OPENKAGE: OPEN-ENDED LIFELONG LEARNING IN THE OPEN WORLD VIA KNOWLEDGE-SHARING AGENT-GROUP EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Open-ended lifelong learning requires agents to accumulate and reuse knowledge while autonomously selecting goals. Existing research often pursues lifelong learning under predefined tasks, curricula, or objectives, and largely entrusts self-evolution to a single agent. We propose OpenKAGE, a training-free framework that realizes this vision through group-level knowledge sharing in open-world environments: (1) each agent maintains a local knowledge graph with two layers—a private memory layer for situated experience and a shareable experience layer holding transferable knowledge as textual records and executable skills; (2) a semantic aggregator periodically consolidates the experience layers into a shared graph, which agents access through intent-aware subgraph retrieval, while memory-layer records remain local. In 30-game-day Minecraft evaluations with two backbone models, OpenKAGE achieves obsidian acquisition and Nether entry, milestones not reached by a matched planner-only baseline, while reducing mean deaths. A cross-group, cross-backbone transfer experiment further suggests that accumulated knowledge can benefit newly initialized agents. Together, these results demonstrate sustained autonomous learning and effective knowledge reuse within the evaluated horizon, supporting OpenKAGE as a practical framework for open-ended lifelong learning.

1 INTRODUCTION

We study *open-ended lifelong learning* as a continuing process in which agents autonomously generate goals, accumulate knowledge from interaction, and reuse that knowledge to acquire new capabilities. Unlike learning toward a fixed terminal task, this process is not organized around a predefined endpoint. A suggestive image comes from *The Wild Robot*: cast ashore on an unfamiliar island with only her built-in directives, Roz learns to coexist with the wildlife and, by raising an orphaned gosling, comes to belong to the island. Future autonomous agents may likewise enter physical or digital worlds carrying only system-level directives, and must confront unfamiliar phenomena, discover new things to do, and devise new strategies without human intervention. Toward this vision, the field has made substantial progress: lifelong-learning methods address how acquired knowledge is retained and reused across experience streams (Parisi et al., 2019); reinforcement and imitation learning have enabled long-horizon control from primitive actions (Baker et al., 2022; Hafner et al., 2023); and LLM-based agents now plan, remember, and act in complex open worlds (Zhu et al., 2023; Yu & Lu, 2025; Tang et al., 2024), accumulating knowledge in many forms: internet-scale grounding (MineDojo), reusable code skills (Voyager), structured memory (JARVIS-1, Optimus-1), episodic traces (MrSteve), knowledge-graph world models (AriGraph), and failure lessons written back into the agent itself (Reflexion, PEAM) (Fan et al., 2022; Wang et al., 2024; 2025; Li et al., 2024; Park et al., 2025; Anokhin et al., 2025; Shinn et al., 2023; Guo et al., 2026). Yet these advances largely pursue lifelong learning under predefined tasks, curricula, or objectives rather than letting agents autonomously decide what to discover and learn, and they largely entrust self-evolution to a *single* agent. Such an agent's bounded history and perspective can lock learning into strategies shaped by a limited range of experiences, and its acquired knowledge does not circulate among agents. The result is isolated agents (Figure 1).

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

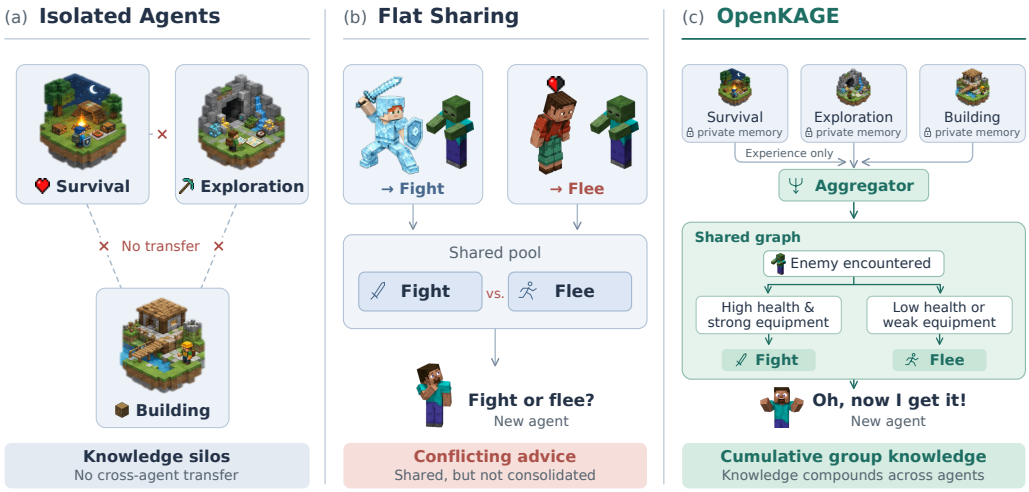

Figure 1: Knowledge sharing through consolidation rather than pooling. (a) Isolated agents acquire complementary experience but cannot transfer it. (b) Flat sharing pools advice without its conditions, leaving conflicting recommendations (fight versus flee). (c) OpenKAGE keeps situated memory local and shares experience with its applicability conditions attached.

A natural response is to *share* experience across agents, and prior work does so at three levels. At the level of *messages*, multi-agent LLM frameworks coordinate through role division and communication (Li et al., 2023; Hong et al., 2024), and generative agents (Park et al., 2023) show that individual memory streams can give rise to shared social conventions through interaction. At the level of *parameters*, federated continual learning fuses what different clients have learned into a global model while keeping their raw data local (Yang et al., 2024). Most recently, at the level of *raw traces*, Group-Evolving Agents (GEA) (Weng et al., 2026) pools the members' evolutionary traces, namely code patches, task logs, and tool-use histories, into a shared set that every offspring agent's reflector reads. Each level demonstrates that sharing helps; yet in all three, what crosses the boundary between agents is not consolidated as explicit, individually inspectable knowledge: message streams and trace pools preserve raw records without reconciling them, and parameter-level fusion does not directly expose knowledge as symbolic records that remain interpretable, retrievable, and debuggable. Sharing raw traces raises a subtler problem: it conflates an agent's situated, private memory with its decontextualized, transferable experience (Figure 1).

OpenKAGE draws on complementary learning systems theory, which holds that new experience is first registered quickly and with reduced interference, then slowly consolidated into structured knowledge (Kumaran et al., 2016), and models world knowledge as a **two-layer graph**. A *memory layer* holds each agent's private, situated experiences and never leaves that agent; an *experience layer* holds decontextualized, transferable textual knowledge records with executable skills attached. Only experience-layer records are provided to the aggregator, while situated memory records remain local to each agent: an agent contributes what it learned, not where it was. Building on this, OpenKAGE constructs **knowledge-sharing**: a semantic aggregator consolidates the experience layers into a shared graph. It merges synonyms, records cross-agent corroboration, and reconciles contradictions, turning dispersed individual experience into a single, cumulatively maintained body of group knowledge. To use this graph, OpenKAGE proposes *intent-aware subgraph retrieval*, which recalls from the graph an intent-relevant subgraph matched to the agent's current goal.

We run OpenKAGE in Minecraft. Each agent uses a planner and a reflector for in-day ReAct-style acting and end-of-day reflection (Yao et al., 2023; Shinn et al., 2023), while a group-level aggregator consolidates experience-layer updates into the shared graph every K days. The loop is act, reflect, aggregate, and retrieve. It is training-free and self-driven: the backbone parameters are never updated, there is no human feedback, no human-authored curriculum and no separate module that assigns tasks, each agent proposing and revising its own goals, and in the main experiments explicit knowledge stores are initialized empty and populated from the group's own exploration. The framework therefore accumulates explicit knowledge from interactions across agents and over time, without requiring an externally curated knowledge base. Over 30 game days on two primary

backbones, OpenKAGE obtains obsidian and enters the Nether, milestones the planner-only baseline does not reach, while reducing mean deaths; starvation deaths fall to roughly a third or a half of the baseline's. A transfer experiment further shows that the accumulated knowledge stores can benefit a newly initialized group on another backbone. These results demonstrate the framework's ability to support open-ended learning through sustained interaction and cumulative knowledge reuse, without an externally prescribed task sequence.

2 OpenKAGE

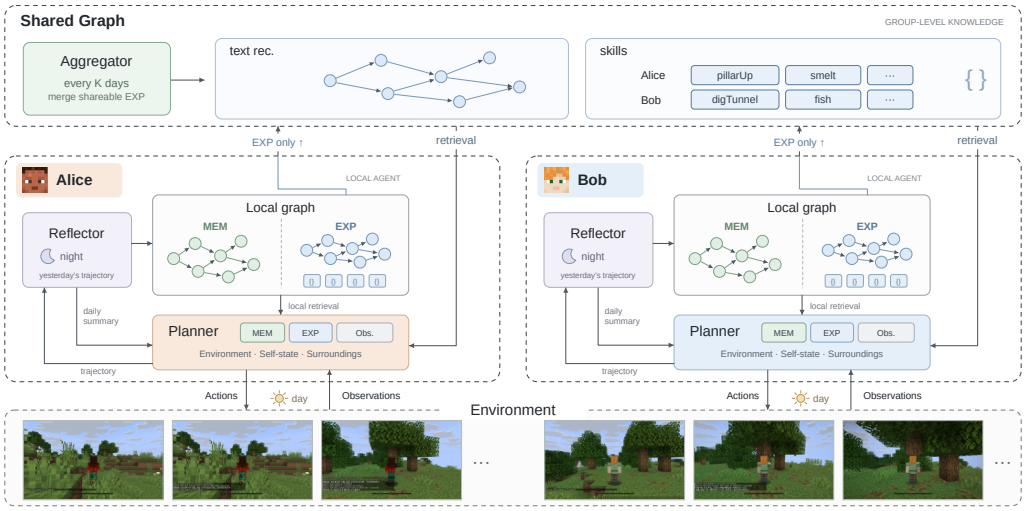

Figure 2: OpenKAGE overview. Each agent keeps a local graph split into a private memory layer and a shareable experience layer; only experience-layer records leave it. Textual knowledge records are consolidated into a group-level shared graph every K days, and skills are shared verbatim under their authors' names. An intent-aware retriever returns what is relevant to the planner. The illustration is schematic; agents receive structured text observations rather than images.

2.1 Overview

OpenKAGE enables a group of agents to accumulate and reuse knowledge from their individual interactions with an open world (Figure 2). Each agent interacts with a separate Minecraft world instance, and agents exchange knowledge rather than in-world messages or resources. Each agent runs a ReAct loop organized into game-day cycles (Yao et al., 2023), using a *planner* to act and a *reflector* to separate situated memories from transferable experience; an *aggregator* periodically consolidates textual knowledge records into a shared graph, while executable skills are shared verbatim through an author-maintained registry; an intent-aware *retriever* supplies relevant subgraphs to the planner. This subsection describes how these pieces operate over time.

Planner. During each game-day cycle, the agent alternates between thinking and acting. Each morning it sets a plan for the day–night cycle (a goal and ordered steps); planning with a goal triggers retrieval from the agent's own graph and the shared graph, so the day's work is informed by previously accumulated group knowledge. The agent then acts by writing short asynchronous JavaScript programs, one program per step, over the environment's base action API (pathfind-and-move, dig, place, craft, equip, . . .). The API provides no task-level routines for gathering, smelting, or shelter construction; packaging them into skills is precisely the agents' job. Executed programs return structured observations (position, health, inventory, nearby entities and blocks, time, advancements). Following Wang et al. (2024), code is the action space: programs are temporally extended, compositional, and directly executable.

Reflector. At the end of each game-day cycle (at dawn), the world freezes and the agent reflects on the elapsed day in two phases. The first re-reads the day's trajectory and writes both layers of its local

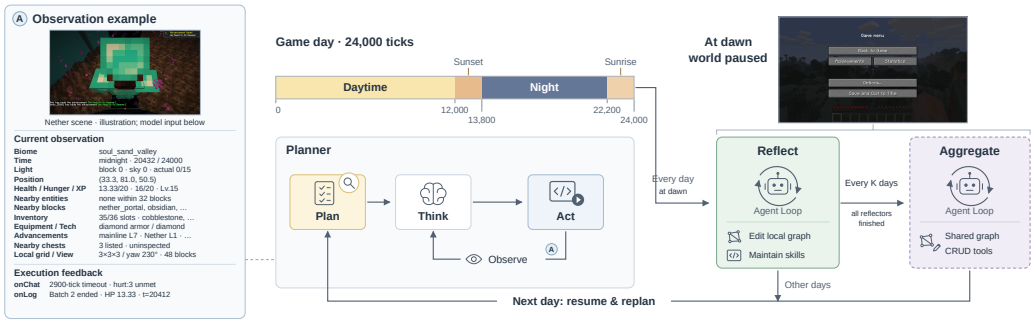

Figure 3: Operation organized into game-day cycles. **Left:** structured observations and execution feedback (schematic). **Right:** within each 24,000-tick day the planner alternates between thinking, acting, and observing; at dawn the world pauses for reflection, and aggregation runs every K days.

graph. It extends the timeline, updates locations and possessions, and proposes decontextualized experience-layer concepts supported by the day's events. The second updates its skills based on per-skill success and failure counts. The reflector also emits a one-line factual summary of the day's progress. Apart from persistent knowledge stores, this summary is the only free-form trajectory text carried from one day to the next: the agent reads the previous day's summary and goals, while everything else is carried by the graphs.

Aggregator. Every K days, after all reflectors finish, aggregation runs, and the agents enter the next window with the shared graph already updated. The loop of act, reflect, aggregate, and retrieve continues for the whole run: competence that any single agent acquires is consolidated where every agent can find it, and the group's knowledge compounds. No step of the loop involves human feedback, a human-authored curriculum, or parameter updates.

