STRUCTUREDCACHE: HIERARCHICAL CACHE-COHERENT EXECUTION FOR EMBODIED MULTI-AGENT COLLABORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Embodied multi-agent collaboration offers a promising approach to solving complex tasks in shared and dynamic environments by enabling multiple agents to coordinate their capabilities and actions. However, efficiently maintaining consistent task states remains challenging because independently acting agents continuously modify the shared environment, which can cause actions to be executed based on outdated task states. Modern CPUs face a related challenge when multiple cores access and update shared data, using caching for efficient data access and cache coherence mechanisms to maintain consistency across cores. Inspired by this, we propose **StructuredCache**, a cache-coherent execution framework for embodied multi-agent collaboration. StructuredCache converts agent-generated actions into structured instructions that specify operation semantics, state dependencies, and execution constraints. It organizes task states through a hierarchical cache and maintains state consistency through pre-execution state validation and post-execution outcome verification. Verified state updates are then committed, while outdated task states are invalidated to support subsequent execution. We evaluate StructuredCache on Minecraft-based crafting, cooking, and construction tasks across multiple large language models and collaboration scales. Experimental results show that StructuredCache consistently improves task success and reduces completion steps, demonstrating its effectiveness in enabling reliable and efficient embodied multi-agent collaboration.

1 INTRODUCTION

Embodied multi-agent collaboration enables multiple autonomous agents to perceive, reason, and act in a shared environment toward common objectives (Zhang et al., 2025b; Chang et al., 2025; Zu et al., 2025; Qin et al., 2025; Wu et al., 2025). By coordinating their capabilities and actions, multiple agents can divide complex objectives into subtasks, combine complementary information and skills, and perform sensing, planning, and execution in parallel (Dong et al., 2024; Kannan et al., 2024; Zhang et al., 2025c; Wu et al., 2024; Hong et al., 2024; Qian et al., 2024). These capabilities make multi-agent systems well suited to complex embodied tasks that require continuous interaction with dynamic environments (Dong et al., 2026; White et al., 2025). However, effective collaboration depends not only on how agents communicate and plan, but also on whether they can maintain consistent task states as the shared environment evolves (Liu et al., 2025b; Xiao et al., 2025; Omidshafiei et al., 2017).

Existing embodied multi-agent frameworks improve collaboration through communication, cooperative planning, shared memory, and task allocation (Zhang et al., 2024; Hu et al., 2024; Liu et al., 2025a; Zu et al., 2025; Zhang et al., 2025b; Tan et al., 2025; Chen et al., 2025; Qin et al., 2025; Zhang et al., 2025a). These approaches provide agents with richer information and stronger coordination capabilities, but typically lack explicit mechanisms for maintaining task-state consistency during execution. In a shared dynamic environment, task states may be continuously modified by different agents, while other agents may continue to rely on previously obtained states. As a result, actions can be generated or executed based on outdated task states, reducing the reliability of collaborative execution.

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.

Modern CPUs face a related challenge when multiple cores access and update shared data (Hennessy & Patterson, 2017; Nagarajan et al., 2020). Caching enables efficient access to frequently used data, while cache coherence mechanisms maintain consistency when shared data are modified across cores (Lenoski et al., 1990). This provides a natural perspective for embodied multi-agent collaboration, where multiple agents independently operate on task states associated with the same evolving environment.

Inspired by modern CPU systems, we propose **StructuredCache**, a cache-coherent execution framework for embodied multi-agent collaboration. StructuredCache converts agent-generated actions into structured instructions that explicitly specify operation semantics, state dependencies, and execution constraints, providing a structured interface between agent decision making and task-state management. Task states are organized through a hierarchical cache according to their access scope and reuse patterns for efficient state access. Based on this organization, cache-coherent execution maintains task-state consistency as multiple agents interact with and update the shared environment. Together, these designs provide efficient and consistent task-state management for embodied multi-agent collaboration. Extensive experiments on Minecraft-based crafting, cooking, and construction tasks across multiple LLM backbones and collaboration scales show that StructuredCache consistently improves task success and reduces completion steps. Our main contributions are summarized as follows:

- We propose **StructuredCache**, a CPU-inspired cache-coherent execution framework for embodied multi-agent collaboration.
- We introduce structured instructions and a hierarchical cache to explicitly encode state dependencies and efficiently organize task states.
- We introduce cache-coherent execution, which validates task states, synchronizes verified updates, and maintains task-state consistency during collaborative execution.

2 Related Work

2.1 Minecraft for Embodied AI Research

Minecraft is a widely used platform for open-world embodied AI, supporting environments such as MineRL (Guss et al., 2019) and MineDojo (Fan et al., 2022). Its procedural environments, rich crafting mechanics, and long-horizon tasks pose substantial challenges for perception, planning, and control. Early studies focused on improving task generalization and enabling agents to execute hierarchical skills (Guss et al., 2019; Fan et al., 2022; Wang et al., 2024). Meanwhile, the MineRL competitions provided standardized benchmarks for evaluating sample-efficient learning from human demonstrations (Guss et al., 2021; Kanervisto et al., 2022). Recent works leverage large-scale video and text data to build generalist Minecraft agents (Baker et al., 2022; Fan et al., 2022; Lifshitz et al., 2023). VPT learns policies from gameplay videos (Baker et al., 2022), MineCLIP provides language-aligned reward signals (Fan et al., 2022), and Steve-1 enables instruction following (Lifshitz et al., 2023). LLM-based agents such as Voyager (Wang et al., 2024), DEPS (Wang et al., 2023), and GITM (Zhu et al., 2023) further improve long-horizon planning through memory and world knowledge (Dong et al., 2024; Packer et al., 2023; Zhong et al., 2024; Shinn et al., 2023; Park et al., 2023). MINDcraft (White et al., 2025) extends this direction to multi-agent collaboration and is therefore adopted as our experimental platform.

2.2 Multi-Agent Collaboration in Embodied AI

Multi-agent embodied AI studies how agents coordinate perception, reasoning, and action in collaborative tasks (Wu et al., 2025; Chen et al., 2024; Li et al., 2023). Early work focuses on cooperative navigation, assistance, and object manipulation. Watch-And-Help requires an agent to infer and assist a partner's goal (Puig et al., 2021), while CoMON explores emergent communication in collaborative navigation (Patel et al., 2021). The ThreeDWorld Transport Challenge and its extensions examine coordination in object search and transportation (Gan et al., 2022; Zhang et al., 2024), and PARTNR evaluates collaboration under spatial, temporal, and capability constraints (Chang et al., 2025). Large language models further enable natural-language communication and planning among embodied agents (White et al., 2025; Kannan et al., 2024; Driess et al., 2023; Huang et al., 2023;

Ichter et al., 2023; Liang et al., 2023; Yao et al., 2023; Sumers et al., 2024). RoCo supports collaborative reasoning through multi-round dialogue (Mandi et al., 2024), while CoELA provides a modular framework for decentralized cooperation (Zhang et al., 2024). Later methods organize agents through hierarchical teams (Guo et al., 2026), shared and adaptive plans (Liu et al., 2025a), collaborative search (Zu et al., 2025), and embodiment-aware task assignment (Chen et al., 2025; Zhang et al., 2025c). Despite these advances, existing studies mainly focus on high-level coordination and planning, while the stale-state problem, where agents act on outdated local or shared information, remains insufficiently addressed.

3 METHOD

Embodied multi-agent collaboration requires agents to maintain consistent task states while interacting with a shared and dynamically changing environment. As different agents continuously update the environment, previously obtained task states may become outdated, making task-state consistency difficult to maintain during collaborative execution. Inspired by modern CPU systems, we propose **StructuredCache**, a cache-coherent execution framework for maintaining consistent task states throughout multi-agent collaboration. As illustrated in Figure 1, StructuredCache consists of three main components: structured instruction representation, hierarchical cache, and cache-coherent execution. First, each supported agent-generated action is converted into a structured instruction that specifies its operation semantics, state dependencies, and execution constraints, providing an explicit interface between agent decisions and task states. Second, StructuredCache organizes task states through a hierarchical cache according to their access scope and reuse patterns, enabling efficient retrieval of instruction-relevant states. Finally, cache-coherent execution validates the required task states before execution and verifies state changes afterward. Verified updates are committed to the cache hierarchy, while outdated task states are invalidated. Together, these components maintain task-state consistency throughout collaborative execution.

