highlights that a remedy in MineEvolve is not merely a natural-language explanation of failure, but an executable constraint with explicit triggers and repair actions that can directly modify the remaining plan within the current episode.

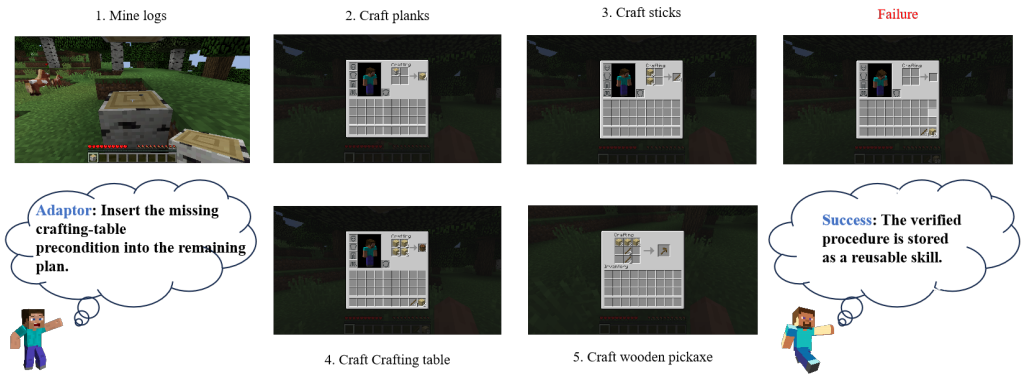


Figure 8: Feedback-conditioned generation of skills and remedies. A failed crafting attempt is converted into a remedy, indicating that recipes requiring a larger crafting grid must first obtain and place a crafting table. Once the corrected execution succeeds, the successful process is further stored as a reusable skill.

Figure 8 shows how MineEvolve accumulates external behavioral knowledge across episodes. The agent initially attempts to craft the target tool directly in the inventory crafting interface, but the recipe requires the larger crafting grid provided by a crafting table, causing the execution to fail. The Monitor records this failure through fields such as GUI state, inventory changes, and the absence of the target item, and summarizes it as typed execution feedback. Based on this feedback, the Inducer generates a remedy: when a crafting failure is caused by the missing prerequisite of a crafting table, the agent should first craft and place a crafting table before retrying the recipe.

Before writing the candidate remedy into the knowledge base, the Curator validates it by checking whether the triggering conditions can be matched by the current state fields, whether the repair action is executable by the low-level policy, and whether the knowledge is sufficiently specific and does not conflict with existing high-confidence knowledge. Once validated, the remedy is stored in the external knowledge base and can be retrieved in future similar tasks. Later, when the agent again plans a crafting task that requires a crafting table, the Adaptor inserts the prerequisite step of "craft and place a crafting table" into the remaining plan based on the retrieved remedy. If the corrected execution succeeds, the Inducer further abstracts this successful process into a skill, recording its trigger conditions, prerequisites, steps, execution effects, and verification rules. This case illustrates that MineEvolve leverages both failure and success: failures generate remedies to avoid repeated mistakes, while successful executions generate skills to reuse validated procedural knowledge.

E Prompt Design and Tool-Calling Details

This section provides the prompt templates used by MineEvolve. The prompts are aligned with the four-stage procedure in the main method: *Monitor* extracts typed execution feedback, *Inducer* converts feedback into candidate skills and remedies, *Curator* validates and retrieves relevant knowledge, and *Adaptor* repairs only the unfinished part of the plan under retrieved skills and active remedies. All prompts use a chat-style interface with explicit `system` and `user` roles. To ensure downstream parsing and validation, the LLM components are required to output strictly structured JSON objects; the Curator template below documents an optional LLM-assisted variant that is not used in the reported experiments.

Planner: system prompt and JSON schema. The planner generates a short sequence of high-level subgoals. Each subgoal must be executable by the low-level executor and must include explicit completion checks. The planner is also instructed to use retrieved skills as positive procedural guidance and remedies as constraints or repair knowledge.

Planner system prompt.

```
You are the high-level planner of MineEvolve, a Minecraft embodied agent.
Generate a short executable plan for the current task. Use retrieved skills as
positive guidance and active remedies as constraints for avoiding repeated
failures.
Output JSON only. Use the following schema:
{
"plan_id": "p_xxxx",
"subgoals": [
{
"subgoal_id": "sg_001",
"condition": "short executable action",
"task_kind": "mine|craft|smelt|use|combat|move|wait",
"executor_hint": "stevei|mcu_craft|mcu_smelt|wait",
"mode": "move|stay",
"timeout_s": 60,
"checks": [{"type":"inv_ge","item":"oak_log","n":1}],
"rationale": "one short reason"
}
],
"global_constraints": []
}
Planning rules:
1) Keep only necessary prerequisites for the target task;
2) Reuse retrieved skills when their preconditions match the current state;
3) Obey active remedies and avoid repeating known failure patterns;
4) Every subgoal must have a verifiable completion check;
5) Do not output explanations outside JSON.
```

Planner: user prompt with retrieved knowledge. The user prompt injects the current task, environment state, current plan prefix when available, retrieved skills, and active remedies. Unlike a generic memory capsule, the injected knowledge is separated into skills and remedies maintained by the Curator.