2.2 Two-layer knowledge graphs

OpenKAGE represents world knowledge as directed labeled graphs $G = (V, E)$, which each agent's reflector maintains: nodes $v \in V$ are *concepts*, each carrying typed *facets* (short textual statements describing one aspect of the concept), and edges E assert typed relations $u \xrightarrow{r} v$. Each agent i owns a *local graph* G_i consisting of a private memory layer and a shareable experience layer; the group maintains one *shared graph* G^* that consolidates the agents' experience over the run.

Every local graph has two layers, following the complementary learning systems account of memory (Kumaran et al., 2016): $V_i = V_i^{\text{mem}} \cup V_i^{\text{exp}}$, with each concept carrying its layer $\ell \in \{\text{mem}, \text{exp}\}$:

- **Memory layer.** Situated, first-person records of what happened to *this* agent, for instance *day_14 / bucket_fill_and_place*: "Irrigated the farm; water washed away one wheat crop. Repaired the pit and replanted it" (Fig. 6, Appendix A.2). Nodes are bound to the agent's coordinate frame and timeline, forming a day-chained backbone with locations, possessions, and events attached.
- **Experience layer.** Transferable knowledge abstracted from the agent's interactions. Shareable records abstract away agent-specific locations, timestamps, and identities while preserving task-relevant preconditions and applicability conditions. The layer contains two forms of transferable knowledge: *textual knowledge records*, e.g., from the same irrigation episode, "fill / pour with `activateItem()`, not `activateBlock()`; use a separate lower pit to stop flooding"; and *executable skills*, parameterized asynchronous functions, such as `fillAndPlaceWaterBucket` or `digStairTunnel`, written by the reflector and invoked by the planner. Nodes use agent-independent concept names, so they are useful to any agent facing a similar situation.

The two forms influence behavior through different channels. Textual knowledge records act *indirectly*: retrieved into context, they inform how the agent plans. Executable skills act *directly*: injected into the execution environment, the code *directly implements* executable behavior.

The separation is enforced at the level of access: the reflector declares the layer of every concept, and the aggregator is given only the experience layers as input, $V^{\text{in}} \subseteq \bigcup_i V_i^{\text{exp}}$, so memory-layer records are not provided to the aggregator; the reflector is instructed to omit agent-specific coordinates and timestamps from shareable records. The two layers have different access channels: the experience layer is consulted *by intent* (goal-conditioned retrieval), while the memory layer is *injected* each morning as a bounded slice of persistent context, with per-category budgets (episodic, spatial, lesson, status) so that the daily flood of day-records cannot displace the slow-decaying spatial records (camps, chests, mines) that may not be retrieved by goal-directed queries.

2.3 KNOWLEDGE SHARING

Sharing must serve both forms of the experience layer, and the two call for different channels. *Textual knowledge records* reach an agent only through its context, so the group is free to rewrite them into whatever consolidated wording serves best. They are maintained by an aggregator in one shared graph, every K days. *Executable skills* are executed rather than read, so in our design their code is shared without modification by the aggregator: it travels verbatim through an author-maintained registry. We describe the two channels in turn.

Textual-experience maintenance. Every K days, an aggregator reads the agents' experience-layer text and updates the shared graph (Appendix A.6). It consolidates textual knowledge records rather than merely appending them: **merging synonyms**, where several agents' wordings of one insight become a single concept: a vector-similarity pre-screen nominates candidate duplicates, and names already in use are reused, so a category named once keeps its name; **recording cross-agent corroboration**, where a facet states explicitly when experience is confirmed by multiple agents, an unconfirmed claim still being admitted to await corroboration; **reconciling contradictions**, where conflicts are first tested for differing applicability conditions (one observation holds by day, the other fails at night) and corrected in place when one side is simply wrong; and **aligning with prior knowledge**, where new experience is folded into existing concepts wherever they match, instead of piling up near-duplicates.

After a successful aggregation the local textual-experience buffers are cleared, $V_i^{\text{exp}} \leftarrow \emptyset$, so each agent's shareable textual output stays bounded at K days of reflection while the shared graph is maintained cumulatively across aggregation windows.

Skill sharing. Skills bypass aggregation: the group exposes a registry through which every agent retrieves from every agent's library, each skill carrying its author and its cumulative success and failure record. A skill is propagated *verbatim* and selected by its record; the author alone revises or retires it, so what circulates in the group is exactly the set of skills their authors still stand behind, and each skill is evaluated and maintained separately by its author. For skills, code delivery and context presentation use different channels: the code is injected wholesale into the execution environment, while only the signatures of the top-B_{skill}=20 skills with the strongest execution records occupy the context, supplemented by signatures retrieved for the current goal.

2.4 INTENT-AWARE SUBGRAPH RETRIEVAL

The *retriever* is the component bringing the graphs back to agents: it is triggered by intent, and during the day the agent may query or expand at will. Unlike skill-library or RAG memory (Wang et al., 2024; Lewis et al., 2020), it exploits graph structure for intent-aware retrieval, in three stages.

Vector anchoring. The query is embedded and matched against concept embeddings; among the top-k_{anc} candidates, an anchor is admitted only if its similarity is within a ratio ρ of the best match, i.e., its cosine similarity is at least ρ times the best candidate's. The gate is relative because absolute cosine similarities are not comparable across embedding models and corpora. The number of anchors is therefore decided by the semantics of the query, rather than by a fixed number of anchors or an absolute similarity threshold.

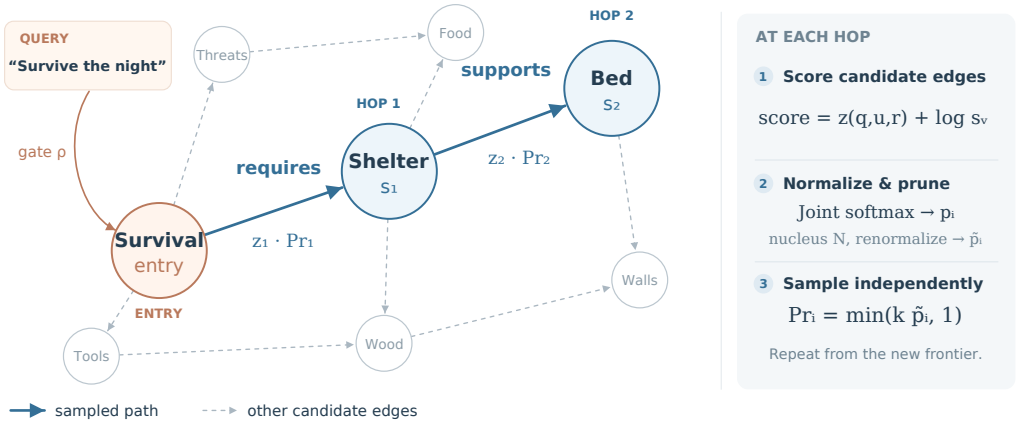

Figure 4: Multi-hop subgraph retrieval (schematic). The retriever selects anchors by relative similarity, then expands along relations scored by query-conditioned relevance and neighbor strength; blue solid arrows trace one sampled path, dashed arrows the other candidates. A structured subgraph is returned, not a ranked list.

Relational expansion. From the anchors, retrieval walks the graph's edges outward for up to h hops. For a frontier concept u with an edge of type r to a neighbor v, the edge score is

$$\text{score}(q, u \xrightarrow{r} v) = z_{q,u,r} + \log s_v, \qquad z_{q,u,r} = \frac{\cos\big(e_q,\, e_{u,r}\big) - \mu_u}{\sigma_u}, \tag{1}$$

where e_q is the query embedding, $e_{u,r}$ is the embedding of the relation's text prefixed with u's name, and μ_u, σ_u are the mean and standard deviation of raw cosine similarities across the relations at u (ties and single-relation nodes fall back to uniform weights). Normalization is within-node rather than global. Here s_v is the neighbor's retrieval-use strength, a monotone counter of how often retrieval has used v; it is initialized at 1 and updated only through retrieval (Appendix A.8).

The unnormalized sampling weight of an edge is thus proportional to $\exp(z_{q,u,r}/\tau)\, s_v^{1/\tau}$, combining query relevance with retrieval-use strength. The candidate edges of a hop, $\mathcal{C}$, are softmax-normalized jointly at temperature τ, so that an anchor with many good edges is not outvoted by a one-edge anchor, then pruned to a nucleus $\mathcal{N} \subseteq \mathcal{C}$ of cumulative mass p_{nuc} and renormalized; each surviving edge is then taken independently with probability

$$\Pr[\text{follow } i] = \min\big(k\,\tilde{p}_i,\ 1\big), \qquad p_i = \frac{\exp(\text{score}_i/\tau)}{\sum_{j\in\mathcal{C}} \exp(\text{score}_j/\tau)}, \tag{2}$$

where score_i is the score of edge i from equation 1, $\tilde{p}_i$ is p_i renormalized over the nucleus, and k is the number of frontier nodes. This rule bounds the expected number of selected edges per hop by the frontier size; it does not enforce equal expansion across frontier nodes, and multiple selected edges may lead to the same node. Lower-probability edges retained in the nucleus may still be sampled: retrieval explores the graph rather than looking it up.

Subgraph recall. The returned representation contains the anchors, the nodes reached by expansion, all their facets, the edges internal to this set, and the edges leading outward (directions available for further expansion, reported as outgoing-relation metadata). The agent receives a structured subgraph, potentially containing multiple connected components, rather than unstructured entries.

3 Experiments

3.1 Setup

Experimental setup. We evaluate OpenKAGE for 30 Minecraft game days on one fixed world seed at hard difficulty, with inventory retained after death. Game time advances only during action execution and pauses during deliberation and reflection. Each group contains $N = 3$ agents

in separate worlds, with distinct personas and aggregation every $K = 2$ days. We use DeepSeek-V4.1-Flash and GLM-5.3, with the same backbone for planning, reflection, and aggregation. Each method–backbone combination has three group runs (nine agent trajectories). Implementation details are in Appendix B.1 and Appendix A.3.

Baseline. The planner-only control uses the same backbone, personas, action API, world seed, and horizon, but has no persistent cross-day memory, reflection, or skill library (Appendix B.3). This comparison measures the combined effect of the knowledge pipeline, not sharing alone.

3.2 Main results

We evaluate OpenKAGE and its baseline on three long-horizon capabilities over 30 game days: exploration progress, survival, and building (settlement). Each cell pools three runs of three agents on one world seed. Metric definitions are in Appendix B.2; detailed data in Appendix B.4.

Table 1: Survival and exploration after 30 game days (three runs of three agents per method–backbone). Deaths and Adv's are per-agent means; Starve is a total. Eq-max is the maximum equipment score (ceiling: 16). Parentheses give attainment counts, with mean first-attainment days reported for Tool era; Obsidian and Nether report mean first-attainment days with counts. Attainment times exclude non-attainers; — means none attained. Metric definitions are in Appendix B.2.

		Survival		Exploration				
Backbone	Method	Deaths ↓	Starve ↓	Eq-max ↑	Tool era ↑	Adv's ↑	Obsidian ↓	Nether ↓
DS-V4.1F	baseline	23.7	13	16 (6/9)	*dmd.* (10.6 day, 9/9)	12.3	—	—
	OpenKAGE	22.0	4	16 (4/9)	*dmd.* (9.0 day, 8/9)	13.1	22.2 day (6/9)	24.0 day (3/9)
GLM-5.3	baseline	39.9	6	14 (3/9)	*dmd.* (7.3 day, 9/9)	10.8	—	—
	OpenKAGE	24.4	3	16 (2/9)	*dmd.* (11.3 day, 8/9)	11.1	23.3 day (4/9)	28.0 day (1/9)

Better exploration. Exploration and survival results appear in Table 1. Diamond-tier tools are attained by most agents under this protocol. What is hard is obtaining obsidian, building a portal, and entering the Nether. In the observed trajectories, agents followed a route involving a diamond pickaxe and an iron bucket, a lava source doused with water, ten blocks mined with that pickaxe, and a frame lit from inside with flint and steel. The gain appears as exploration time grows: only OpenKAGE obtains obsidian, around day 20 and later, and only OpenKAGE goes on to enter the Nether. Wang et al. (2024) observe that open-ended exploration goals are abstract and therefore "challenging to execute without an appropriate curriculum," and address this with a dedicated curriculum module that proposes the next task. OpenKAGE has no such module: the acting agent proposes and revises its own goal each day from its current state and its accumulated knowledge. On both backbones, agents following that self-generated route reach its end within the 30 days. These results show that open-ended learning is sustained through interaction and cumulative knowledge reuse under this protocol, with no task assigned from outside the loop.

Better survival. Mean deaths decrease from 23.7 to 22.0 on DeepSeek-V4.1-Flash and from 39.9 to 24.4 on GLM-5.3. Total starvation deaths across the nine agents decrease from 13 to 4 and from 6 to 3, respectively. OpenKAGE also unlocks more advancements (12.3 vs. 13.1 and 10.8 vs. 11.1 per agent), so the reduction in deaths is not bought by exploring less. These results indicate improved survival under the evaluated protocol.

Table 2: Basic utility blocks remaining in the final world on the DeepSeek-V4.1-Flash backbone. Values are per-agent means over nine agents.

Method	Crafting table	Chest	Furnace	Bed	Total
baseline	9.67	4.11	5.89	2.00	21.67
OpenKAGE	2.67	2.67	1.56	1.33	8.22

We cannot judge building ability from the final-world counts alone. The baseline leaves approximately 2.6× as many basic utility blocks —crafting tables, chests, furnaces, beds—in the final world as OpenKAGE (Table 2). This difference reflects how settled an agent is only to a limited extent, and it does not demonstrate any building advantage for OpenKAGE within the 30 days.