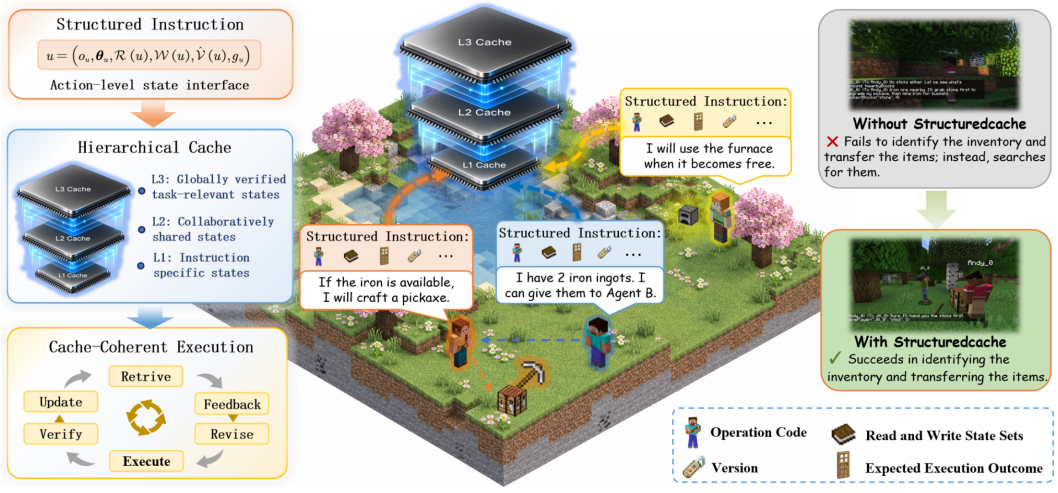


Figure 1: Overview of StructuredCache for embodied multi-agent collaboration. Given an agent-generated action, StructuredCache converts it into a structured instruction, organizes and retrieves task states through a hierarchical cache, and performs cache-coherent execution to maintain task-state consistency.

3.1 STRUCTURED-INSTRUCTION REPRESENTATION

Let $a_{i,t}$ denote the action generated by agent i at decision round t. StructuredCache converts each action into a structured instruction $u_{i,t}$:

$$a_{i,t} \rightarrow u_{i,t}, \qquad u = \left(o_u, \boldsymbol{\theta}_u, \mathcal{R}(u), \mathcal{W}(u), \hat{\mathcal{V}}(u), g_u\right). \tag{1}$$

The o_u identifies the operation type, $\boldsymbol{\theta}_u$ contains the corresponding operands, $\mathcal{R}(u)$ and $\mathcal{W}(u)$ denote the read and write state sets, $\hat{\mathcal{V}}(u)$ records the versions of referenced states, and g_u specifies the expected execution outcome. The complete set of states accessed by the instruction is defined as

$$\mathcal{A}(u) = \mathcal{R}(u) \cup \mathcal{W}(u). \tag{2}$$

The operation type specifies executable actions, such as collecting, crafting, transferring, smelting, searching, and construction-related block placement or clearing. The operands specify the relevant item, quantity, agent, or position. This representation exposes the states involved in each action, enabling StructuredCache to retrieve and validate the required task states before execution.

3.2 Hierarchical Cache

Let $\mathcal{Z}$ denote the set of task-relevant state units. A state unit $z \in \mathcal{Z}$ may describe an agent inventory entry, a resource, an object, a tool, a shared facility, a spatial region, or an item handover. Task states involved in multi-agent collaboration differ in their access frequency, sharing scope, and reuse patterns. Following the hierarchical cache design of modern CPUs, StructuredCache organizes task states into three cache levels:

$$\begin{aligned} &\text{L1}: \quad \text{States immediately required by the current agent,} \\ &\text{L2}: \quad \text{States repeatedly shared during multi-agent collaboration,} \\ &\text{L3}: \quad \text{Globally verified task-relevant states.} \end{aligned} \tag{3}$$

To identify states retained in the shared L2 cache, StructuredCache updates each state's collaboration heat at cache event k:

$$h_{k+1}(z) = \beta h_k(z) + n_r^k(z) + \lambda_w n_w^k(z) + \lambda_a \left(m_k(z) - 1\right) + \lambda_c c_k(z), \tag{4}$$

where $0 < \beta < 1$ controls historical heat decay, $n_r^k(z)$ and $n_w^k(z)$ denote the numbers of recent reads and writes, $m_k(z)$ denotes the number of distinct agents accessing z within a recent window, and $c_k(z)$ indicates whether z is involved in item handover, facility occupation, or spatial conflict. The coefficients λ_w, λ_a, and λ_c control the contributions of state modification, cross-agent access, and coordination-related conflicts, respectively. The cache priority of state z under instruction u is then defined as

$$\pi_u(z) = \begin{cases} 3, & z \in \mathcal{A}(u), \\ 2, & z \notin \mathcal{A}(u) \ \wedge h_k(z) \geq \delta_2, \\ 1, & \text{otherwise}, \end{cases} \tag{5}$$

where a larger $\pi_u(z)$ indicates a higher caching priority and δ_2 is the admission threshold of the shared L2 cache.

Each L1 entry stores $(s_i^{\text{L1}}(z), v_i^{\text{L1}}(z), b_i^{\text{L1}}(z), a_i^{\text{L1}}(z))$, where s, v, b, and a are its value, version, validity bit, and last access time. L2 entries also store heat $h_k(z)$; L3 stores verified values and versions. Write ownership $\ell(z)$ is tracked separately. Let $\mathcal{C}_i^{\text{L1}}$ and $\mathcal{C}^{\text{L2}}$ denote agent i's private L1 and the shared L2, respectively. Their capacities satisfy $|\mathcal{C}_i^{\text{L1}}| \leq B_1$ and $|\mathcal{C}^{\text{L2}}| \leq B_2$ with $B_1 < B_2$, where B denotes the cache capacity, whereas L3 keeps all environment-verified task states.

When a new state must be inserted into a full L1 cache, an invalid entry is replaced first. If no invalid entry is available, the least recently accessed state that is not required by the current instruction is selected:

$$z_{\text{rep}}^{\text{L1}} = \arg \min_{z \in \mathcal{E}\left(\mathcal{C}_i^{\text{L1}}, u\right)} a_i^{\text{L1}}(z), \tag{6}$$

where $\mathcal{E}(\mathcal{C}_i^{\text{L1}}, u)$ excludes states in $\mathcal{A}(u)$. When the shared L2 cache is full, an invalid entry is also replaced first. Otherwise, StructuredCache replaces the state with the lowest collaboration heat and uses access recency to break ties:

$$z_{\text{rep}}^{\text{L2}} = \arg \min_{z \in \mathcal{E}(\mathcal{C}^{\text{L2}}, u)} \left(h_k(z), a^{\text{L2}}(z)\right), \tag{7}$$

where $\mathcal{E}(\mathcal{C}^{\text{L2}}, u)$ excludes states protected by the current instruction and the minimization is lexicographic.

For valid L1 and L2 entries, StructuredCache maintains

$$b_i^{\mathrm{L1}}(z) = 1 \quad \Longrightarrow \quad v_i^{\mathrm{L1}}(z) = v^{\mathrm{L3}}(z), \tag{8}$$

and

$$b^{\mathrm{L2}}(z) = 1 \quad \Longrightarrow \quad v^{\mathrm{L2}}(z) = v^{\mathrm{L3}}(z). \tag{9}$$

An entry is used only when valid and version-matched to L3. Given an instruction u assigned to agent i, each state $z \in \mathcal{A}(u)$ is retrieved by probing the hierarchy in order:

$$\mathrm{Read}(i, u, z) = \begin{cases} \mathcal{C}_i^{\mathrm{L1}}(z), & b_i^{\mathrm{L1}}(z) = 1 \wedge v_i^{\mathrm{L1}}(z) = v^{\mathrm{L3}}(z), \\ \mathcal{C}^{\mathrm{L2}}(z), & b^{\mathrm{L2}}(z) = 1 \wedge v^{\mathrm{L2}}(z) = v^{\mathrm{L3}}(z), \\ \mathcal{C}^{\mathrm{L3}}(z), & \text{otherwise.} \end{cases} \tag{10}$$

Here, $\mathrm{Read}(i, u, z)$ denotes the cache lookup operation that retrieves state z required by instruction u for agent i. If L3 lacks a verified entry, the lookup fails and the instruction is gated. Whenever a state is accessed, its access time and collaboration heat are updated.