Planner user prompt.

```
Task goal: craft an iron pickaxe
Current state:
inventory={"iron_ingot":3,"stick":2}, coords=[100,64,200], HP=20, hunger=20
Current plan prefix:
completed=[]
Retrieved skills from Curator:
- id: skill_craft_tool_at_table
trigger: target requires 3x3 crafting grid
preconditions: ["have required ingredients", "crafting table available"]
steps: ["place crafting table", "open crafting table", "craft target item"]
verification: inventory contains target item
Active remedies from Curator/Inducer:
- id: remedy_missing_crafting_table
trigger: crafting fails because no 3x3 grid is available
repair: insert obtain-or-place-crafting-table before retrying recipe
scope: unfinished plan suffix
Generate the shortest executable plan that reaches the task goal.
```

Inducer: skill generation from successful feedback. When Monitor produces a sequence of successful and verifiable feedback records, Inducer asks the LLM to abstract them into a reusable skill. The output follows the unified knowledge-entry format used by MineEvolve.

Skill induction prompt.

```
System:
You are the Inducer module of MineEvolve. Convert successful typed execution
feedback into one reusable skill. The skill must be specific, executable, and
verifiable.
Output JSON only.
User:
Task goal: craft sticks
Successful feedback segment:
- e_001: z="collect oak log", y=1, delta_inv={"oak_log":4}, f="NONE", p=1.0,
stagnant=false
- e_002: z="craft planks", y=1, delta_inv={"oak_planks":16}, f="NONE", p=1.0,
stagnant=false
- e_003: z="craft sticks", y=1, delta_inv={"stick":4}, f="NONE", p=1.0,
stagnant=false
Output schema:
{
"type": "skill",
"id": "skill_xxxx",
"trigger_context": ["..."],
"preconditions": ["..."],
"steps": ["..."],
"verification": {"type":"inv_ge","item":"stick","n":4},
"observed_effects": ["..."],
"supporting_feedback": ["e_001","e_002","e_003"],
"confidence": 0.8
}
```

Inducer: remedy generation from failed or stagnant feedback. When Monitor detects repeated failures or low-progress execution, Inducer generates a remedy rather than a free-form reflection. A remedy must specify when it applies, what risk pattern it captures, and what repair action should be inserted into the unfinished plan.

Remedy induction prompt.

```
System:
You are the Inducer module of MineEvolve.  Convert failed or stagnant typed
execution
feedback into one executable remedy.  The remedy must include a trigger, failure
type,
risk pattern, repair action, and repair scope.  Do not output generic advice such
as
"try again" or "be careful".  Output JSON only.
User:
Current subgoal:  collect oak log
Recent feedback window:
- e_121:  y=0, delta_inv={}, f="NAV_STUCK", p=0.05, stagnant=true,
coords=[118,64,208], note="repeated movement near same obstacle"
- e_122:  y=0, delta_inv={}, f="NAV_STUCK", p=0.04, stagnant=true,
coords=[119,64,208], note="no inventory progress"
Output schema:
{
"type":  "remedy",
"id":  "remedy_xxxx",
"trigger_context":  ["navigation failure", "low progress", "no inventory gain"],
"failure_type":  "NAV_STUCK",
"risk_pattern":  "agent repeats movement near obstacle without collecting target
item",
"repair_action":  ["clear blocking block or choose alternative route",
"then retry collecting target resource"],
"scope":  "unfinished_plan_suffix",
"applicability":  ["same or similar navigation context", "target item not
increased"],
"supporting_feedback":  ["e_121","e_122"],
"confidence":  0.8
}
```

Curator: validation and retrieval of knowledge entries. Curator prevents vague, conflicting, or non-executable knowledge from entering the external knowledge store. It also retrieves a compact set of relevant skills and remedies under the prompt budget. In the implementation used for all reported experiments, schema checking, context matching, executability checking, specificity checking, conflict checking, merging, and retrieval are deterministic functions, and the Curator therefore adds no LLM call; the template below is the LLM-assisted configuration of our implementation and is documented for completeness.