3.3 Ablations

Each condition removes one component from the full system, and conditions are named by what they remove (Appendix B.5, and the column heads of Table 3). All run on DeepSeek-V4.1-Flash, on which the full system and the no-graph control separate most consistently (§3.2); each condition is compared with Full on key milestones. These comparisons provide preliminary evidence about component contributions, but do not establish necessity or interaction-free effects.

Table 3: Ablation results (DeepSeek-V4.1-Flash, 30 game days). Entries are the number of agents that reached the milestone: out of nine for Full and B (three runs pooled) and out of three elsewhere (one run). Conditions are named by what they remove—$-$S sharing, $-$R subgraph retrieval, $-$M the memory layer, $-$E the experience layer, B all graphs—and are defined in Appendix B.5.

	Full	$-$S	$-$R	$-$M	$-$E	B
Obsidian ↑	6/9	1/3	1/3	0/3	1/3	0/9
Nether ↑	3/9	0/3	1/3	0/3	0/3	0/9

Each ablated variant is evaluated in one three-agent group run, whereas Full and the baseline each use three group runs; the values of each condition are in Appendix B.5.

- **Both layers sustain exploration; memory matters more.** Removing either layer lowers milestone attainment: Full reaches obsidian with six of nine agents and the Nether with three, whereas $-$M, which drops the private memory layer while aggregation and the shared graph keep filling, reaches neither, and $-$E, which empties the experience layer so nothing crosses between agents, reaches obsidian with one of three agents and the Nether with none. Each comparison rests on a single three-agent run.
- **Sharing and retrieval govern knowledge flow.** $-$R keeps the graph but flattens retrieval to embedding similarity, so knowledge flows poorly to the acting agent; $-$S empties the shared graph, so there is no flow at all. $-$R reaches obsidian with one of three agents and the Nether with one of three; $-$S reaches obsidian with one of three and the Nether with none. Both fall below Full, which reaches obsidian with six of nine and the Nether with three, and $-$S, with no flow at all, is the weaker of the two.

3.4 Cross-group and cross-backbone knowledge transfer

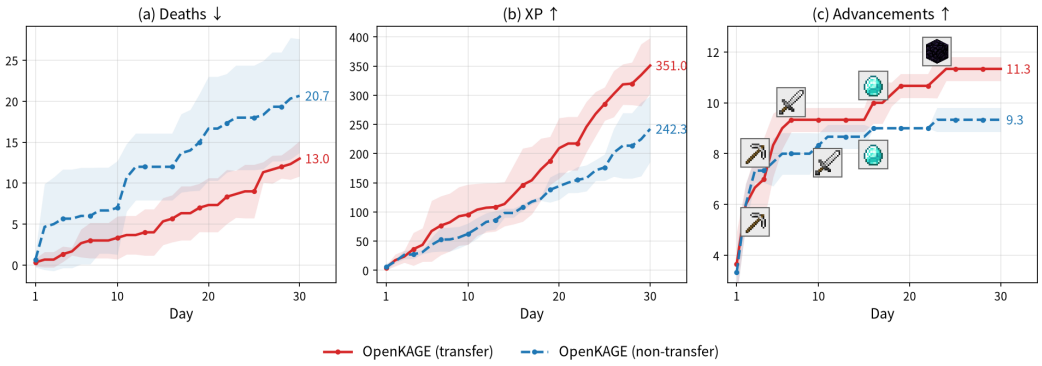

Figure 5: Knowledge transfer. The shared graph and skill library of a DeepSeek-V4.1-Flash OpenKAGE group after 30 days are transferred to a fresh GLM-5.3-Flash group (solid); the non-transfer group (dashed). Curves are three-agent means; bands are ± 1 standard deviation across agents.

This experiment asks whether the accumulated knowledge stores are a transferable asset: does their value survive the group that produced them, and the backbone that produced them? The **transfer group** is a fresh GLM-5.3-Flash group whose knowledge stores are initialized with the shared graph and skill library that a DeepSeek-V4.1-Flash OpenKAGE group had accumulated by the end of its 30 days; the **non-transfer group** is the same configuration initialized from empty stores. Both then follow the identical 30-day protocol.

The two groups separate on all three reported metrics (Figure 5). Deaths fall from 20.7 to 13.0 per agent; final cumulative XP rises from 242.3 to 351.0 ($+45\%$); advancements unlocked per agent rise from 9.3 to 11.3. The extra advancements are diamond and obsidian acquisition: diamonds are reached by three of three transfer agents (days 16, 16 and 24) against one of three, and obsidian by one of three (day 23) against none. In this configuration, initializing a new group with the accumulated shared graph and skill library improves all three metrics over empty-store initialization.

4 Related Work

Open-world learning and agent memory. Voyager combines an automatic curriculum with a growing executable skill library for autonomous Minecraft exploration (Wang et al., 2024): the next task is proposed by that curriculum module and interaction is organized around carrying it out. In OpenKAGE goal generation sits inside the acting agent's own loop, so a goal is proposed and revised against the agent's current state and accumulated knowledge rather than issued by a separate module. The distinction is about how goals are produced and how they govern interaction, not about whether exploration is autonomous: Voyager's curriculum is automatic, and it is evaluated on open-ended exploration. Other agents organize long-horizon behavior around structured knowledge or memory: injected task knowledge (Zhu et al., 2023; Wang et al., 2025), a hybrid graph-and-pool memory (Li et al., 2024), an episodic What-Where-When memory (Park et al., 2025), a knowledge-graph world model (Anokhin et al., 2025), a causal graph (Yu & Lu, 2025; Tang et al., 2024). In OpenKAGE the knowledge stores start empty and are filled only from the group's own interaction. The distinction we draw is therefore not autonomous exploration or dual memory, which these systems already have, but the unit of accumulation: the memory–experience distinction becomes a sharing boundary, so situated records stay local while transferable knowledge is consolidated across agents.

Multi-agent experience sharing. Multi-agent frameworks coordinate through communication and role division (Li et al., 2023; Hong et al., 2024); GEA instead treats the group as an evolutionary unit and pools members' trajectories into a shared set (Weng et al., 2026), and federated continual learning combines knowledge through model parameters (Yang et al., 2024), leaving no inspectable record. OpenKAGE maintains an inspectable shared graph: the aggregator is instructed to merge equivalent records, record corroboration, and reconcile conflicting applicability conditions, and skills follow a separate verbatim-sharing channel. These are architectural distinctions; our experiments do not isolate consolidation from unaggregated sharing or compare against GEA.

Retrieval-based memory. Retrieval-augmented generation (RAG) retrieves external knowledge for generation (Lewis et al., 2020), while GraphRAG and AriGraph exploit graph representations (Edge et al., 2024; Anokhin et al., 2025). In agent memory the same pattern usually yields a flat hit list: items are embedded, a query retrieves nearest neighbors, and the list enters the context (Wang et al., 2024). Such a list ranks by query similarity but does not carry the dependencies, preconditions and failure modes that bind its items to the situation. OpenKAGE combines vector anchoring with query-conditioned relational expansion over the group's evolving graph, returning a subgraph with that structure intact; its focus is retrieving knowledge maintained through cross-agent consolidation, rather than graph retrieval alone. Appendix A.10 provides a detailed system-level comparison.

5 Conclusion

We presented OpenKAGE, a training-free framework that supports group-level knowledge accumulation through private memory, shared experience consolidation, skill sharing, and intent-aware retrieval. Across two backbones in 30-day Minecraft evaluations, it reached milestones absent from a planner-only baseline and reduced mean deaths; cross-backbone transfer provided preliminary evidence of knowledge reuse. Together, these results demonstrate sustained autonomous learning and effective knowledge reuse within the evaluated horizon, supporting OpenKAGE as a practical framework for open-ended lifelong learning.

Limitations and outlook. Independent runs are limited, particularly for ablations, leaving component contributions insufficiently established. Evaluation on stronger backbones and comparisons with stronger memory-equipped and knowledge-sharing baselines remain necessary.

AI USE STATEMENT

Generative AI tools were used in three capacities during the preparation of this work, each subject to human oversight: (i) to aid and polish the writing of the manuscript; (ii) for retrieval and discovery, e.g., locating and surveying related work; and (iii) for research ideation and execution, including assistance with code development and experiment orchestration. All experimental design, data interpretation, and conclusions are the responsibility of the authors.

ETHICS STATEMENT

This work is conducted in the Minecraft simulation environment and involves no human subjects or real-world intervention. The agents' knowledge-sharing mechanism involves no user data.

REPRODUCIBILITY STATEMENT

Experimental settings, prompts, interfaces, and metric definitions are documented in the main text and appendices; Appendix B.1 tabulates the full run configuration with the keys under which each value is stored. Model configurations, interactions, and daily environment snapshots are logged to support result verification. Anonymized code, per-run results, and analysis scripts will be provided to reviewers and area chairs through OpenReview during the discussion period, followed by a public release after review.

REFERENCES

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Andrey Kravchenko, Mikhail Burtsev, and Evgeny Burnaev. AriGraph: Learning knowledge graph world models with episodic memory for LLM agents. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 12–20, 2025. URL https://arxiv.org/abs/2407.04363.

Bowen Baker, Ilge Akkaya, Peter Zhokhov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (VPT): Learning to act by watching unlabeled online videos. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2206.11795.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024. URL https://arxiv.org/abs/2404.16130.

Linxi Fan, Mengyao Gong, Mengnan Zhong, Xuan Zhang, Jiannan Liu, and Anima Anandkumar. MineDojo: Building open-ended embodied agents with internet-scale knowledge. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2022. URL https://arxiv.org/abs/2206.08853.

Yuchen Guo, Junli Gong, Weicheng Wang, Hongmin Cai, Yiu-ming Cheung, and Weifeng Su. PEAM: Parametric embodied agent memory through contrastive internalization of experience in Minecraft. *arXiv preprint arXiv:2605.27762*, 2026. URL https://arxiv.org/abs/2605.27762.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023. URL https://arxiv.org/abs/2301.04104.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, pp. 23247–23275, 2024. URL https://arxiv.org/abs/2308.00352.

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL https://arxiv.org/abs/2005.11401.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 51991–52008, 2023. URL https://arxiv.org/abs/2303.17760.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 49881–49913, 2024. URL https://arxiv.org/abs/2408.03615.

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. URL https://arxiv.org/abs/1802.07569.

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 (UIST)*, 2023. URL https://arxiv.org/abs/2304.03442.

Junyeong Park, Junmo Cho, and Sungjin Ahn. MrSteve: Instruction-following agents in Minecraft with what-where-when memory. In *International Conference on Learning Representations (ICLR)*, pp. 16922–16960, 2025. URL https://arxiv.org/abs/2411.06736.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 8634–8652, 2023. URL https://arxiv.org/abs/2303.11366.

Significant Gravitas. AutoGPT. https://github.com/Significant-Gravitas/AutoGPT, 2023. Autonomous LLM agent loop; accessed 2026-09-25.

Xiaojuan Tang, Jiaqi Li, Yitao Liang, Song-Chun Zhu, Muhan Zhang, and Zilong Zheng. Mars: Situated inductive reasoning in an open-world environment. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, pp. 17830–17869, 2024. URL https://arxiv.org/abs/2410.08126.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Zihao Wang, Shaofei Cai, Anji Liu, Hongyuan Jin, Chao Hou, Ying Zhang, Xiaojian Lin, Xiangyu He, Zilong Zheng, Yu Yang, Xiaojian Ma, and Yitao Liang. JARVIS-1: Open-world multi-task agents with memory-augmented multimodal language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(3):1894–1907, 2025. URL https://arxiv.org/abs/2311.05997.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-evolving agents: Open-ended self-improvement via experience sharing. *arXiv preprint arXiv:2602.04837*, 2026. URL https://arxiv.org/abs/2602.04837.

Xin Yang, Hao Yu, Xin Gao, Hao Wang, Junbo Zhang, and Tianrui Li. Federated continual learning via knowledge fusion: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 36(8): 3832–3850, 2024. URL https://arxiv.org/abs/2312.16475.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

Shu Yu and Chaochao Lu. ADAM: An embodied causal agent in open-world environments. In *International Conference on Learning Representations (ICLR)*, pp. 22886–22905, 2025. URL https://arxiv.org/abs/2410.22194.

Xinping Zhao, Xinshuo Hu, Zifei Shan, Shouzheng Huang, Yao Zhou, Xin Zhang, Zetian Sun, Zhenyu Liu, Dongfang Li, Xinyuan Wei, Youcheng Pan, and Yang Xiang. KaLM-Embedding-V2: Superior training techniques and data inspire a versatile embedding model. *arXiv preprint arXiv:2506.20923*, 2025. URL https://arxiv.org/abs/2506.20923.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the Minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023. URL https://arxiv.org/abs/2305.17144.

A METHOD

All prompts quoted in this appendix are given in English translation; the Chinese originals that drive the runs are released with the code.

A.1 THE OPENKAGE ALGORITHM

A run is a group of N agents, each in its own world, playing D game days; agents never meet in-world and share only at the knowledge layer. Each day has two parts, acting by day and reflecting at night, and every K days the group consolidates. The agents run in parallel, and a barrier holds the group at the end of each night and again after each consolidation, so no agent begins a day before the group's knowledge for the previous one is final.

Algorithm 1 OpenKAGE, one run. Stages marked *barrier* block until all N agents have reached them.