The retrieved state is always promoted to the executing agent's L1 cache:

$$\mathcal{C}_i^{\mathrm{L1}}(z) \leftarrow (s^*(z), v^*(z), 1, t)\,, \qquad z \in \mathcal{A}(u), \tag{11}$$

where $s^*(z)$ and $v^*(z)$ are returned by $\mathrm{Read}(i, u, z)$. If the accessed state satisfies the L2 admission condition, it is also inserted into or refreshed in the shared L2 cache:

$$\mathcal{C}^{\mathrm{L2}}(z) \leftarrow (s^*(z), v^*(z), 1, h_k(z), t)\,, \qquad \mathrm{Admit}_{\mathrm{L2}}(z) = 1. \tag{12}$$

After execution, states that remain highly reused are retained in L2, while states with decreasing collaboration importance become eligible for replacement.

3.3 Cache-Coherent Execution

StructuredCache enforces cache coherence through a closed loop around each agent generated action. At decision round t, agent i first proposes a candidate action $a_{i,t}^{\mathrm{cand}}$. The proposed action is then converted into an instruction $u_{i,t}$, and each state involved in $\mathcal{A}(u_{i,t}) = \mathcal{R}(u_{i,t}) \cup \mathcal{W}(u_{i,t})$ is resolved sequentially through L1, L2, and L3. The resolved states are exposed as instruction scoped feedback:

$$\mathcal{F}_{i,t}(u_{i,t}) = \left(\mathcal{F}_{i,t}^{\mathrm{L1}}, \mathcal{F}_{i,t}^{\mathrm{ver}}, \mathcal{F}_{i,t}^{\mathrm{rec}}\right), \tag{13}$$

where $\mathcal{F}_{i,t}^{\mathrm{L1}}$ contains L1-derived local facts, $\mathcal{F}_{i,t}^{\mathrm{ver}}$ contains L2/L3-supported verified facts, and $\mathcal{F}_{i,t}^{\mathrm{rec}}$ contains active execution receipts. The agent revises its action as

$$a_{i,t}^{\mathrm{rev}} = \pi_i\left(h_{i,t}, a_{i,t}^{\mathrm{cand}}, \mathcal{F}_{i,t}(u_{i,t})\right), \tag{14}$$

where $h_{i,t}$ denotes its local observation and dialogue history. The revised action is re-encoded as $\tilde{u}_{i,t}$. Before physical execution, StructuredCache applies

$$G(i, \tilde{u}_{i,t}) = G_{\mathrm{coh}}(i, \tilde{u}_{i,t}) \cdot G_{\mathrm{rec}}(i, \tilde{u}_{i,t}). \tag{15}$$

Here, $G_{\mathrm{coh}} = 1$ only if all states in $\mathcal{A}(\tilde{u}_{i,t})$ can be resolved from the hierarchy, their resolved versions match L3, $|\mathcal{A}(\tilde{u}_{i,t})| \le B_1$, and no state in $\mathcal{A}(\tilde{u}_{i,t})$ is held for writing by another active instruction. G_{rec} blocks unresolved repeated actions when the verified states required for retry have not changed. If $G(i, \tilde{u}_{i,t}) = 0$, the instruction is returned as feedback without physical execution; otherwise, it is issued to the environment.

Let e_t and e_{t+1} be the observations immediately before and after execution, and let $\phi(\cdot)$ extract task-relevant states. StructuredCache verifies the instruction outcome by

$$y(\tilde{u}_{i,t}) = \mathbb{I}\left[g_{\tilde{u}}\left(\phi(e_t), \phi(e_{t+1})\right) = 1\right]. \tag{16}$$

Here, $y(\tilde{u}_{i,t}) = 1$ means that the expected outcome is verified. For an item transfer, $g_{\tilde{u}}$ checks both the sender's inventory decrease and the recipient's inventory increase. Environment-observed state changes are synchronized to L3 regardless of whether the instruction achieves its expected outcome. For each task-relevant state z whose observed value changes, StructuredCache assigns a new version from a global counter V:

$$V \leftarrow V + 1, \qquad s_{t+1}^{\mathrm{L3}}(z) \leftarrow \phi(e_{t+1}, z), \qquad v_{t+1}^{\mathrm{L3}}(z) \leftarrow V. \tag{17}$$

The update invalidates stale cached copies. If z belongs to the current instruction's access set, it also refreshes the executing agent's L1 and, when admitted, the shared L2.

If $y(\tilde{u}_{i,t}) = 0$, StructuredCache re-observes affected states and creates an execution receipt

$$\rho = \left(\mathrm{sig}(\tilde{u}_{i,t}), r_\rho, \mathcal{K}_\rho, \bar{\mathcal{V}}_\rho\right), \tag{18}$$

where $\mathrm{sig}(\tilde{u}_{i,t})$ is the action signature formed from the agent identity, operation code, and canonicalized operands; r_ρ is the failure reason, $\mathcal{K}_\rho$ is the set of verified states whose change permits reconsideration, and $\bar{\mathcal{V}}_\rho$ records their versions when the receipt is created. Subsequent rounds expose ρ through $\mathcal{F}_{i,t}^{\mathrm{rec}}$ and use it in G_{rec} until a release state changes. Occupied write states are released after the execution.

4 Experiments

4.1 Experimental Setup

Benchmark and Tasks. We evaluate StructuredCache on the open-source MINDcraft benchmark (White et al., 2025), which provides multi-agent collaborative tasks in Minecraft under partial observability. MineCollab contains three representative task families: (1) **Crafting.** Agents are provided with different resources and crafting knowledge, and must communicate, exchange materials, and coordinate multi-step actions to produce target items. (2) **Cooking.** Agents collaboratively collect ingredients, prepare intermediate food items, and coordinate the use of shared facilities such as furnaces, smokers, and crafting tables to complete target recipes. (3) **Construction.** Agents collaboratively build structures according to predefined blueprints, requiring material allocation, spatial coordination, and long-horizon execution while avoiding conflicts between agents. We evaluate StructuredCache on all three tasks which cover different forms of collaboration, including resource exchange, shared facility usage, spatial coordination, and long-horizon task execution.

Implementation Settings. StructuredCache is evaluated under two collaboration scales with 2 and 3 agents, using three LLM backbones: GPT-5.6-Luna, Gemini-2.5-Flash, and DeepSeek-V4-Flash. For each backbone, all compared methods use the same MINDcraft system prompts, few-shot examples, high-level action interfaces, environment initialization, and task configurations. StructuredCache is implemented as an execution-layer module between agent decision making and Minecraft action execution. Cache lookup, state-version checking, coherence validation, and verified state updates are performed deterministically without additional LLM calls. All compared methods are evaluated on the same task instances and environment settings. Additional implementation details and parameter settings are provided in the supplementary material.

4.2 Evaluation Metrics

We evaluate StructuredCache using two metrics that measure task performance and execution efficiency across all three task families.

Task Success Rate (TSR). Task Success Rate (TSR) measures the proportion of episodes that satisfy the task-completion criterion. Let $s_n \in \{0, 1\}$ indicate whether the n-th episode is successfully completed. Given N evaluation episodes, TSR is defined as

$$\mathrm{TSR} = \frac{1}{N} \sum_{n=1}^{N} s_n. \tag{19}$$

A higher TSR indicates better task completion performance.