Curator prompt (optional LLM-assisted variant, not used in the reported experiments).

```
System:
You are the Curator module of MineEvolve.  Validate candidate skill/remedy
entries
and select the most relevant entries for the current planning context.  Reject
entries
that are incomplete, not matchable to state fields, non-executable, overly
generic,
or conflicting with high-confidence knowledge.  Output JSON only.
User:
Current task goal:  craft iron pickaxe
Current state:  inventory={"iron_ingot":3,"stick":2}, coords=[100,64,200]
Recent failure types:  []
Prompt budget:  512 tokens
Candidate knowledge entries:  [ ...  ]
Existing knowledge store summary:  [ ...  ]
Output schema:
{
"accepted":  ["skill_id_1", "remedy_id_1"],
"rejected":  [
{"id":"...", "reason":"schema|match|exec|specificity|conflict"}
],
"retrieved_skills":  ["skill_id_1"],
"retrieved_remedies":  ["remedy_id_1"],
"budget_used":  0
}
```

Adaptor: knowledge-guided local plan repair. Adaptor is invoked when repeated failures or stagnation occur. It preserves the completed or still-valid plan prefix and regenerates only the unfinished suffix under retrieved skills and active remedies.

Adaptor local repair prompt.

```
System:
You are the Adaptor module of MineEvolve.  Repair only the unfinished suffix of
the current plan.  Preserve the completed or still-valid prefix.  Use retrieved
skills
as positive guidance and active remedies as repair constraints.  Output JSON
only.
User:
Task goal:  collect oak log
Current state:  inventory={}, coords=[118,64,208]
Current plan:
- sg_001:  move to oak tree [failed]
- sg_002:  mine oak log [unfinished]
Frozen prefix:  []
Recent typed feedback:
- e_121:  y=0, f="NAV_STUCK", p=0.05, stagnant=true, delta_inv={}
Retrieved skills:  []
Active remedies:
- remedy_nav_obstacle:  if navigation is stagnant with no inventory gain,
clear blocking blocks or choose an alternative route before retrying.
Output schema:
{
"plan_id":  "p_repair_xxxx",
"kept_prefix":  [],
"repaired_suffix":  [
{
"subgoal_id":  "sg_001r",
"condition":  "clear blocking dirt",
"task_kind":  "mine",
"executor_hint":  "stevei",
"mode":  "stay",
"timeout_s":  45,
"checks":  [{"type":"path_clear"}],
"repair_source":  "remedy_nav_obstacle"
}
],
"active_remedies_used":  ["remedy_nav_obstacle"]
}
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer:[Yes]

   Justification: The main claims in the abstract and introduction are supported by the proposed MineEvolve framework, as shown in Abstract and Introduction 1.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss limitations in the Conclusion and Limitations section 5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include theoretical results. All stated relations are definitions and empirical evaluation criteria rather than theorems, so no assumptions or proofs are required.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer:[Yes]

   Justification: We provide the backbone model, data sources, hyperparameters, and settings details in B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Our code is available at https://github.com/xzw-ustc/MC-MineEvolve.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer:[Yes]

   Justification: In experiment section.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [N/A]

   Justification: [N/A]

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix B reports the compute setup: all experiments use a server with 8 NVIDIA A40 GPUs, primarily for parallel Minecraft environment interaction, visual observation processing, and inference of the STEVE-1 low-level execution policy, together with the evaluation-time LLM-call budget shared across all methods.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: Conform.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer:[Yes]

    Justification: We discuss broader impact in the Ethics and broader impact paragraph of the Conclusion section 5: MineEvolve is a planning framework for a sandbox game and is not tied to a deployed application, its positive impact is that improved long-horizon recovery reduces the human supervision embodied agents require, and its potential negative impact is that the same recovery capability could transfer to settings where autonomous failure recovery has safety implications. We release only planning code and knowledge schemas, which contain no model weights.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [N/A].

    Justification: The paper does not use existing assets.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper introduces new research artifacts, including the MineEvolve framework implementation, experiment configurations, prompts, evaluation protocol, and derived external behavior knowledge such as skills and remedies generated from Minecraft simulation trajectories. These assets are documented in the paper and supplementary material, including the task suite, feedback fields, knowledge representation, ablation settings, and evaluation protocol. The assets are generated in a sandboxed Minecraft environment and do not contain human-subject data, personally identifiable information, or Internet-scraped personal content. Any released code, prompts, configurations, and synthetic or simulation-derived logs will be anonymized for submission and accompanied by documentation describing their intended use, limitations, and reproduction instructions.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing, user studies, behavioral experiments with human participants, or any paid human data collection or annotation. All experiments are conducted in a simulated Minecraft environment using benchmark tasks, agent-environment interaction logs, and automatically generated evaluation trajectories. Therefore, there are no participant instructions, screenshots, or compensation details to report.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing, user studies, behavioral experiments with human participants, or collection of data from human subjects. All experiments are conducted in a simulated Minecraft environment using benchmark tasks, agent-environment interaction logs, and automatically generated evaluation trajectories. Therefore, no study participants are involved and IRB approval or equivalent human-subjects review is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification:The paper uses LLMs as core components of the proposed system and describes their roles in the system architecture. Specifically, the paper explains how LLM-based modules are used for high-level planning, knowledge induction, and knowledge-conditioned plan repair, and how their outputs are incorporated into the agent pipeline. Knowledge curation (validation, merging, conflict filtering, and retrieval) is implemented as deterministic functions and introduces no additional LLM call.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.