Require: group of N agents with local graphs G_a; shared graph G^*; D days; window K

for $d = 1$ **to** D **do**
  for each agent a **in parallel do**
    inject standing memory (per-category budgets) and the top-20 skill signatures
    plan a goal for the day; retrieval returns a subgraph of $G_a \cup G^*$ keyed to that goal
    repeat
      observe, write and execute one program over the atomic API or a skill from the registry, take feedback
      re-planning re-triggers retrieval
    until the day ends
    night, stage 1: reflection writes V_a^{mem} and V_a^{exp} from the day's trajectory
    barrier: every agent has finished stage 1
    night, stage 2: skill maintenance against the day's per-skill tallies
  end for
  barrier: night d complete {day d's knowledge is final}
  if $d \bmod K = 0$ **then**
    consolidation merges $\{V_a^{\text{exp}}\}_{a=1}^{N}$ into G^*; on success $V_a^{\text{exp}} \leftarrow \emptyset$
  end if
  barrier: day $d{+}1$ begins with G^* already updated {no lag}
end for

Two consequences of this schedule are worth stating. First, because consolidation is synchronous rather than running in the background, every agent entering day $K + 1$ already sees the conclusions of window K in the shared graph, so the knowledge feedback loop has no lag. Second, because the private experience layer is cleared once per window, it holds one window's output rather than the whole trajectory's, which keeps its size bounded as D grows. The consolidation step itself, namely what it merges, is given in Appendix A.7.

A.2 KNOWLEDGE GRAPH SCHEMA

This appendix specifies the data model behind §2.2: what a concept carries, what a facet and an edge are, how the layer and the memory category are declared, the two kinds of experience node, and which component may write each field. Each agent's local graph and the group's shared graph use the same model; they differ in who writes them and in which layer they carry. Concept names, facet labels, and relation names are free text in the agents' working language (Chinese in our runs); as elsewhere in the paper we quote the identifier and give an English gloss, and the originals are released with the code.

Concepts. A concept is a stable identity for one thing the agent knows: a day, a place, a procedure, or a hazard. It is the unit at which the graph is addressed, retrieved, and consolidated (Table 4).

Two properties of this table matter downstream. First, `layer` and `category` are declared once, at creation, by the reflector, and are not re-derived from content afterwards; the layer is a property of the concept, not of any one facet, so a concept that later acquires facets of another kind does not

Table 4: Concept fields. **Who writes:** R = the reflector (private graph), A = the aggregator (shared graph), Ret = the retrieval path of the acting agent. No other component may write any field.

Field	Content	Written by
`slug`	ASCII snake-case identifier, unique; the handle every operation addresses	R / A
`name`	free text; a *proper* name in the memory layer, a *common* name in the experience layer	R / A
`summary`	one-line statement of what the concept is	R / A
`layer`	mem or exp, declared when the concept is created	R
`category`	`episodic`, `spatial`, `lesson` or `status`; empty for experience concepts	R (memory only)
`strength`	real, initialized to 1.0; monotone, and only the retrieval path writes it	Ret only
timestamps	`created_at`, `updated_at`: game day in the local graph, window index in the shared graph	R / A, Ret

silently change layers. Second, `updated_at` advances both when the content changes and when retrieval returns the concept. Being used counts as being active, which is what makes it the ordering key for the morning injection.

Facets. A concept carries any number of facets, each a short statement of one aspect: a label `kind` and one to two sentences of `content`. At most one facet of a given kind may exist under one concept, because the pair (concept, kind) is unique. Extending what is known about an aspect therefore rewrites that facet in place instead of appending a second one. The kind is a short class-label the reflector names itself, not an entry from a fixed vocabulary; the criteria it is given (name a class, answer a question of the form "where is it" or "how is it done", and reuse the names visible in the graph index) are stated verbatim in the phase-1 prompt of Appendix A.5.

Edges. An edge is typed and directed, $u \xrightarrow{r} v$, and is unique per (from, to, relation), so the same pair may be linked by several relations. The relation is free-named by the reflector under one constraint: the name must read as a predicate, i.e. "*from r to*" must read as a sentence. This is why §2.4 scores a relation by embedding its name together with the concept it leaves from: a vague name such as `related_to` is semantically distant from a specific query and therefore tends to score low. That is a property of the scores, not a rule: no check rejects a relation name, and whether an edge is traversed is decided by sampling (Appendix A.8), so no relation is excluded by construction. Edges are also the one place where the two layers meet: a local graph links an experience concept to the memory episodes that support it, and the aggregator is shown the names of those memory-side endpoints as labels only. This one is a property of the tool interface: the aggregator's tools take no argument that addresses the memory layer, so it can neither read memory contents nor write them into the shared graph (Appendix A.7).

Layers. The layer tag separates what is situated from what is transferable. The reflector applies a single test at creation, given verbatim in Appendix A.5: could someone who did not live that day take this node and use it as it stands? Memory concepts never leave the agent; experience concepts are what the aggregator reads, and are cleared from the local graph once a window has been consolidated. Category is declared for memory concepts only; it is empty for experience concepts, which are selected by retrieval rather than injected. Figure 6 shows examples of both layers from the persisted runs.

Memory categories and the morning budget. The memory layer is injected each morning as a bounded slice of standing knowledge. The slice is drawn per category rather than by a single recency rank, because the four categories go stale at different rates, and a single rank lets the day records written every night crowd out everything else.

The budgets reserve capacity for spatial and status records, which are few and slow and which a similarity query is unlikely to surface: the query "where is my chest" is not similar to "what I intend to do now", so these records get reserved capacity instead of competing with the daily flow of day records. A memory concept with no declared category is not injected at all, though it stays reachable by retrieval, and the reflector's own view of the memory layer at night is a wider index of the most



Figure 6: Situated memory and transferable experience. Five examples from different agents and runs: **left**, private memory keeps episode-specific events and locations; **right**, the experience layer holds reusable textual knowledge records and executable skills. Arrows mark thematic correspondence, not graph edges.

recent concepts rather than the injected slice. In 30-night runs the memory layer ends with on the order of 10^2 concepts, with per-agent ranges of roughly 30–112 episodic, 7–60 spatial, 0–18 lesson and 1–2 status, so most buckets are saturated and the caps bind. Two short texts ride along with the layer and go wherever it goes: the goal the agent set for itself the previous day, and the one-sentence summary of that day's progress produced at dawn. They are the layer's textual surface in the planner's opening message, and a condition without the layer carries neither (Appendix A.4).

The two experience node types. Both live in the experience layer, and both are what leaves the agent, but they are stored and shared differently because they act differently (§2.2). **Textual knowledge records** are ordinary concepts with $\ell = \text{exp}$: facets hold agent-independent statements, and there is no numeric consensus field. Cross-agent corroboration is written into the content itself, and a corrected belief is kept as a further facet recording what was believed, what is true, and how the change was found, rather than being overwritten silently. **Executable skills** are separate records, holding the intent, signature, code, flags, and count pairs specified in Appendix A.9. It is written by

Table 5: The four memory categories and their injection budgets. A bucket takes the concepts of that category in decreasing order of `updated_at`; budgets are caps, not targets, and are not redistributed across buckets.

Category	Holds	Goes stale	Budget
`episodic`	what is bound to a specific day: the day record, that day's blockers and findings	within a few days	18
`spatial`	durable assets with a location: camps, chests, mines, workbenches	slowly, when exhausted	10
`lesson`	what an encounter left behind: deaths, misleading clues	not with time	4
`status`	the rolling self-model: current progress, backlog, equipment plan	overwritten daily	3

its author's reflector from a program that ran successfully that day, revised or retired only by that author, and crossed between agents as code, untouched.

Strength. Every concept carries one real-valued strength, initialized to 1.0 and monotone; the mechanism is as in §2.4. Only the retrieval path of the acting agent writes it, by the two gains of Table 10, and no operation in the reflector's or the aggregator's tool sets writes it. Strength records use, not authorship.

What the model deliberately does not carry. Embeddings are not fields of the graph, so retrieval can be retuned without touching the knowledge. Facets and edges carry no strength and no decay or archival flag. Usefulness is a property of what the agent reaches for, and everything the agent reaches for is a concept. And no node points back into the trajectory it came from: the day's raw events are compressed into the graph at dawn, and the only text that survives into the next day unchanged is the one-line summary of progress.

Enforced and requested properties. The sections below mix two kinds of property: those the implementation guarantees, which the tool layer checks and on violation fails loudly, and those the prompts ask a model to produce, which are standing instructions whose observance nothing checks at run time and which we do not audit. The first kind is stated where it applies; the second lives entirely in the prompts released with the code.

A.3 Agent–world interface

The agent's only way to change the world is to write a short asynchronous JavaScript program, one per step, over an *atomic* action API exposed through the Mineflayer bot library; it never emits low-level controls. Programs are temporally extended, compositional, directly executable and, once verified, can be saved as skills (§2.1). This appendix lists the API in scope, the execution semantics of a step, and the observation returned after one.

The atomic API. Everything below is already in the scope of the generated program; nothing is imported.

One feature of this list is deliberate. There are no *capability* primitives, that is, no `mineBlock`, `craftItem`, `smeltItem` or `killMob`, names that are Voyager's: with the API atomic, a composite capability like "smelt iron" is not a primitive the agent is handed: it composes it in its own program, and the skill channel is what lets such a composition persist and travel to the other agents rather than being the only way to perform it (Appendix A.10 gives the argument in full). The single exception is `waitUntil`, because the client's own tick-wait is decoupled from the server clock, so a hand-written wait loop cannot move the world forward at all.

Execution semantics. One `write_code` call is one step. The program is evaluated as the body of an asynchronous function; world-changing calls must be awaited, because an unawaited one keeps acting on the world while the agent has already moved on to the next step. The agent declares its own budget for the step in *game ticks*, with a default of 500 and bounds of 100 and 3,000; a full day–night cycle is 24,000 ticks, and the budget is spent in world time, not wall-clock time. A step ends when its budget is spent, when the agent dies, or at midnight, whichever comes first. Between steps the world is frozen, so game time advances only while a program is executing (§3.1).

Observation. Every step returns the execution result together with a complete fresh observation, a text rendering of world state rather than pixels. Fields are fixed in content and order; their sensing radii differ, and each line states its own, so the agent never has to align them by guesswork.

The three spatial fields answer different questions and are not redundant: the adjacent grid is read through walls and answers whether the agent can stand, move or dig where it is; the vision field is ray-traced with real occlusion, covers only the horizontal $\pm 60^\circ$ and vertical $\pm 30^\circ$ in front of the agent, and answers what lies that way and whether it is worth walking to; the block-type set is a directionless overview of what this neighbourhood contains. Two conventions hold throughout: directions are absolute axes ($+x$, $-z$, $+x-z$, …) rather than left/right or compass points, so a direction read from one field can be copied into a movement call; and every field reports facts only.

Table 6: The atomic action API, in the scope of every generated program.

Group	Calls
Pathfinding	`bot.pathfinder.goto` with a goal object: `GoalNear`, `GoalNearXZ` (within a radius of a point or a column), `GoalBlock`, `GoalXZ`, `GoalY` (a cell, a column, a height)
Goals for objects	`GoalGetToBlock`, `GoalLookAtBlock` (reach a block to touch or see it), `GoalPlaceBlock`, `GoalFollow`, `GoalInvert` (retreat), `GoalCompositeAny` / `GoalCompositeAll`
Direct control	`bot.setControlState` for movement keys, `bot.look` and `bot.lookAt` for turning
Blocks	`bot.findBlock`, `bot.findBlocks`, `bot.blockAt` to read; `bot.dig` to mine one block (no tool choice, no approach, no pickup); `bot.placeBlockSafe` to place the held block on a face of a reference block, returning where it landed; `bot.collectBlock.collect` for a small batch
Crafting	`bot.recipesFor`, `bot.recipesAll` for the recipes of an item id; `bot.craft` to execute one, given the recipe, a count, and the crafting-table block where one is needed
Inventory, containers	`bot.inventory.items()`, `.count()`, `.emptySlotCount()` over 36 slots; `bot.openContainer` and `bot.openFurnace` return a window whose `withdraw` and `deposit` move items
Equipment, consumption	`bot.equip` and `bot.unequip` over six destinations, namely four armour slots, main hand, and off hand; `bot.consume`, `bot.sleep`, `bot.toss`
Combat	`bot.nearestEntity`, `bot.entities`; `bot.pvp.attack` and `bot.pvp.stop` for melee, `bot.hawkEye.autoAttack` at range
Self and time	`bot.entity.position`, `bot.health`, `bot.food`, `bot.heldItem`, `bot.entity.onGround`, `bot.time`
Data handles	`mcData` for item and block ids, recipes and drop tables; `Vec3` for coordinates; `console.log`, whose output comes back with the step
Waiting	`waitUntil` for a number of ticks, or until the first of a set of events, namely `hurt:N`, `healed`, `fed`, `no_mobs_nearby:R`
Skills	`await skillName(bot, ...)`; skill definitions are placed in the same scope

The rules that turn those facts into decisions (for instance the light thresholds at which hostile mobs spawn) are stated in the agent's standing instructions, and applying them is the agent's job.

A.4 THE PLANNER

The planner is a single ReAct episode that runs from dawn to dusk: it is given two standing messages and one opening message, and then acts through six tools until the day ends. Everything it is told is identical across agents and across conditions; the only thing that varies per agent is its persona. Prompts are given below in English translation; the originals that drive the runs are released with the code.

Standing messages. The first is the persona: the agent's name and a short description of how it tends to explore. This is the only per-agent difference in the whole system (§3.1): Table 8 lists the eight personas used in this paper, three in the main groups and five more in the group-size study of Appendix B.6, each written from the same template and differing only in the tendencies it names. The second is the domain prompt, in four parts: *who you are* (three long-horizon directions, namely survive, explore, and build, of which each day's goal is one small step), *how you act* (one JavaScript program per step, with an explain line, the code, and a tick budget; no infinite loops), *how you reach the world* (the atomic API of Appendix A.3), and *what you see* (the observation of Appendix A.3, facts only). The full text follows below.