Completion Steps (CS). Completion Steps (CS) measures the average number of environment-interacting actions during task execution. Given N evaluation episodes with M agents, CS is defined as

$$\mathrm{CS} = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{M} T_{n,i}, \tag{20}$$

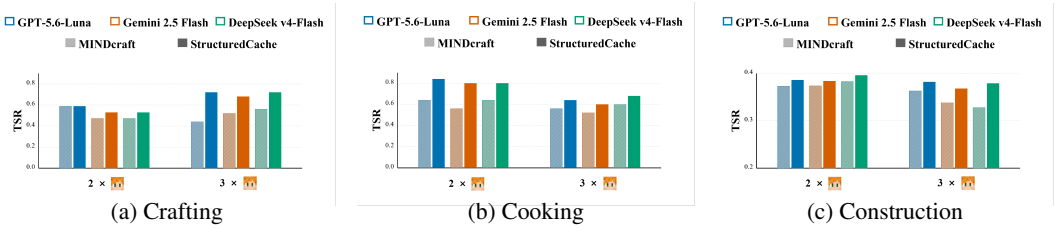
where $T_{n,i}$ denotes the number of environment-interacting actions executed by agent i in the n-th episode. A lower CS indicates more efficient collaborative execution.

Table 1: Performance comparison of MINDcraft, VillagerAgent, and StructuredCache across different LLMs on crafting, cooking, and construction tasks. TSR and CS denote Task Success Rate (%) and Completion Steps, respectively. A dash indicates an unavailable or unresolved value.

Task	Method	GPT-5.6 Luna		Gemini-2.5 Flash		DeepSeek V4-Flash	
		TSR↑	CS↓	TSR↑	CS↓	TSR↑	CS↓
 Crafting	MINDcraft	50.0	17.5	50.0	20.5	52.4	19.4
	VillagerAgent	45.2	31.8	42.9	25.4	47.6	23.1
	StructuredCache	**66.7**	**14.4**	**61.9**	**15.2**	**64.3**	**15.6**
 Cooking	MINDcraft	60.0	31.4	54.0	32.4	62.0	32.1
	VillagerAgent	60.0	39.8	52.0	35.3	54.0	35.0
	StructuredCache	**74.0**	**25.4**	**70.0**	**26.5**	**74.0**	**27.3**
 Construction	MINDcraft	36.8	202.7	35.6	200.1	35.7	205.2
	VillagerAgent	4.09	**124.2**	5.45	**91.2**	3.15	**41.5**
	StructuredCache	**38.4**	158.3	**37.6**	168.4	**38.8**	161.5

4.3 Overall Performance.

Table 1 compares StructuredCache with MINDcraft across three LLMs on crafting, cooking, and construction tasks in terms of Task Success Rate (TSR) and Completion Steps (CS).
Results are averaged across settings with 2 and 3 agents. StructuredCache consistently improves task success across all evaluated LLMs and task families. On crafting, StructuredCache improves TSR from 50.0% to 66.7% with GPT-5.6-Luna, from 50.0% to 61.9% with Gemini-2.5-Flash, and from 52.4% to 64.3% with DeepSeek-V4-Flash, corresponding to a consistent gain of 13.5 percentage points. On cooking, TSR increases from 60.0% to 74.0%, from 54.0% to 70.0%, and from 62.0% to 74.0%, respectively. StructuredCache also achieves consistent improvements on construction across all three LLMs. In addition to improving task success, StructuredCache consistently reduces CS across all evaluated tasks and LLMs. The reduction is particularly pronounced on construction, where CS decreases from 202.7 to 158.3, from 200.1 to 168.4, and from 205.2 to 161.5 across the three LLMs, respectively. These results show that StructuredCache provides consistent benefits across different LLMs in both task completion and execution efficiency.



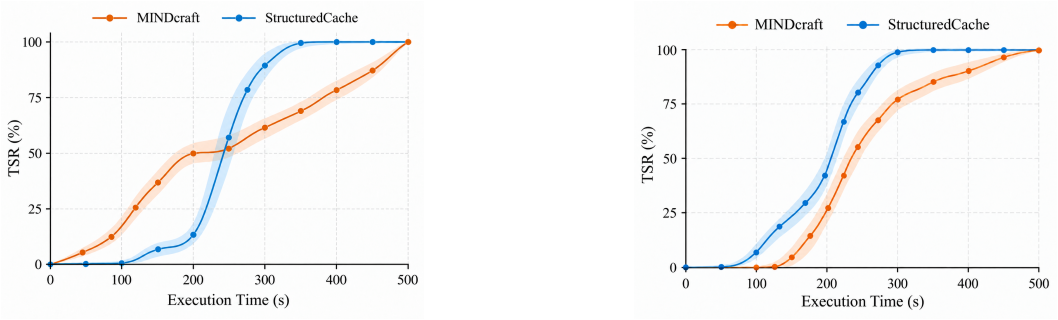
(a) Crafting (b) Cooking (c) Construction

Figure 2: Comparison of task success rates with different numbers of collaborating agents across three tasks. denotes one agent, and $N\times$ represents N collaborating agents.

Moreover, we evaluate StructuredCache with different numbers of collaborating agents. Figure 2 reports task success rates under different collaboration scales. For crafting, the largest performance gains are observed with 3 agents. The TSR increases from 44.0% to 72.0% with GPT-5.6-Luna, from 52.0% to 68.0% with Gemini-2.5-Flash, and from 56.0% to 72.0% with DeepSeek-V4-Flash. The performance advantage is also maintained with 2 agents. For cooking, MINDcraft shows decreasing performance as the number of agents increases, while StructuredCache achieves consistently higher performance across different settings. With 2 agents, the TSR gains reach 20.0, 24.0, and 16.0 percentage points for GPT-5.6-Luna, Gemini-2.5-Flash, and DeepSeek-V4-Flash, respectively. For

Table 2: Ablation study of different components of StructuredCache.

Task	Baseline	w/o Hierarchical Cache	w/o Structure	StructuredCache
Crafting	52.4	60.0	52.2	**64.3**
Cooking	62.0	64.0	62.4	**74.0**
Construction	35.7	36.3	35.8	**38.8**



(a) Crafting (b) Cooking

Figure 3: Completion-time curves on tasks completed by both methods. Each curve shows the cumulative fraction completed by time t.

construction, StructuredCache achieves consistent gains with both 2 agents and 3 agents. Overall, the results demonstrate that StructuredCache remains effective under different collaboration scales.

4.4 Ablation Study

To evaluate the contribution of different components in StructuredCache, we conduct ablation experiments on DeepSeek v4-Flash. Specifically, we ablate two key components: hierarchical cache and structured instruction. Since these components are jointly involved in the decision process, removing one component may affect the interaction with the remaining modules. Therefore, when constructing each ablation variant, we keep the original decision principles and state management mechanism unchanged, while adapting the removed component's interface to ensure a fair comparison under the same task settings.

Effect of Hierarchical Cache. We compare StructuredCache with a variant without hierarchical cache, where all task-relevant states are maintained in a single shared state repository. As shown in Table 2, removing hierarchical cache decreases TSR from 64.3% to 60.0% on crafting, from 74.0% to 64.0% on cooking, and from 38.8% to 36.3% on construction. These results demonstrate the effectiveness of hierarchical cache in organizing task-relevant states with different access patterns.

Effect of Structured Instruction. We further evaluate the contribution of structured instruction by removing the explicit modeling of operation semantics, state dependencies, and execution constraints. As shown in Table 2, removing structured instruction reduces TSR from 64.3% to 52.2% on Crafting, from 74.0% to 62.4% on Cooking, and from 38.8% to 35.8% on Construction. These results indicate that structured instruction helps reduce ambiguity in task interpretation and execution by providing explicit operation semantics and state dependencies, leading to more consistent agent decisions. Additional analysis of efficiency improvements is provided in the appendix.