The opening message. At dawn the previous day's conversation is discarded, and the day opens with one user message assembled from five blocks, each skipped when its value is empty: the day number with the instruction to plan and not to stop calling tools; *yesterday's goal*, the goal text the agent itself set, carried across verbatim; *yesterday's progress*, one factual sentence produced at dawn (no night-time episodes, no goal-setting, because the agent decides that itself from the

Table 7: Observation fields returned after each step.

Field	Radius	Content
Biome	own cell	biome of the block underfoot
Time	—	phase label (`sunrise`, `day`, `noon`, `sunset`, `night`, `midnight`) and the current tick out of 24,000
Light	own cell	block light and sky light, each 0–15, reported separately: the first is what the agent can change (torches), the second is exposure to sky and does not fall at night
Position	—	absolute coordinates, to one decimal, since block operations are sensitive to a one-block difference
Vitals	—	health and food out of 20 with saturation, XP level and points, and oxygen when it is not full
Nearby entities	32 blocks	per type: how many, the distance to the nearest, and the absolute-axis directions of the three nearest
Nearby block types	17×5×17	the set of block types present, deduplicated and without direction
Inventory	—	items carried, and slots used out of 36
Equipment	—	the six worn and held slots, empty shown as none, with remaining and maximum durability; these slots are *not* part of the inventory line
Progress	—	for each advancement tree, the deepest advancement reached in it
Tech	—	tier of the best finished tool carried or worn
Nearby chests	16 blocks	coordinates and known contents: seen but never opened, opened and empty, or the items recorded
Adjacent grid	3×3×3	the 25 cells around the body in three layers (head, feet, below), each with an absolute direction; `unknown` means unloaded, not air
Vision	48 blocks	sectors by absolute axis and vertical level, each giving the block-type share with near–far distances, the fraction of rays that reach the full range, and the nearest point's coordinates

situation it wakes into); the memory block of Appendix A.2, followed by a count of the shared graph's contents, since shared experience is *not* injected but read by intent through `retrieve`; and the first observation of the day, placed last and behind a boundary marker. Apart from the graphs, these blocks are all that crosses the day boundary. If the agent stops calling tools while the day is not over, a fresh episode is started with the same standing context, its current plan, and the instruction not to stop.

Tools.

What comes back. A `write_code` call returns the execution feedback together with a complete fresh observation, so the agent never has to ask what the world looks like now. A `createplan` call, or a revision, returns the retrieved experience and the skill signatures as well: planning is what keys retrieval to an intent (§2.4); the signature block is ranked as in Appendix A.9, and the code of every skill is in scope regardless. At nightfall the framework itself issues one retrieval, with a fixed query covering shelter and defence as well as productive work, and appends it to that step's feedback: without it the agent reads dusk purely as a signal to hide, and stops finding things to do. All of this injection is incremental: concepts and signatures already shown earlier in the day are not shown again. The day's conversation is carried in full and discarded at dawn; no truncation threshold is applied while it runs (Appendix B.1), and each step appends one program, one observation and one feedback block.

The domain prompt (English translation). The full system prompt that follows the persona; {placeholders} are filled per configuration. The action and observation catalogues it references are Tables 6 and 7; sections that restate those tables are elided here.

```
## 1. Who you are and what you are here to do

You advance three long-horizon directions in this world. The goal you set
each day with createplan is the small goal for one day-night cycle; these
```

Table 8: The eight personas. An agent receives its own row as the first standing message; nothing else in the system varies per agent. Each is written from the same template, an MBTI cognitive-function stack, the type is in parentheses after the name, which names what the agent is drawn to and what it systematically underweights, namely its dominant and inferior functions. The full text is released with the code, together with eight further personas not used here.

Agent	How it tends to explore	What it underweights
Alice (ESTP)	Enters the situation first and trusts what the world feeds back now over anything reasoned out in advance; improvises quickly and dislikes standing still.	The longer-run cost that a repeated short-term gamble is quietly accumulating.
Bob (INFP)	Goes where the thing itself feels worth doing—new terrain, an untried method—and keeps options open rather than committing to a direction early.	Structure and commitment: will hold back and under-push rather than stake on a plan it cannot yet see through.
Carol (ISTJ)	Checks what is already known before spending a step on it: reuses what worked verbatim, skirts what only partly matches, and treats gaps as the only places worth new effort.	Open possibility; at night tends to audit and qualify what it already recorded rather than widen it.
Dave (ISTP)	Works in small pieces, one thing at a time, and wants a verdict on each step before starting the next; probes cheaply first and states the state before and after.	The shape of the whole: reports one step's detail precisely, rarely what the day added up to.
Erin (ISFJ)	Refuses any bet whose loss would wipe out what has been accumulated; keeps a fixed home and works out and back from it in short trips instead of carrying everything onward.	Uncertain upside; as risk rises it drops productive work early to secure safety.
Frank (ENTJ)	Sets a target first, works backwards to what reaching it requires, then pushes through in one pass; re-targets outright when the target proves not worth it.	Whatever cannot be scored against the plan, and the option of waiting: standing still is the one thing it treats as always wrong.
Greg (ENTP)	Treats any settled method as the thing most worth doubting; when stuck it changes the premise rather than retrying the same approach.	Follow-through and fine detail: abandons a half-finished line for a more promising idea.
Helen (INTP)	Wants the mechanism before the move: why a method works, and under what condition it stops working; derives first and probes to confirm a conjecture.	Time and momentum; will keep turning a question over rather than act on a partial answer.

Table 9: The planner's tools. Only `write_code` changes the world.

Tool	What it does
`createplan`	Sets the goal for this day–night cycle and the ordered steps it decomposes into. Planning with a goal triggers retrieval, whose result comes back with this call.
`updateplan`	Marks a step running, succeeded or failed with a note, or replaces the remaining steps given a reason. Revising triggers retrieval keyed to that reason.
`retrieve`	Queries the graphs by intent: which layer (experience or memory), which scope (own, shared or both), and how many hops of relational expansion.
`expand`	Deepens from one concept along its relations, optionally along one relation only. This is the deliberate, high-precision form of use that scores strength (Appendix A.2).
`retrieve_skills`	Looks up skill signatures by intent, up to three. The code itself is already in scope, so a signature is all that is needed to call one.
`write_code`	Executes one program: the only tool that changes the world.

```
three are where you are headed.

Survive: keep your health, stock resources, avoid danger,
extend survival.
Explore: seek out the unknown, take risks, widen your horizons, keep
upgrading.
Build: raise a base, set up crafting tables and furnaces, make camp, make
a home.

---

## 2. How you act: one JavaScript program per step

You direct each step by writing a JavaScript program. Each step is one
write_code call with three fields:

  explain       a short statement of this step's plan and why
  code          an executable ASYNC snippet (await directly; no
                synchronous infinite loops -- they halt everything,
                including time itself)
  timeout_ticks the execution budget for this action, in ticks
                (one day is 24000 ticks): default {default_ticks},
                maximum {max_ticks}

One step does one clear thing, so progress accumulates. No infinite loops
or recursion.

### The await rule

Every action that changes the world or waits for it (dig, place, craft,
open container -- everything marked await in the table) is asynchronous
and MUST be awaited. An unawaited one becomes an orphan background task
that keeps manipulating the world while you have already moved on to the
next step. Pure reads (bot.blockAt, bot.findBlock) are synchronous; no
await.

---

## 3. How you reach the world

[The full atomic-action table is in the interface appendix. Everything is
in scope without imports: bot, mcData, Vec3, the pathfinding Goal
classes, waitUntil, and skills -- await skillName(bot, ...).]

---

## 4. What you see after each step

[Each step returns an observation with fixed fields in fixed order; the
full field table is in the interface appendix. All fields report facts
only -- whether a place is reachable or worth going to is yours to
judge.]

### The adjacent 3x3x3 grid: the 25 cells around the body

Three layers (above the head, at the body, below the feet), each cell
tagged with an absolute-axis direction -- this is what decides where you
can stand, move, or dig. unknown means the chunk is not loaded; it does
NOT mean air, and it is not walkable.

### Vision: real occlusion, not an x-ray

Rays are cast from the eyes; a ray stops at the first block -- what a
hill or a tree hides, you do not see (unlike the 25-cell grid, which
reads through walls). Sectors are keyed by absolute axis and vertical
level (level = straight ahead -- the layer you will actually walk into;
up = raised; down = lowered). Each sector gives: the block types as
shares of the field of view with near-far distances (the dominant share
is that direction's main body -- a forest, a cliff face); open N% -- the
fraction of rays that travel the full 48 blocks unobstructed (that
direction is empty); and the nearest point's absolute coordinates --
directly usable for walking or digging. Vision covers only horizontal
+-60 degrees and vertical +-30 around the current heading: your back is
blind. Walking and digging change your heading, and with it which axes
this step could see.
```

A.5 THE REFLECTOR

At dawn the world is frozen and the day is closed out. The reflector runs in two phases, each a separate session with its own tool set, and the group is held at a barrier between them so that no agent starts the next phase before every agent has finished the previous one. What follows is given in English translation; the originals that drive the runs are released with the code.

Phase 1: writing the graph. The reflector is told that tomorrow its context will be empty and that the graph is the only thing it carries across, so this is a compression, not a transcript, and choosing what to keep is its main judgement. It is given:

- its persona, the day number, and the identifier to use for the day's own concept;
- the day's server-authoritative facts: what new items it obtained, where each advancement tree stands, how many steps it took, its deaths grouped by cause, and the milestones it reached;
- the full trajectory of the day, that is, each step's program, the observation it acted on, and the feedback it got back;
- an index of its own graph: every concept with its layer tag, the kinds of facet it carries and the relations it has, but not the content, which is read on demand;

It is then asked to write both layers: for the memory layer, the day's concept chained to the previous day and a concept with numeric coordinates for every place reached or built; for the experience layer, whatever generalises into a sentence that holds for someone else, under a common name. The operative rules it is given, namely the layer test of Appendix A.2, the per-category skeleton, how each field is named, and the skill-creation gate, are stated verbatim in the prompt below.

Tools. Phase 1 has one narrow tool per operation over concepts, facets, edges and skills: create, update, merge and delete a concept; search and fetch one; add, rewrite and delete a facet; and add, change or remove a relation. Nothing in this set can score a concept: strength is written only by retrieval (Appendix A.2), and none of them revises or retires a skill, which is what the phase-2 session is for; it carries five skill tools of its own, namely create, list, search, revise and retire.

Phase 2: skill maintenance. The second session maintains the agent's own skill library, with only the five skill tools, against what the library actually did during the day. Its brief is to act only on what is failing, to leave what works alone, and to retire what it cannot fix. The full maintenance loop, and its prompt, are given in Appendix A.9.

The one-line summary. A separate, tool-free call at dawn produces the *Yesterday's progress* block of Appendix A.4: one factual sentence on what advanced on the previous day's goal. It carries no night-time episodes and no goal-setting, because the next day begins in different light and among different threats.

The phase-1 system prompt (English translation). {placeholders} are filled per day. Section 3 is the same representation the aggregator is given (Appendix A.7), with the two fields only the reflector declares: layer and category; it also carries what the reflector must read off the graph index before writing. The prompt opens with the three things the reflector does (one per layer, plus skills), then what the two layers are.

```
You are maintaining YOUR OWN private graph. Tomorrow your context will be
emptied -- this graph is the only thing you carry across. So this is
a COMPRESSION, not a transcript: choosing what to keep is the most
important judgement you make now.

## 1. The three things you do

1. The memory layer -- what happened today: create the day's concept
   recording the day (WITH coordinates, the day number, the outcome)
   and chain it to the previous day by a relation that reads as
   precedence; one concept per place you reached or built today, with
   concrete coordinates.
2. The experience layer -- what today generalises into: conclusions
   under common names that hold for someone else. What failed, killed
```

```
   you or blocked you today can almost always yield one (why it went
   this way, under what conditions not to do it again); the advancement
   you reached today becomes one common-name experience, so that
   someone who was not there can follow it. One experience for a whole
   day is almost always a miss -- but do not PAD it by filing "where I
   was today" in different words. Link the experiences to each other
   with manage_edge: the directions most often walked are what does it
   require, what does it lead to, when does it stop holding.
3. Skills -- the flows that ran today: generalise them into a
   parameterised async function and store them in the skill library
   (the gate is in section 4).

## 2. The memory layer and the experience layer

                       memory layer            experience layer
  the name is          a PROPER name:          a COMMON name:
                       "day_2", "my_camp_1"    "night_movement_risk"
  it answers           where am I, what        how is this kind of
                       happened, what is       thing done, how does
                       unfinished              it go wrong
  person               first person, WITH       de-subjectified: no
                       provenance (day,        "I", no day number,
                       coordinates)            no coordinates
  the test             someone who did not     someone else can take
                       live that day cannot    it and use it as it
                       take it                 stands
  who reads it         only tomorrow's you     it WILL be aggregated
                                               into the shared graph

The first is the raw material of the second: "on day 2 I was killed by a
zombie at Y=12 with no torches" is memory; "stock enough torches before
entering an unlit area" is experience.

- WRITE BOTH LAYERS: memory only -> what you distilled is shared with
  no one (the experience layer is the ONLY thing aggregated);
  experience only -> tomorrow's you does not know where camp is or how
  far it got.
- What you cannot generalise, leave in memory honestly: a single event
  written up as a universal law is worse than no entry at all.
- Keep the raw case and the distilled conclusion as separate records --
  distillation loses detail, and the raw case is what re-distillation
  runs on.

Four categories of the memory layer: every memory concept declares one.
Tomorrow you wake with an empty context and only part of the memory is
injected -- the category decides which part: different categories go
stale at different rates.

  episodic  bound to one specific day: "day_3", that day's blockers
            and findings                          -- stale within days
  spatial   durable assets with a location:       -- stale slowly
            camps, chests, mines, workbenches        (a mine empties)
  lesson    what an encounter left behind:        -- never stale
            deaths, misleading clues
  status    the rolling self-model: current       -- overwritten
            progress, backlog, equipment plan        daily

The criterion is how the thing goes stale, not what it is about: "on
day 3 I found a mine to the east" is episodic (bound to day 3); "that
mine to the east" (with coordinates, existing across days) is spatial.
status is ONE concept, updated in place every day with update_facet --
thirty days of "current state" concepts means there is no current
state.

The skeleton of the memory layer:
- A timeline: one concept per day, chained by a relation that reads as
  precedence; the day's events hang on the day's concept.
- Anything with a location (camp, chest, mine) carries CONCRETE
  COORDINATES as numbers, "(112, 68, -340)" -- "to the east" is as good
  as nothing -- attached by a relation that reads as location or
  containment. Coordinates are the only information that never goes
  stale across days (within one trajectory the world is not
  regenerated), and the anchor by which "what I was doing yesterday"
  continues.

## 3. Knowledge representation: the graph structure

Your private graph is made of three things, and these fields are all you
can write:

| What      | Fields                                  | You decide          |
```

```
|-----------|---------------------------------------|--------------------|
| Concept   | slug / name / summary / layer / category | all of them        |
| facet     | kind / content                        | kind; content is the text |
| edge      | from --relation--> to                 | relation           |

- slug: ASCII snake-case, unique in the graph, the only handle you have
  for get_concept / manage_edge later (a-z / 0-9 / underscore only).
- name: the concept's name, in Chinese. Which kind of name depends on
  the layer, as section 2 gives it (proper names in memory, common
  names in experience).
- summary: one sentence saying what the concept is.
- layer: memory or experience, declared when you create the concept,
  criterion in section 2. The layer is a property of the concept, so
  the two kinds of thing do not go into one concept.
- category: required for memory concepts, one of the four, criterion in
  section 2. Not used for experience concepts.
- kind: the short label (4-8 characters) naming what class of content
  this is, answering questions like "where is it" or "how is it done";
  it is a label, not the content (the content goes in `content`). One
  facet per kind per concept and at most six facets under one concept,
  so a kind has to be able to summarise.
- content: the text of the facet, at most 400 characters (the tool
  refuses a longer one). Each facet is embedded as a vector of its own,
  and what exceeds the embedder's window is not retrieved, with no
  error: long does not mean better remembered, it means the second half
  is invisible. On hitting the cap, compress (keep coordinates,
  numbers and conditions, cut throat-clearing) or split (into another
  kind, or into a new concept).
- relation: the predicate of the outgoing edge -- substitute both ends
  and read it aloud: "from <relation> to" must read as a sentence
  (occurred_at reads; location does not -- a noun cannot be read
  aloud, so the direction can only be guessed).

You are also given an INDEX of the graph -- every concept with its
layer, the kinds of facet it carries and the relations it has; the
content is read on demand with get_concept. Read it before writing:
- How to read a relation: "rel -> x" is THIS concept rel x; "rel <- x"
  is x rel this concept.
- Recognising "I have already recorded this": the test is whether the
  QUESTION this sentence answers coincides with the question some
  existing entry answers, not whether the names look alike. Where they
  coincide, fold it in with update_facet, which is a whole-text
  rewrite.
- Places are recognised by coordinate: the identity of a camp, chest or
  mine is its position, not its name (a name carrying the day number
  never collides), so compare by coordinate and extend the entry you
  find rather than creating one per day.
- The memory layer may be listed only in part. When it is, the index
  opens with a line saying how many are listed and how many there are,
  and the exception applies: search_concepts before creating a memory
  concept, because older ones were dropped and not seeing them does not
  mean they were never recorded. The experience layer is not subject to
  this: what is not listed there is genuinely absent.

Names are Chinese throughout, except slug, which is the database key.
Reuse the names you have already used: each day is an independent
session that cannot see the others, and the only place the old names are
visible is the graph index above, which lists the facet kinds and the
relation names of every concept -- scan it first. One
category carrying a different name every day is a category that does
not exist.

## 4. Skills (your own higher-level functions) -- the other output,
   parallel to textual experience

Everything above is TEXTUAL experience (create_concept / add_facet):
it records understanding but does not execute. The other kind of
output is skills: your own higher-level functions. The raw material
when you write code during the day is of two kinds: the low-level
actions and the skills that already exist, its own and other agents'.

The shape: a PARAMETERISED async function whose first argument is bot.
A low-level action is one atomic operation; a skill is a reusable unit
that arranges several low-level actions and/or other skills -- packed
into one parameterised call that tomorrow, and the other agents, can
run with await skillX(bot, ...). What it stores, and what the
signature looks like, is decided by the flow that actually ran that
day. A skill must pass the generalisation gate before anything else:

1. Signature level: a parameterised function, not a transcript of one
   run. Hard-coded coordinates, item names or counts welded into the
```

```
   body make it a memory, not a skill -- change the input and it cannot
   run. The test: can it still run with different arguments? If not,
   do not store it as a skill.
2. Applicability level: it does not depend on one world state.
   Preconditions the skill needs (furnace placed, pickaxe in hand) go
   into preconditions; do not assume they hold.

Red lines (on top of the gate): store only code that succeeded and
achieved its intent today; the code must contain an async function
whose name matches the signature (validated by the tool), first
argument bot; write the intent concretely -- retrieval matches on it,
"when smelting iron ore" is far more retrievable than "mining"; the
code you store comes from the steps that ran today -- or is a synthesis
of several of them (each step of the trajectory is marked "> generated
code: ...") -- one step, or several steps such as "mine the ore",
"return to the furnace", "smelt" and "collect the ingots" gathered into
one complete flow.

Finally: when all tool calls are done, call no more tools; output a
one-sentence summary of what you wrote (which concepts and skills you
created), and stop.
```

A.6 THE TWO SHARING CHANNELS

§2.3 separates the experience layer into two channels with different integrity requirements: textual knowledge records are rewritten by the aggregator, executable skills must arrive untouched. Figure 7 shows both on data from one persisted run: a consolidated concept assembled from several agents' wordings on the left, and one author's skill propagating verbatim into every agent's execution environment on the right.

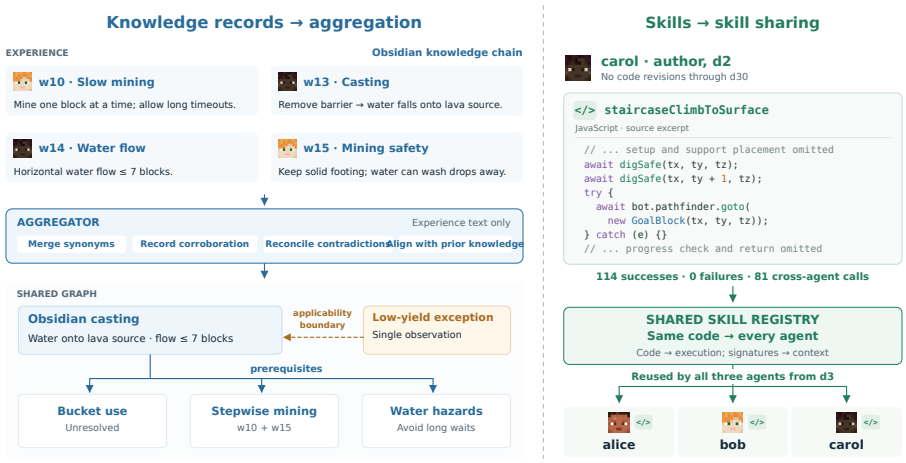

Figure 7: The two channels of knowledge sharing. **Left:** textual experience from different agents is consolidated into one shared graph. **Right:** skills travel verbatim; Carol's `staircaseClimbToSurface` (day 2) was reused by all three agents from day 3, with 114 successes and 81 cross-agent calls by day 30.

A.7 THE AGGREGATOR

Every K days, once every reflector has finished, the aggregator reads what the agents' experience layers hold and updates the shared graph (§2.3). Two guarantees of different kinds are in play here, and we state them separately. What is enforced is *access*: the aggregator is given the experience layers only, and its tools take no argument that addresses the memory layer, so no memory-layer content can be copied into the shared graph. What is requested is *content*: reflectors are instructed to phrase experience records without agent-specific coordinates and dates (Appendix A.5). The access restriction by itself does not guarantee that such details are absent from what the reflectors wrote, and we do not check that they are, so we make no claim about leakage beyond the access boundary. The one thing that does appear from the memory side is the *name* of a memory concept an experience is linked to, shown as a private anchor: readable, not retrievable, never writable into

the shared graph. The local textual-experience buffers are cleared once the window's contribution has been merged, so a private experience layer spans one window of reflection, while private memory and skill libraries are kept and the shared graph keeps growing.

What the aggregator is given. One block per agent: its experience concepts, each with its summary, its facets under their labels, and its relations, with the direction of every relation spelled out. Then a pre-screen: for each candidate, up to three concepts already in the shared graph whose similarity exceeds a threshold, offered explicitly as a hint and not a verdict. Then the labels and relation names the shared graph already uses, with counts, but not a list of what the graph contains: each window is an independent session, and whether something is already there is something the aggregator has to look up; the label list answers a different question, what this graph calls this kind of thing. The aggregator is told to work in order: read all N submissions and group them first, then search the shared graph and read what is already there before writing, then write.

The aggregator's system prompt (English translation). The prompt is short by design: it defines the role and the four duties of §2.3, states what the input looks like, and gives the representation, that is, what a concept, a facet and an edge are and what each field may hold (section 3). Everything that is a property of an operation, what a call does and what it refuses, is left to the tool descriptions, which are in the context as well and are not repeated here.

```
You are the experience-merging agent: several agents have each
finished a stretch of exploration and each distilled some experience;
you fuse them into a shared experience graph for everyone to reuse,
reading and writing that graph directly with tools.

## 1. The four things you do

1. Merge like with like: several agents distilled the same thing in
   different wordings -> store it as ONE facet: keep the clearest,
   most parameterised version and fold the others' details in.
2. Weight by consistency: when several agents INDEPENDENTLY reached
   the same experience, update_facet synthesises a more precise one,
   and "confirmed independently by several sources" goes into the
   facet text (independent reproduction is the core credibility
   signal, and it must show in the content). A claim made by one
   agent only, uncorroborated, is still admitted and stored -- give
   future windows the chance to confirm it.
3. Resolve conflicts: when two experiences contradict, do not
   force-merge by picking one. First ask whether they have different
   applicability conditions -- that is usually the truth (A holds by
   day, B fails at night). If so, synthesise one statement recording
   when it holds and when it fails (in the text, plus an edge in the
   right direction). If one side is simply wrong, correct the
   conclusion.
4. Align with what the shared graph already holds: it may already
   carry priors (textbook seeds) or experience from earlier windows.
   Where a new item matches an existing concept, fold it in; do not
   create duplicates.

## 2. What the input you are given looks like

Each agent's private graph has two layers: the memory layer (where
camp is, what happened on which day) is not given to you -- each
stretch of exploration runs in a freshly regenerated world, so
coordinates and dates mean nothing to anyone else; what you get is
the experience layer: what it distilled and believes holds for
others, with the facet text given in full. So you do not need to
"extract rules from episodes" -- the agents have done that; what you
do is turn N independently distilled experiences into one trustworthy
body of public knowledge.

Two properties of this input:
- The agents' slugs are not resolvable in your hands (get_concept
  always returns "not found"): your tools are bound to the shared
  graph only, and the agents' experience exists only as the text in
  the user message.
- A peer marked [memory] is the NAME of a memory-layer concept (e.g.
  verified_by <- day_1_death [memory]): only the name is shown, and
  it shows what supports this experience or what blocked it. It does
  not enter the shared graph -- no concept for it, no endpoint of a
  manage_edge call, and such names are not copied into facet text.

## 3. Knowledge representation: the graph structure
```

```
The shared graph is made of three things, and these fields are all you
can write:

| What            | Fields                     | You decide         |
|-----------------|----------------------------|--------------------|
| Concept         | slug / name / summary      | all three          |
| facet           | kind / content             | kind; content is the text |
| edge            | from --relation--> to      | relation           |

- slug: ASCII snake-case, unique in the graph, the only handle you
  have for get_concept / manage_edge later (a-z / 0-9 / underscore
  only).
- name: the concept's name, in Chinese. The shared graph holds common
  names only: a coordinate, "day N" or "my" in a name means an
  individual case leaked in -- rewrite it as a common name.
- summary: one sentence saying what the concept is.
- kind: the short label (4-8 characters) naming what class of content
  this is, answering questions like "how is it done" or "when does it
  fail"; it is a label, not the content (the content goes in
  `content`). One facet per kind per concept and at most six facets
  under one concept, so a kind has to be able to summarise.
- content: the text of the facet, at most 400 characters (the tool
  refuses a longer one). Each facet is embedded as a vector of its
  own, and what exceeds the embedder's window is not retrieved, with
  no error: long does not mean better remembered, it means the second
  half is invisible. On hitting the cap, compress (keep numbers and
  conditions, cut throat-clearing) or split (into another kind, or
  into a new concept).
- relation: the predicate of the outgoing edge -- substitute both
  endpoints and read it aloud: "from <relation> to" must read as a
  sentence (requires is fine; precondition is not -- a noun does not
  read as a sentence and the direction is anyone's guess; nominal
  relations submitted by agents are rewritten into predicates when
  merging).

Finally: when all tool calls are done, call no more tools; output a
one-sentence summary of which experiences you merged, and stop.
```

A.8 Retrieval: implementation and hyperparameters

This appendix completes §2.4 with the implementation and the hyperparameter values. All embedding goes through one asymmetric instruction-tuned encoder (KaLM, 896-d) (Zhao et al., 2025), over two indices keyed by slug, one for concepts and one for skills. A concept is indexed as several shards—one identity shard over its name and summary and one shard per facet—and a query scores a concept by fusing its shards back into one cosine-scale score, so that a facet can bring back the concept it belongs to and a short name cannot outvote a well-supported one. Anchoring admits a concept by the relative gate of §2.4 alone, with no absolute floor: cosine values are not comparable across encoders or corpora, whereas the relative gate is portable. On expansion, query–relation similarities are z-standardized within each entry's candidates before entering Eq. equation 1, which restores their spread without changing the ranking: every relation text shares the entry name as a prefix, so the raw similarities sit in a band in which softmax would be nearly uniform. The candidate set is the relation types actually present on the current frontier. Recall returns every facet of every node reached, which is bounded by the (concept, kind) uniqueness of Appendix A.2, and every injected block is deduplicated against the conversation so far, so that a repeated injection returns only the increment. Retrieval defaults to the experience layer, since the memory layer is already injected each morning.