4.5 Execution Efficiency Analysis

Figure 3 compares the task completion curves of MINDcraft and StructuredCache under DeepSeek-V4-Flash by reporting TSR over execution time on the common-success subset. On crafting tasks, MINDcraft progresses faster in the early stage, while StructuredCache rises rapidly after approximately 250 seconds, surpasses MINDcraft, and reaches full task completion substantially earlier. On cooking tasks, StructuredCache achieves higher TSR throughout most of the execution process and also reaches full completion earlier than MINDcraft. Overall, StructuredCache shows a clear

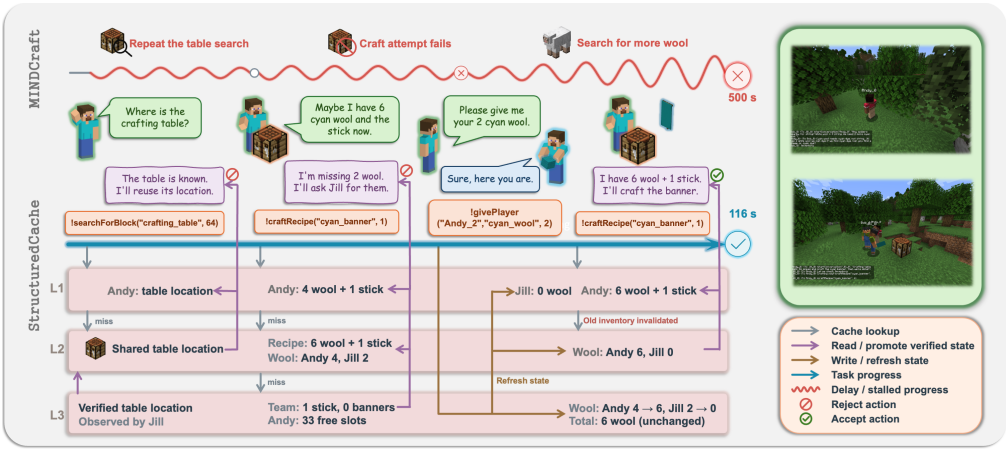


Figure 4: Case study of a cyan-banner crafting task. StructuredCache reuses the crafting-table location, blocks premature crafting, and coordinates wool transfer, while MINDCraft times out.

advantage in later-stage task completion, indicating more efficient collaborative execution. This behavior is consistent with its cache-coherent execution design, where action-related task states are validated before execution and verified updates are synchronized afterward, reducing unnecessary execution based on outdated states and allowing agents to make more effective progress toward task completion.

4.6 Case Study

We compare StructuredCache with MINDCraft on a cyan-banner crafting task to examine how verified states affect collaborative decisions. The task requires six cyan wool blocks and one stick. All materials are available within the team but distributed across agents.

Hierarchical Cache for State Reuse. Jill first discovers the crafting table. In StructuredCache, the verified location is stored in L3 and promoted to the shared L2 state. When Andy needs the crafting table, he retrieves this verified state through the cache lookup path and directly navigates to the table without redundant exploration. In contrast, MINDCraft relies on local observations and repeatedly searches for the table, leading to unnecessary exploration.

Cache-Guided State Verification. After reaching the table, Andy attempts to craft a cyan banner. In MINDCraft, Andy executes crafting with only four locally available wool blocks, failing to account for Jill's two blocks. He then searches for additional wool and times out after redundant exploration. In StructuredCache, Andy retrieves Jill's wool blocks from the verified team state, completes the transfer, and successfully crafts the cyan banner at 116s.

Analysis. This case demonstrates that StructuredCache enables agents to reuse verified environment states, verify action feasibility before execution, and coordinate distributed resources across teammates, thereby reducing redundant exploration and invalid actions.

5 Conclusion

We propose **StructuredCache**, a CPU-inspired cache-coherent execution framework for embodied multi-agent collaboration. StructuredCache integrates structured instructions, a hierarchical cache, and cache-coherent execution to provide efficient state access while maintaining task-state consistency. Experiments on Minecraft-based crafting, cooking, and construction tasks show consistent improvements in task success and execution efficiency across different LLM backbones and collaboration scales. Further analysis shows that StructuredCache maintains its effectiveness across different numbers of collaborating agents and enables faster task completion during execution. These results highlight the importance of task-state consistency for reliable and efficient embodied multi-agent collaboration.

AI USE STATEMENT

We used AI tools to assist with language editing, grammar checking, and checks for consistency in wording and reported numerical values. The authors reviewed all AI-assisted suggestions and take responsibility for the accuracy, integrity, and final content of this paper.

REFERENCES

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*, volume 35, pp. 24639–24654, 2022. doi: 10.52202/068431-1789.

Matthew Chang, Gunjan Chhablani, Alexander Clegg, Mikael Dallaire Cote, Ruta Desai, Michal Hlavac, Vladimir Karashchuk, Jacob Krantz, Roozbeh Mottaghi, Priyam Parashar, Siddharth Patki, Ishita Prasad, Xavier Puig, Akshara Rai, Ram Ramrakhya, Daniel Tran, Joanne Truong, John Turner, Eric Undersander, and Tsung-Yen Yang. PARTNR: A benchmark for planning and reasoning in embodied multi-agent tasks. In *International Conference on Learning Representations*, volume 2025, pp. 65205–65268, 2025.

Junting Chen, Checheng Yu, Xunzhe Zhou, Tianqi Xu, Yao Mu, Mengkang Hu, Wenqi Shao, Yikai Wang, Guohao Li, and Lin Shao. EMOS: Embodiment-aware heterogeneous multi-robot operating system with LLM agents. In *International Conference on Learning Representations*, volume 2025, pp. 56670–56690, 2025.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024.

Guanting Dong, Xiaoshuai Song, Yuyang Hu, Jiajie Jin, Chenghao Zhang, Yifei Chen, Xiaoxi Li, Huaying Yuan, Xinyu Yang, Tongyu Wen, Jiejun Tan, Hongjin Qian, Shijue Huang, Junting Lu, Zhenyu Li, Wanjun Zhong, Yutao Zhu, Tat-Seng Chua, Zhicheng Dou, and Ji-Rong Wen. Towards long-horizon agents: A survey. *Preprints*, 2026. doi: 10.20944/preprints202607.1328.v1.

Yubo Dong, Xukun Zhu, Zhengzhe Pan, Linchao Zhu, and Yi Yang. VillagerAgent: A graph-based multi-agent framework for coordinating complex task dependencies in Minecraft. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 16290–16314. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.964.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. PaLM-E: An embodied multimodal language model. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 8469–8488. PMLR, 2023.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. MineDojo: Building open-ended embodied agents with internet-scale knowledge. In *Advances in Neural Information Processing Systems*, volume 35, pp. 18343–18362, 2022. doi: 10.52202/068431-1333.

Chuang Gan, Siyuan Zhou, Jeremy Schwartz, Seth Alter, Abhishek Bhandwaldar, Dan Gutfreund, Daniel L. K. Yamins, James J. DiCarlo, Josh McDermott, Antonio Torralba, and Joshua B. Tenenbaum. The ThreeDWorld transport challenge: A visually guided task-and-motion planning benchmark towards physically realistic embodied AI. In *2022 International Conference on Robotics and Automation (ICRA)*, pp. 8847–8854, 2022. doi: 10.1109/ICRA46639.2022.9812329.

Xudong Guo, Kaixuan Huang, Jiale Liu, Wenhui Fan, Natalia Vélez, Qingyun Wu, Huazheng Wang, Thomas L. Griffiths, and Mengdi Wang. Embodied LLM agents learn to cooperate in organized teams. *IEEE Transactions on Computational Social Systems*, 13(2):2514–2530, 2026. doi: 10.1109/TCSS.2025.3637527.

William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. MineRL: A large-scale dataset of Minecraft demonstrations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 2442–2448, 2019. doi: 10.24963/ijcai.2019/339.

William H. Guss, Mario Ynocente Castro, Sam Devlin, Brandon Houghton, Noboru Sean Kuno, Crissman Loomis, Stephanie Milani, Sharada Mohanty, Keisuke Nakata, Ruslan Salakhutdinov, John Schulman, Shinya Shiroshita, Nicholay Topin, Avinash Ummadisingu, and Oriol Vinyals. The MineRL 2020 competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:2101.11071*, 2021.

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6 edition, 2017.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven 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*, volume 2024, pp. 23247–23275, 2024.

Shengchao Hu, Li Shen, Ya Zhang, and Dacheng Tao. Learning multi-agent communication from graph modeling perspective. In *International Conference on Learning Representations*, volume 2024, pp. 12963–12978, 2024.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 1769–1782. PMLR, 2023.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023.