Table 10 lists the values used in every run; they are identical across conditions.

A.9 Skill lifecycle and the verbatim channel

This appendix completes §2.3 on the skill side: the record a skill carries, how it is created and maintained, and how it crosses agent boundaries without ever being rewritten. The design question is the mirror image of the textual channel's: there, knowledge is prose and needs an LLM to merge it; here, knowledge is executable code, and what the world reports back is unambiguous, namely whether a call resolved or threw, so the machinery is measurement and bookkeeping, and the only LLM judgement needed is "which skill is failing and how do I fix mine". What that measurement records is *execution completion*, not semantic correctness: a call that resolves has run, which is a

Table 10: Retrieval hyperparameters, identical across all conditions. The temperature and h govern only edge sampling; anchoring is deterministic given the index.

Symbol	Value	Role
k	4	vector candidates considered for anchoring
ρ	0.80	relative similarity gate: admit within ρ of the best match
h	2	hop cap; the agent chooses $1..h$ per query
p	0.95	nucleus pruning of low-probability edges, after softmax
τ	1.0	softmax temperature over edge scores
Δ_{expand}	1.0	strength gain when the agent expands a concept
Δ_{retrieve}	0.1	strength gain when a query returns a concept

weaker claim than having achieved what the caller intended, and no stage of the pipeline checks the latter.

The record. A skill is one row in its author's library: an `intent` (the natural-language condition under which it should be used, which is the text the skill index embeds, so retrieval matches on *when to use it*, not on the code), a `signature` (what the context shows: the call form, which must begin with the function name defined in the code, since the acting agent calls by signature and a mismatch would make the skill unreachable, which the creation tool rejects), the `js_code` itself, optional `preconditions`, and a `verified` flag. Beside these, the two count pairs and the revision count of Appendix A.2: the current-version pair is the evaluation window, the all-time pair the skill's lifetime reliability. One number answers "did my fix work", the other "do I trust this slot at all".

Creation. Skills are written from programs that executed successfully that day and whose outcome the reflector reads as having achieved its intent; the creation prompt states this threshold and the reflector applies it, and no independent test re-checks it. This includes skills composed from several successful fragments of a day, in which case the composed function is not separately executed before it is registered. Skills are written by the phase-1 reflector in the first place, and by the phase-2 maintenance session when it adds one. A name collision is rejected outright: the JS function name and the slug are unique across the whole group. A near-duplicate *intent*, by contrast, is only reported back to the author as an advisory list, since whether two intents name the same job is a judgement the author makes and not a fact the tool can settle.

Measurement: the verbatim channel. The code of every verified skill is injected wholesale into the execution environment, in the same scope as the atomic API and available to every `write_code` step at zero context cost, while what enters the context is only signatures: a base block of the top-20 by cumulative success, supplemented by signatures retrieved for the current goal. Listing all signatures instead fills the context with skills unrelated to the current goal. Use is measured by the world, not inferred from text: each injected skill is wrapped so that a call either resolves (recorded as an execution completion) or throws (recorded as a failure, with the error message kept); the counts record neither whether the call achieved the caller's purpose nor whether a caller invoked the skill for the purpose its intent names. The counts are routed back each night, deterministically, to the *author's* library, so the record a maintainer sees is the skill's true usage across the whole group, and a user's failing calls adjust the author's record, not the user's.

Maintenance and retirement. The phase-2 session (Appendix A.5) reads, for each of its own skills: both count pairs, the revision count, and up to ten deduplicated error messages from the current window. Its brief is to act only on what is failing, with the fix driven by the error pattern; the prompt below states it in full. Revising the code resets the evaluation window while the lifetime totals keep accumulating. This is what makes maintenance a feedback loop rather than guesswork; editing only the intent or preconditions does not reset the window, as that is documentation, not code. A skill that cannot be fixed is *retired*, that is, demoted out of the verified set so that it is no longer injected or retrieved. Retirement is reversible by rewriting. A skill nobody ever calls is read as a retrieval failure of its intent, and is fixed by making the intent more specific ("when smelting iron ore" matches; "mining" does not). The author may revise or retire only its own skills, and the barrier structure of Appendix A.1 ensures that no agent starts a day calling a skill that was retired overnight.

Propagation. Sharing is a read-only registry over the per-agent libraries: retrieval fans out to every library, merges by similarity with each hit tagged by author and counts, and nothing is copied into a central store or ever rewritten in transit. The code that runs is the author's code, unmodified, and its revision history lives in exactly one place. The contrast with the textual channel (§2.3) is what the pair is for: prose has to be *interpreted* to be merged, so it goes through an aggregator, whereas code can be reused as it stands, so what the group accumulates about it is measurement rather than an opinion.

The skill-maintenance prompt (English translation). The phase-2 prompt, abridged to its operative rules.

```
You are the maintainer of your own skill library. Each dawn, the skills
you wrote were called by many users during the past day; now you look at
how they performed, and whether to improve or retire them.

You will see all of your own skills, one line each: cumulative
successes/failures of the current code version since its last revision;
lifetime successes/failures over all versions, never reset; how many
times the code has been revised; and real error samples from this
window, deduplicated, at most ten per skill.

Judge each skill; act on the ones with problems. A skill with only
failures, or a high failure rate, has a bug, not bad luck: rewrite it
with update_skill, reading the errors---timeouts: split the work or
request a longer budget; missing inputs: check the inventory first;
pathfinding failures: aim at a looser target; wrong blocks: re-read the
world. One you cannot fix, demote for now: a retired skill stops being
injected, and stopping the loss beats letting it burn your step budget
again tomorrow. Skills that work are left alone. A skill nobody calls
means its intent is too broad or too narrow to be retrieved---that too
is fixed by rewriting.

Touch only your own skills. Revising code resets the counters: that is
your clean evaluation window, and tomorrow's counts tell you whether the
fix worked. Do not let a bad skill linger---it is injected every day and
wastes your budget until you fix it or retire it.

When you are done calling tools, output a one-sentence summary of what
you maintained, changed, or retired, and why.
```

A.10 Relation to prior systems

§4 positions OpenKAGE against prior work argument by argument; this appendix compresses the same comparison into a single view. Table 11 collects related agent-memory and experience-sharing systems along the axes that the mechanism sections of this paper isolate: who accumulates, what the artifact of accumulation is, whether situated and transferable knowledge are separated, whether what is shared gets consolidated, whether an accumulated store is transferred to a new group, and how memory is given back at decision time. The rows are not validated under one protocol: these systems were evaluated in different environments, text games for AriGraph, coding benchmarks for GEA and Minecraft for the remaining rows, so the table compares designs rather than reported numbers.

Three rows deserve a sentence beyond the table, because the resemblance is closest and the difference easiest to miss.

AriGraph. The nearest single-agent ancestor of our graph, evaluated on text games (Anokhin et al., 2025): an agent grows a knowledge-graph world model with episodic and semantic layers, explores it locally when deciding, and revises it from interaction. OpenKAGE adopts the graph memory and the two-way split, and differs in what the split is *for*. In AriGraph both layers serve one agent's understanding; in OpenKAGE the split is the sharing boundary. The semantic layer is what may leave the agent, and the episodic layer never does. A second mechanism (aggregation, Appendix A.7) exists precisely to reconcile N agents' semantic layers into one graph. Retrieval also differs in kind: AriGraph's agent walks its own graph interactively, whereas our retrieval is intent-keyed, scored, and sampled, and runs over a graph the acting agent did not write.

Voyager. The source of our action space and of the skill artifact, and the row our skill channel most resembles. The difference that matters most is where goals come from. Voyager organizes

Table 11: Related agent-memory and experience-sharing systems. "Unit" is who accumulates; "artifact" is what accumulation produces; "memory → context" is how stored knowledge reaches the acting agent. Rows are compared on design, not on reported scores.

System	Unit	Artifact	Situated / transferable split	Shared & consolidated	Store moved to a new group	Memory → context
Voyager	1	personal skill library (code)	— (skills only)	—	— (one agent)	embedding top-k over skill descriptions
GITM / JARVIS-1	1	structured external knowledge + text memory	—	—	n.e.	retrieval over knowledge base
Optimus-1	1	hybrid graph + experience pool	—	—	n.e.	graph query for planning
MrSteve	1	What-Where-When episodic trace	episodic only	—	n.e.	spatiotemporal retrieval
AriGraph	1	KG world model, episodic + semantic	✓ (two layers, one agent)	—	n.e.	local subgraph exploration
ADAM	1	causal graph	—	—	n.e.	graph-guided reasoning
MARS	1	induced rule set	—	—	n.e.	rules in prompt
GEA	group	pooled evolutionary traces	—	pooled, not consolidated	n.a.	reflector reads the pool
OpenKAGE	group	two-layer graphs + skills; shared consolidated graph	✓ (memory stays private)	✓ (semantic aggregation)	✓ (§3.4)	intent-aware subgraph retrieval

✓ = demonstrated for the object named in the cell; — = not applicable to the system as designed; n.e. = not evaluated in the published description; n.a. = a transfer result is reported for a different object, so the column does not apply. GEA reports transfer of the evolved framework across coding models (Weng et al., 2026); our transfer experiment moves an accumulated knowledge store, namely the shared graph and the skill library, into a freshly initialized group, which is a different object and is not comparable with that result.

exploration around tasks proposed by a dedicated automatic curriculum module; OpenKAGE integrates goal generation into the acting agent's ongoing interaction loop, in which the agent selects and revises its own goals using its current state and its accumulated knowledge, with no separate curriculum module. The distinction concerns how goals are generated and how they govern interaction, not whether human intervention is required: Voyager's curriculum is automatic, and the system is evaluated on open-ended exploration, tech-tree mastery and map coverage. Taking the group as given, three further differences follow. Skills are shared through a registry rather than living in one library, so a skill's record accumulates every agent's calls, not just its author's; maintenance is driven by that cross-agent record (Appendix A.9); and skills are the *second* channel, alongside the consolidated textual graph, which has no analogue in Voyager. The single-run ablations of Appendix B.5 indicate that the two channels are not substitutes, though one run per condition cannot establish it. The action space also starts lower: everything in scope is an atomic Mineflayer call, where Voyager's authors supply hand-written capability primitives from step one. Strong primitives mask what we measure: an agent that can call `smeltItem` can smelt before it has learned anything, so part of any growth we attribute to accumulated knowledge would instead be the primitives working. With the API atomic, a composite capability is something the group has to acquire, rather than something available from the start.

GEA. The nearest group-level system, evaluated on coding benchmarks (Weng et al., 2026): the group is an evolutionary unit whose members' traces are pooled and read back by a reflector. OpenKAGE also lets the group read what every member produced, but the shared artifact is a graph whose records are consolidated rather than a set of parallel traces, and skills travel on a separate channel. The two also differ in what they report as transfer, as the note under Table 11 states. We do not compare numbers with GEA: its evaluation protocol and ours differ in environment, control interface and budget (Appendix B.3).

B EXPERIMENTS

B.1 REPRODUCIBILITY CONFIGURATION

Table 12 lists the settings that determine how a run behaves and that a reader needs in order to rebuild one. Values are identical in every reported condition unless a row says otherwise, and the configuration file released with the code carries every one of them. Parameters that are already tabulated elsewhere are not repeated here: retrieval hyperparameters in Table 10 (Appendix A.8), memory injection budgets in Table 5 (Appendix A.2), the action API and observation fields in Appendix A.3, and the personas in Table 8 (Appendix A.4).

Table 12: Reproducibility configuration. Values are identical in every reported condition unless a row says otherwise; the configuration file released with the code carries every value, and each run records the configuration it used in its own summary.