Anssi Kanervisto, Stephanie Milani, Karolis Ramanauskas, Nicholay Topin, Zichuan Lin, Junyou Li, Jianing Shi, Deheng Ye, Qiang Fu, Wei Yang, Weijun Hong, Zhongyue Huang, Haicheng Chen, Guangjun Zeng, Yue Lin, Vincent Micheli, Eloi Alonso, François Fleuret, Alexander Nikulin, Yury Belousov, Oleg Svidchenko, and Aleksei Shpilman. MineRL diamond 2021 competition: Overview, results, and lessons learned. In *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*, volume 176 of *Proceedings of Machine Learning Research*, pp. 13–28. PMLR, 2022.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. SMART-LLM: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024. doi: 10.1109/IROS58592.2024.10802322.

Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148–159, 1990. doi: 10.1145/325164.325132.

Guohao Li, Hasan 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*, volume 36, pp. 51991–52008, 2023. doi: 10.52202/075280-2264.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160591.

Shalev Lifshitz, Keiran Paster, Harris Chan, Jimmy Ba, and Sheila McIlraith. STEVE-1: A generative model for text-to-behavior in Minecraft. In *Advances in Neural Information Processing Systems*, volume 36, pp. 69900–69929, 2023. doi: 10.52202/075280-3064.

Jie Liu, Pan Zhou, Yingjun Du, Ah-Hwee Tan, Cees G. Snoek, Jan-Jakob Sonke, and Efstratios Gavves. CaPo: Cooperative plan optimization for efficient embodied multi-agent cooperation. In *International Conference on Learning Representations*, volume 2025, pp. 70791–70815, 2025a.

Peiqi Liu, Zhanqiu Guo, Mohit Warke, Soumith Chintala, Chris Paxton, Nur Muhammad Mahi Shafiullah, and Lerrel Pinto. DynaMem: Online dynamic spatio-semantic memory for open world mobile manipulation. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, 2025b. doi: 10.1109/ICRA55743.2025.11127619.

Zhao Mandi, Shreeya Jain, and Shuran Song. RoCo: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 286–299, 2024. doi: 10.1109/ICRA57147.2024.10610855.

Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Springer Cham, 2 edition, 2020. doi: 10.1007/978-3-031-01764-3.

Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P. How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2681–2690. PMLR, 2017.

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

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*, pp. 1–22. Association for Computing Machinery, 2023. doi: 10.1145/3586183.3606763.

Shivansh Patel, Saim Wani, Unnat Jain, Alexander G. Schwing, Svetlana Lazebnik, Manolis Savva, and Angel X. Chang. Interpretation of emergent communication in heterogeneous collaborative embodied agents. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 15953–15963, 2021.

Xavier Puig, Tianmin Shu, Shuang Li, Zilin Wang, Yuan-Hong Liao, Joshua B. Tenenbaum, Sanja Fidler, and Antonio Torralba. Watch-and-help: A challenge for social perception and human-AI collaboration. In *International Conference on Learning Representations*, 2021.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810.

Yiran Qin, Li Kang, Xiufeng Song, Zhenfei Yin, Xiaohong Liu, Xihui Liu, Ruimao Zhang, and Lei Bai. RoboFactory: Exploring embodied agent collaboration with compositional constraints. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 10075–10085, 2025.

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*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

Huajie Tan, Xiaoshuai Hao, Cheng Chi, Minglan Lin, Yaoxu Lyu, Mingyu Cao, Dong Liang, Zhuo Chen, Mengsi Lyu, Cheng Peng, Chenrui He, Yulong Ao, Yonghua Lin, Pengwei Wang, Zhongyuan Wang, and Shanghang Zhang. RoboOS: A hierarchical embodied framework for cross-embodiment and multi-agent collaboration. *arXiv preprint arXiv:2505.03673*, 2025.

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.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian (Shawn) Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with LLMs enables open-world multi-task agents. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34153–34189, 2023. doi: 10.52202/075280-1480.

Isadora White, Kolby Nottingham, Ayush Maniar, Max Robinson, Hansen Lillemark, Mehul Maheshwari, Lianhui Qin, and Prithviraj Ammanabrolu. Collaborating action by action: A multi-agent LLM framework for embodied reasoning. *arXiv preprint arXiv:2504.17950*, 2025.

Di Wu, Xian Wei, Guang Chen, Hao Shen, and Bo Jin. Generative multi-agent collaboration in embodied AI: A systematic review. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 10723–10732, 2025. doi: 10.24963/ijcai.2025/1190.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

Yuchen Xiao, Weihao Tan, Joshua Hoffman, Tian Xia, and Christopher Amato. Asynchronous multi-agent deep reinforcement learning under partial observability. *The International Journal of Robotics Research*, 44(8):1257–1286, 2025. doi: 10.1177/02783649241306124.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Guibin Zhang, Muxin Fu, Kun Wang, Frank Wan, Miao Yu, and Shuicheng Yan. G-Memory: Tracing hierarchical memory for multi-agent systems. In *Advances in Neural Information Processing Systems*, volume 38, pp. 12988–13018, 2025a. doi: 10.52202/085713-0439.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. In *International Conference on Learning Representations*, volume 2024, pp. 19373–19401, 2024.

Hongxin Zhang, Zeyuan Wang, Qiushi Lyu, Zheyuan Zhang, Sunli Chen, Tianmin Shu, Behzad Dariush, Kwonjoon Lee, Yilun Du, and Chuang Gan. COMBO: Compositional world models for embodied multi-agent cooperation. In *International Conference on Learning Representations*, volume 2025, pp. 49996–50019, 2025b.

Xiaopan Zhang, Hao Qin, Fuquan Wang, Yue Dong, and Jiachen Li. LaMMA-P: Generalizable multi-agent long-horizon task allocation and planning with LM-driven PDDL planner. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2025c.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946.

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.

Lizheng Zu, Lin Lin, Song Fu, Na Zhao, and Pan Zhou. Collaborative tree search for enhancing embodied multi-agent collaboration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 29513–29522, 2025. doi: 10.1109/CVPR52734.2025.02748.

A APPENDIX

A.1 STRUCTUREDCACHE PROMPTS

StructuredCache adds a runtime policy for instruction-scoped cache retrieval, coherence gating, and bounded action revision. Dynamic fields such as agent names, candidate commands, valid command signatures, and resolved cache facts are inserted at runtime and are therefore not part of the fixed prompt text.

Listing 1: Fixed cache-aware runtime policy.

```
For each physical candidate, the runtime derives A(u)=R(u) union W(u),
reads L1 then L2 then L3, promotes fallback hits to private L1, and
    shows
only that candidate and its resolved L1 entries.
On a coherent hit, execute the exact Agent candidate through the gate;
     do
not call the model.
On a verified miss, contradiction, no-progress advisory, or unresolved
receipt, allow one bounded atomic-action revision; re-encode and
coherence-gate any replacement.
Valid environment-verified facts are authoritative only for their
    listed
keys, scope, and bound versions. Missing or undisplayed state is
    unknown
unless a complete snapshot proves count zero.
Choose routes, roles, ordering, and communication from your local
    history;
the cache never assigns a global plan. Do not copy raw cache payloads
    into
ordinary replies.
Before crafting, obtain a successful structured plan for the exact
    item so
its recipe becomes verified cache state. Craft/smelt feedback exposes
    one
locally selected verified direct route and its run count; alternatives
remain in L3.
If an unsent draft asks about a named Agent's items, one read-only
    complete
inventory projection may trigger one factual rewrite; absent means
    zero
only when projection_complete=true.
After execution, use the one execution receipt and refreshed L1. Only
committed is complete. A failure stays bound to its release-key
    versions,
so change evidence, parameters, coordination, or action before
    retrying.
A verified no-match search covers only its recorded area; move to
uncovered space or change strategy.
Use at most one command per response. Ask teammates in plain language:
never write their command or put nested command syntax inside a
conversation message. Internal opcodes are not commands.
```

A.2 BOUNDED ACTION REVISION

When the coherence check detects a verified miss, contradiction, no-progress risk, or unresolved execution receipt, the runtime allows one bounded atomic-action revision. The revised command is re-encoded as a structured instruction and checked again before physical execution. Runtime-inserted fields such as `agent_name`, `valid_revision_commands`, `candidate_command`, and `instruction_context` are excluded from the fixed prompt.

Listing 2: Fixed action-revision policy.

```
You already proposed the physical Mindcraft command below.
The runtime derived A(u), queried L1 then L2 then L3, promoted every
```

```
fallback hit into L1, and is showing only the resulting current-instruction
L1.
This is the action-revision stage after instruction-scoped cache retrieval.
Judge exactly one atomic Mindcraft command, not whether it completes the
whole request or subgoal by itself.
COMMIT when the verified facts support its exact parameters and it makes a
necessary intermediate step or other non-negative task progress.
A one-item transfer, one craft step, or one acquisition step may COMMIT
even when later steps are still required.
Do not REPLAN merely because another item, transfer, craft, or coordination
step must happen afterward.
The candidate command has already passed syntax and parameter-domain
validation.
The cache feedback is scoped to this candidate instruction. Undisplayed
task state is unknown, not a contradiction.
If the exact same candidate should execute now, return exactly: COMMIT
If the scoped facts require a different immediately executable response,
return exactly: REVISE <one valid Mindcraft command for yourself>
When local_shortages names another verified holder: if you are missing the
item, use a conversation command to ask that holder; only a holder may give
the item.
When recipe evidence is missing, use getCraftingPlan if available;
otherwise ask a permitted teammate for the exact plan.
Return REPLAN <short reason> only when the change cannot be expressed
safely as one command and needs another full planning pass.
If no useful action should be taken now, return exactly:
ABORT <short reason>
Output only one of COMMIT, REVISE plus one command, REPLAN plus a short
reason, or ABORT plus a short reason.
```

A.3 Structured Instruction and Cache Serialization

Each candidate physical command is converted into a structured instruction with an opcode, parameters, read set, write set, bound state versions, and an execution guard:

$$u = (o_u, \boldsymbol{\theta}_u, \mathcal{R}(u), \mathcal{W}(u), \hat{\mathcal{V}}(u), g_u), \qquad \mathcal{A}(u) = \mathcal{R}(u) \cup \mathcal{W}(u). \tag{21}$$

The runtime resolves only states in $\mathcal{A}(u)$ and exposes them as current-instruction L1 facts. It does not dump the full global cache into the prompt.

Listing 3: Prompt-visible cache serialization template.

```
{
  "instruction_id": "<task_id>:<agent>:<session_prefix>:<sequence>",
  "opcode": "<atomic_operation>",
  "params": { "<parameter>": <value> },
  "facts": [
    {
      "key": "<state_key>",
      "value": { "<environment_verified_field>": <value> },
      "version": <state_version>,
      "valid": <boolean>,
      "source": "<resolved_cache_source>"
    }
  ],
```

```
  "missing_verified_states": ["<state_key>"],
  "decision_advisory": <runtime_advisory_or_null>,
  "action_local_assessment": <instruction_scoped_assessment_or_null>,
  "active_execution_constraint": <optional_receipt_constraint>
}
```

The prompt-visible cache block is terminated by two fixed rules:

Listing 4: Fixed authoritative-cache boundary rules.

```
AUTHORITATIVE CACHE RULE: For the listed keys, scope, and bound versions,
the latest valid verified facts above override conflicting dialogue, memory,
wiki text, or model knowledge.
Undisplayed state is unknown unless a complete snapshot proves absence.
Assess only this candidate and do not substitute recipe inputs, quantities,
owners, or assemblers from memory.
```

A.4 Task-Specific State Templates

This section gives access templates for the operation families used by StructuredCache in the MINDcraft experiments, shown in Table 3. The operations described in the main paper include collecting, crafting, transferring, smelting, searching, and construction-specific placement and clearing. The templates below also cover supporting query and movement operations. Smelting uses the same structured-instruction interface as other production actions.

Table 3: One-to-one correspondence between runtime operations and corresponding templates.

Operation	Template	Primary read states	Guard / verification
Query	Inventory / recipe projection	inventory, recipe, resource	read-only snapshot
Move	Navigation	agent pose, target location, region	reached / no-path receipt
Search	Area-bounded search	region, target object/entity	found or verified no-match
Collect	Resource acquisition	resource, inventory, region	inventory/block delta
Craft	Crafting	recipe, inventory, resource, facility	output inventory delta
Smelt	Cooking / smelting	recipe, input, fuel, facility	output inventory delta
Transfer	Item handover	sender/recipient inventories	bilateral inventory delta
Place	Facility/block placement	inventory, block, region	block-state delta
Construction place	Blueprint-coordinate placement	target, inventory, support	target block observed
Construction clear	Blueprint-coordinate clearing	target block	target becomes air
Receipt	Write-back / retry control	affected write keys	environment-verified commit

A.5 Query and Inventory Projection

Read-only queries expose bounded factual state and do not write task progress. They are used for inventory projection, recipe lookup, and local status checks. A complete inventory projection is the only case where absence may be interpreted as zero for the projected scope.

Listing 5: Query access template.

```
opcode = query
R(u) = {
  [inventory:<agent>:<item_j>],
  [resource:<item_j>],
  [recipe:<item_j>],
  [task_progress]
}
W(u) = {}
preconditions = [
  states_fresh(R(u))
]
guard = read_only_snapshot(scope=<query_scope>,
                           projection_complete=<boolean>)
```

A.5.1 MOVE

Move commands resolve the agent's current position, target location, and optional region status. They are considered successful only when the execution receipt indicates that the agent reached the target or was already close enough.

Listing 6: Move access template.

```
opcode = move

R(u) = {
  pose:<agent>,
  location:<target>,
  [region:<target_region>]
}

W(u) = {
  pose:<agent>,
  [region:<target_region>:visited_by:<agent>]
}

preconditions = [
  states_fresh(R(u)),
  target_resolved(location:<target>)
]

guard = navigation_receipt(target=<target>,
                           success_markers=["reached",
                                            "already_close_enough"])
```

A.5.2 SEARCH

Search commands are bounded by the recorded search area. A no-match result does not prove global absence; it only verifies that the recorded area did not contain the target under the executed search parameters.

Listing 7: Search access template.

```
opcode = search

R(u) = {
  pose:<agent>,
  region:<search_area>,
  target:<block_or_entity>
}

W(u) = {
  region:<search_area>:search_result:<target>,
  [location:<target_instance>]
}

preconditions = [
  states_fresh(R(u)),
  search_area_bounded(region:<search_area>)
]

guard = area_bounded_search_result(target=<block_or_entity>,
                                   area=<search_area>,
                                   result=<found_or_no_match>)
```

A.5.3 COLLECT

Collect commands access both team-level resource state and executor-local inventory state. This distinction allows the runtime to reason about resource availability for collaboration while still checking whether the executing agent can perform the immediate command.

Listing 8: Collect access template.

```
opcode = collect

R(u) = {
  pose:<agent>,
  resource:<item>,
  inventory:<agent>:<item>,
  [block:<item_source>:<x>:<y>:<z>],
  [entity:<item_source>:<entity_id>]
}

W(u) = {
  resource:<item>,
  inventory:<agent>:<item>,
  [block:<item_source>:<x>:<y>:<z>],
  [entity:<item_source>:<entity_id>]
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u))
]

guard = acquisition_delta(item=<item>,
                          agent=<agent>,
                          quantity=<quantity>)
```

A.5.4 CRAFTING

For a crafting target `output`, recipe inputs `ingredients`, and an optional crafting facility, StructuredCache uses the following access template.

Listing 9: Crafting access template.

```
opcode = craft

R(u) = {
  recipe:<output>,
  inventory_space:<agent>,
  resource:<ingredient_j>,
  inventory:<agent>:<ingredient_j>,
  resource:<output>,
  inventory:<agent>:<output>,
  [facility:crafting_table:<x>:<y>:<z>]
}

W(u) = {
  resource:<output>,
  inventory:<agent>:<output>,
  resource:<ingredient_j>,
  inventory:<agent>:<ingredient_j>,
  [facility:crafting_table:<x>:<y>:<z>]
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u)),
  recipe_verified(recipe:<output>)
]

guard = environment_delta(item=<output>, quantity=<quantity>)
priority = { immediate_l1: 3, shared_l2: 2 }
```

A.5.5 COOKING AND SMELTING

Cooking tasks use the same crafting template for non-smelting operations. For smelting, the instruction additionally binds input food, fuel, and furnace or smoker state.