Group	Value
Backbone	DeepSeek-V4.1-Flash (DS-V4.1F) and GLM-5.3 (GLM); transfer target GLM-5.3-Flash; one backbone serves all three roles of a run
Temperature	0.3 planner, 0.7 reflector, 0.7 aggregator
Top-p	not set
Reasoning	`low` for all three roles
Output cap	131,072 tokens per planner step, 65,536 per reflector and aggregator step, 8,192 for the night note
Step cap	299 tool calls per day, 99 per reflector session, 99 per aggregation
Context	no truncation threshold: a day's conversation is carried in full and discarded at dawn
Encoder	KaLM, asymmetric instruction-tuned, 896-d (Zhao et al., 2025), over two indices, one for concepts and one for skills
Query / document	an instruction prefix is prepended to queries only, never to indexed text
Truncation	512-token encoder window; shards cut at 512 characters; facet content above 400 characters is rejected by the tool, not truncated
Game	Minecraft Java Edition 1.21.11 (Fabric server), Mineflayer 4.37.1 client
World	seed 7 for every reported run, natural spawn point of that seed, difficulty hard, `keepInventory` on, one day–night cycle = 24,000 ticks, game time frozen during deliberation and reflection, game clock at $2.5\times$
Agents	$N = 3$ (Alice, Bob, Carol), each in a separate server instance generated from the same seed; $N = 5$ and $N = 8$ add the next personas in a fixed order
Horizon	30 nights; aggregation every $K = 2$ nights, i.e. 15 windows

Two protocol choices are not configuration values and are stated here because they affect how the numbers should be read. First, across backbones only milestone first-reach times are compared and not final values, because the two backbones differ in absolute survival by more than the effect we measure. Second, the 30-day horizon is an evaluation budget, not a stopping criterion of the framework: agents keep proposing goals, accumulating experience and maintaining their knowledge with no predefined terminal task, and nothing in the loop asks them to stop at day 30. The transfer experiment of §3.4 uses GLM-5.3-Flash, a smaller model in the same family as the GLM-5.3 backbone, as its target backbone; its source stores are the shared graph and skill library of one completed DeepSeek-V4.1-Flash run at day 30, the non-transfer control is initialized from empty stores, and both then follow the identical 30-day protocol.

B.2 METRIC DEFINITIONS

All metrics come from the structured snapshots recorded at the end of each day. Deaths is cumulative per agent, averaged over the group. We also recorded XP per life (final-day cumulative XP divided by deaths $+1$) but do not report it: its run-to-run spread is several times the gaps we measure, so it carries no signal at this sample size. Eq-max is the highest four-slot armor score reached on any day (16 is the ceiling). Tool era is the highest technology tier reached in the agent's equipment, iron or diamond in our runs, and read from the tool tier itself rather than from obtaining obsidian, which is a milestone with its own column; both are reported with the number of agents that reached them. Adv's is the number of distinct advancements unlocked, pooled over the five advancement trees

(Minecraft's built-in advancement system; `https://minecraft.wiki/w/Advancement`). Obsidian is the day of first obtainment, averaged over the agents that obtained it; Tool era carries the day of first reaching that tier, and Nether the day of first entry, each averaged over the agents that reached it and reported with that count in parentheses. Table 3 reads two of the same milestones by a different convention: because it has one run per condition, the day-of-first-reach varies more across the three agents of a single run than across conditions, so there Obsidian and Nether each report the number of agents out of the three that completed the milestone at all by the final day, namely obtaining obsidian, entering the Nether. The two conventions are not mixed within a table. Diamond is dropped from that table: it is reached by almost every agent in every condition, 8 of the 9 Full agents included, so the column cannot separate them. Deaths is dropped for the same reason from the other side—its spread across runs of one condition is wider than the gaps between conditions—and the ablation section reports the numbers it needs in text rather than in a column. Starve counts the deaths of a cell—or of a run, in Table 13—whose recorded cause was starvation. *Basic structures* (Table 2) is likewise computed from the end state: per agent, the crafting tables, chests, furnaces and beds left standing in the final world save, averaged over the 9 agents of a cell.

B.3 Baselines

The same-backbone baseline The baseline is OpenKAGE with the knowledge machinery removed and nothing else changed: same backbone, prompts, tool set, night count, group size, world seed, and day–night protocol. The baseline agent keeps the planner of §2.1. It still sets the day's goal and ordered steps itself, still acts by writing one short program per step over the same atomic API, and still reasons over the observations and feedback that come back. What it loses is persistence. No local graph is written, no shared graph is consolidated, no skill is registered or retrieved, and no memory slice is injected in the morning; because in OpenKAGE the reflector is the process that maintains an agent's graph and emits the one line that crosses the day boundary, removing the graphs also removes the nightly reflection, so no text at all is carried from one day to the next. Retrieval has nothing to retrieve, and the planner meets each day with the model, the observations, and nothing else. The residue is exactly the standard single-agent LLM-agent stack (Yao et al., 2023; Shinn et al., 2023; Significant Gravitas, 2023), in the same family as the planner-only control of Wang et al. (2024); we do not claim the two match item by item in configuration. Relative to it, OpenKAGE adds persistent memory, nightly reflection, the skill channel, intent-aware retrieval and group-level consolidation. The gap between the two rows of a backbone is therefore the combined effect of that knowledge stack under a matched agent–world interface: it is not an estimate of what sharing alone contributes, and it does not establish superiority over other memory-equipped agents.

Scope of the comparison We use a planner-only control to measure the combined effect of persistent knowledge, reflection, retrieval and sharing under a matched agent–world interface. This comparison does not isolate the contribution of group-level consolidation, nor does it establish superiority over existing memory-equipped agents: differences in control interfaces and evaluation budgets prevent direct numerical comparison with published results. The conditions of Appendix B.5 are internal controls of the same kind; in particular $-$S, which removes both sharing channels while leaving each agent its own graph, skills and reflection, is a no-sharing control inside OpenKAGE rather than a stand-in for a strong single-agent system. What makes published numbers incomparable with ours is the protocol rather than the effort of obtaining them: the systems we would compare against are scored on objectives supplied to them, MineDojo task suites (Fan et al., 2022), item-collection tasks (Zhu et al., 2023), multi-task schedules (Wang et al., 2025; Li et al., 2024) or the proposals of an automatic curriculum (Wang et al., 2024; Yu & Lu, 2025), whereas OpenKAGE runs one continuous 30-day loop with no externally assigned task and no separate curriculum module; and several of them start with what our protocol treats as the thing to be acquired, namely external knowledge (Fan et al., 2022; Baker et al., 2022; Zhu et al., 2023; Wang et al., 2025) or hand-written capability primitives (Wang et al., 2024).

What *can* be held fixed across a pair is the mechanism, so that is the comparison we make; Table 11 (Appendix A.10) places the prior systems along the axes the mechanism sections isolate.

B.4 FULL MAIN RESULTS

Each row of Table 1 pools three runs; this appendix lists those runs individually. Runs of the same backbone–method cell share the world seed and differ only in group identity, so the spread across the three rows of a cell is that cell's run-to-run spread.

Table 13: Per-run values behind Table 1: one row per run of three agents. Deaths and Adv's are means over the three agents of that run; Starve is a total for the run; Eq-max is a score with the count of agents that reached the ceiling, and Tool era, Obsidian and Nether report the mean first-attainment day over the agents that attained them, with that count. A dash indicates that no agent reached the milestone within 30 game days.

Backbone	Run	Deaths ↓	Starve ↓	Eq-max ↑	Tool era ↑	Obsidian ↓	Nether ↓	Adv's ↑
DS-V4.1F	baseline-1	22.0	2	16 (3/3)	*dmd.* (14.3 day, 3/3)	—	—	12.3
	baseline-2	22.3	5	16 (2/3)	*dmd.* (7.0 day, 3/3)	—	—	12.3
	baseline-3	26.7	6	16 (1/3)	*dmd.* (10.3 day, 3/3)	—	—	12.3
	OpenKAGE-1	14.0	2	16 (2/3)	*dmd.* (6.0 day, 3/3)	24 day (2/3)	—	13.3
	OpenKAGE-2	29.0	0	16 (1/3)	*dmd.* (10.3 day, 3/3)	17 day (2/3)	21.0 day (2/3)	13.7
	OpenKAGE-3	23.0	2	16 (1/3)	*dmd.* (11.5 day, 2/3)	25.5 day (2/3)	30.0 day (1/3)	12.3
GLM-5.3	baseline-1	42.7	2	13 (1/3)	*dmd.* (6.7 day, 3/3)	—	—	11.0
	baseline-2	41.7	2	14 (1/3)	*dmd.* (8.0 day, 3/3)	—	—	10.3
	baseline-3	35.3	2	14 (2/3)	*dmd.* (7.3 day, 3/3)	—	—	11.0
	OpenKAGE-1	22.7	1	16 (1/3)	*dmd.* (12.5 day, 2/3)	22.5 day (2/3)	—	10.7
	OpenKAGE-2	29.0	1	16 (1/3)	*dmd.* (9.7 day, 3/3)	24 day (1/3)	—	11.3
	OpenKAGE-3	21.7	1	12 (3/3)	*dmd.* (12.0 day, 3/3)	24 day (1/3)	28.0 day (1/3)	11.3

B.5 ABLATION CONDITIONS

Each condition of §3.3 removes one mechanism; conditions are named by what they remove. What each removal leaves intact matters for reading the gaps, so it is stated per condition. What each removal leaves intact matters for reading the gaps, so it is stated per condition.

- $-$S**. No sharing (individual memory).** Private graphs and subgraph retrieval stay; both sharing channels go: no aggregation (the shared graph stays empty, retrieval sees only the agent's own graph) and no cross-agent skill reuse (skills are read from the author's library only).
- $-$R**. Flat retrieval (RAG-style).** The full knowledge pipeline stays; retrieval degrades to a top-k embedding-similarity flat list, with no edge expansion and no subgraph structure. Only retrieval varies, though the knowledge agents subsequently act on may diverge as they take different actions.
- $-$M**. No memory layer.** The experience layer, aggregation, the shared graph, and skills all keep operating; reflection no longer writes memory concepts and the morning injection disappears. Unlike the others, this one varies *which layer* stores knowledge, not what is stored.
- $-$E**. No experience layer.** Reflection writes only to the memory layer: no textual experience record is written and the skill channel is off, so no skill is created, maintained, injected or reused across agents. The aggregator receives nothing, the shared graph stays empty, and experience retrieval returns nothing.
- **B. No graph (bare model).** Removes every graph, private and shared: no memory, no aggregation, and no skills. This is the no-graph control of the main experiment, here also the floor of the suite.

Two caveats follow from the design itself. First, $-$S and $-$E both leave the shared graph empty; they differ in whether the agent's private experience is still written. It is under $-$S but not under $-$E, so their gaps overlap by construction and must not be read as additive independent contributions. Second, each condition is one run of three agents, and the spread across agents within a condition is often comparable to the gaps, so a condition whose gap is smaller than that spread needs a second run before it can be interpreted.

Table 14 reports each condition on the readouts of Table 1. Table 3 in the main text keeps only the two milestones on which the conditions separate; the columns here show what each removal costs

on survival and on advancement progress. Full and B pool the DS runs of Table 13, so only the four one-run conditions are broken out.

Table 14: Ablation conditions on the readouts of Table 1 (DeepSeek-V4.1-Flash, 30 game days). One row per condition, each condition being a single run of three agents. Deaths and Adv's are means over the three agents of that run; Starve is a total for the run; Eq-max is a score with the count of agents that reached the ceiling, and Tool era, Obsidian and Nether report the mean first-attainment day over the agents that attained them, with that count.

Cond.	Deaths ↓	Starve ↓	Eq-max ↑	Tool era ↑	Obsidian ↓	Nether ↓	Adv's ↑
−S	21.0	0	16 (1/3)	*dmd.* (6.7 day, 3/3)	25 day (1/3)	—	13.0
−R	28.0	1	16 (2/3)	*dmd.* (6.3 day, 3/3)	24 day (1/3)	29.0 day (1/3)	12.3
−M	23.0	1	16 (1/3)	*dmd.* (11.7 day, 3/3)	—	—	11.3
−E	17.0	2	14 (1/3)	*obsid.* (26.0 day, 1/3)	26.0 day (1/3)	—	12.0

B.6 Scaling: long-horizon accumulation and group size

Sections 3.2–3.4 show that OpenKAGE reaches milestones the baseline does not reach, and that the knowledge it accumulates transfers to a fresh group on another backbone. Two questions remain. Does per-agent progress grow when the group grows? Does progress keep accumulating once the horizon is doubled? The following runs examine both axes, group size and time horizon; they are supplementary and single-run where stated, so they support observations rather than conclusions.

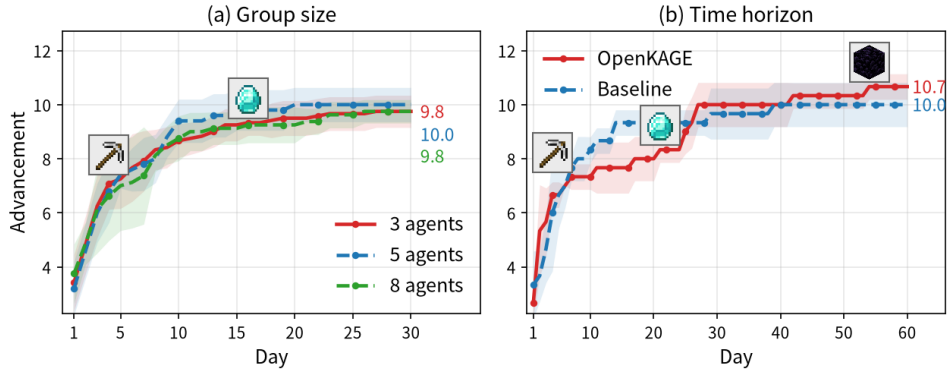

Figure 8: Scaling in group size and time horizon (GLM-5.3-Flash). Mean advancements unlocked per agent; bands are ±1 sd across agents. (a) Group size: N=3, 5 and 8 agents, 30 game days each. (b) Time horizon: OpenKAGE and the baseline over 60 game days at N=3.

Similar endpoints across evaluated group sizes. Groups of N=3, 5 and 8 agents running 30 game days each end at 9.8, 10.0 and 9.8 advancements per agent; the gaps between them are smaller than the spread across the agents of any one group (Figure 8(a)).

Time scale: a longer horizon reaches the next milestone. At N=3, extending the protocol from 30 to 60 game days takes OpenKAGE from 9.8 to 10.7 advancements per agent, and reaches obsidian on day 54 (1 of 3 agents). The baseline, given the same 60 days, stalls at a bottleneck from day 40 on, and no agent reaches obsidian (Figure 8(b)).