Listing 10: Smelting access template.

```
opcode = smelt

R(u) = {
  recipe:<output>,
  inventory_space:<agent>,
  resource:<input>,
  inventory:<agent>:<input>,
  resource:smelting_fuel,
  inventory:<agent>:smelting_fuel,
  resource:<output>,
  inventory:<agent>:<output>,
  facility:<furnace_or_smoker>:<x>:<y>:<z>
}

W(u) = {
  resource:<output>,
  inventory:<agent>:<output>,
  resource:<input>,
  inventory:<agent>:<input>,
  resource:smelting_fuel,
  inventory:<agent>:smelting_fuel,
  facility:<furnace_or_smoker>:<x>:<y>:<z>
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u)),
  recipe_verified(recipe:<output>)
]

guard = environment_delta(item=<output>, quantity=<quantity>)
priority = { immediate_l1: 3, shared_l2: 2 }
```

A.5.6 ITEM TRANSFER

For item handover, StructuredCache verifies both sender and recipient inventory changes. Sender decrease and recipient increase must jointly hold in the environment.

Listing 11: Transfer access template.

```
opcode = transfer

R(u) = {
  inventory:<sender>:<item>,
  inventory:<recipient>:<item>,
  resource:<item>
}

W(u) = {
  inventory:<sender>:<item>,
  inventory:<recipient>:<item>,
  resource:<item>
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u)),
  inventory:<sender>:<item> >= <quantity>
]

```

```
guard = bilateral_inventory_delta(
  sender=<sender>,
  recipient=<recipient>,
  item=<item>,
  quantity=<quantity>
)
```

A.5.7 PLACE

Place commands modify both the executor inventory and a spatial block or facility state. They are used when a task requires placing a block, creating a shared facility, or updating a task-relevant spatial region.

Listing 12: Place access template.

```
opcode = place

R(u) = {
  inventory:<agent>:<block_item>,
  pose:<agent>,
  block:<x>:<y>:<z>,
  region:<placement_region>
}

W(u) = {
  inventory:<agent>:<block_item>,
  block:<x>:<y>:<z>,
  region:<placement_region>,
  [facility:<facility_type>:<x>:<y>:<z>]
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u)),
  inventory:<agent>:<block_item> >= 1
]

guard = block_state_delta(position=<x,y,z>,
                          expected_block=<block_item>)
```

A.5.8 CONSTRUCTION PLACEMENT AND CLEARING

The construction commands are `!placeBlockAt` and `!breakBlueprintBlockAt`, both with explicit coordinates. Before either command, the runtime observes the target coordinate and resolves its expected block from the task blueprint. Placement additionally observes adjacent support blocks. The coordinate observations are committed to L3 and promoted to the acting agent's L1. Shared resource information can aid planning, but it is not in these atomic access sets. An occupied, incorrect target must be cleared before placement. The placement skill checks local inventory availability. Each action is verified by observing the target block afterward.

Listing 13: Construction placement access template.

```
opcode = place

R(u) = {
  block:<x>:<y>:<z>,
  inventory:<agent>:<block>,
  support:<sx>:<sy>:<sz>
}

W(u) = {
  block:<x>:<y>:<z>,
  inventory:<agent>:<block>
}

preconditions = [
```

```
  states_fresh(R(u)),
  writes_available(W(u)),
  blueprint_target(<x,y,z>) = <block> != air,
  observed(block:<x>:<y>:<z>) = air,
  support_usable(support:<sx>:<sy>:<sz>),
  inventory:<agent>:<block> >= 1
]

guard = command_success_and_block_observed(
  position=<x,y,z>, expected_block=<block>)
```

Listing 14: Construction clearing access template.

```
opcode = clear

R(u) = {
  block:<x>:<y>:<z>
}

W(u) = {
  block:<x>:<y>:<z>
}

preconditions = [
  states_fresh(R(u)),
  writes_available(W(u)),
  blueprint_target(<x,y,z>) is defined,
  observed(block:<x>:<y>:<z>) != air,
  observed(block:<x>:<y>:<z>) != blueprint_target(<x,y,z>)
]

guard = command_success_and_block_observed(
  position=<x,y,z>, expected_block=air)
```

A.5.9 Execution Receipt and Commit

Execution receipts are not agent actions, but they are part of the cache-coherent loop. A receipt binds the executed instruction, guard result, verified state keys, and release-key versions used to prevent blind repeated retry.

Listing 15: Execution receipt and commit template.

```
receipt = {
  "receipt_id": "<instruction_id>:receipt:<event_index>",
  "instruction_id": "<instruction_id>",
  "opcode": "<opcode>",
  "command": "<mindcraft_command>",
  "status": "<committed|blocked|failed>",
  "success": <boolean>,
  "reason": {
    "code": "<environment_verified|coherence_reject|execution_failed
        >",
    "detail": "<short_reason>"
  },
  "verification": <guard_specific_result>,
  "verification_fence": {
    "cache_version": <version>,
    "completed_at_event": <event_index>
  },
  "refreshed_l1": {
    "scope": "execution_receipt_instruction_l1",
    "entries": { "<state_key>": "<verified_state_entry>" }
  }
}
```

A.6 Additional Experimental Configuration

A.6.1 Cache Hyperparameters

Table 4 lists the hyperparameter settings of StructuredCache used throughout the experiments, including cache capacities, the heat decay factor, event weights, the L2 admission threshold, and the heat statistics window.

Table 4: StructuredCache hyperparameters used in the experiments.

Parameter	Value
L1 capacity per agent	12
Shared L2 capacity	24
Version window	8
Heat decay β	0.85
Write-access weight λ_{write}	1.5
Cross-agent weight λ_{agent}	1
Coordination-event weight $\lambda_{\text{coordination}}$	2
L2 admission threshold	2
Heat statistics window	24

A.6.2 Construction Setting

Construction uses the MINDcraft blueprint tasks with 2 or 3 agents. Each task specifies the target blocks and their coordinates, and distributes the required materials among agents. Baseline and StructuredCache use the same blueprints, initial inventories, agent counts, and task time limits. Agents coordinate material use and place or clear blocks in the shared world. An episode succeeds only when the construction validator finds no mismatch between the built structure and its blueprint. Task success rate (TSR) and completion steps (CS) are computed as in the main paper.

A.7 The Pseudo Code

Algorithm 1 summarizes the coherent execution process of StructuredCache. Before execution, the candidate action is resolved against the cache hierarchy and revised when stale, missing, or conflicting states are detected; after execution, the outcome is verified, and the cache is updated or synchronized accordingly, with an execution receipt stored for subsequent decisions.

A.8 Task Performance across Collaboration Scales

Tables 5 and 6 report the task success rates under different collaboration scales. StructuredCache generally improves performance over the baseline across both 2-agent and 3-agent settings, with particularly clear gains in the 3-agent crafting task and the 2-agent cooking task.

Table 5: Crafting task success rate (%) by collaboration scale.

Method	Scale	GPT-5.6	Gemini	DeepSeek
Baseline	2-agent	58.8	47.1	47.1
StructuredCache	2-agent	58.8	52.9	52.9
Baseline	3-agent	44.0	52.0	56.0
StructuredCache	3-agent	72.0	68.0	72.0

Algorithm 1 StructuredCache Coherent Execution

Require: Agent i, candidate action $a_{i,t}^{\text{cand}}$

Encode $a_{i,t}^{\text{cand}}$ as instruction u with $\mathcal{A}(u) = \mathcal{R}(u) \cup \mathcal{W}(u)$.
Resolve states in $\mathcal{A}(u)$ through L1→L2→L3 and construct instruction scoped feedback $\mathcal{F}_{i,t}(u)$.
if $\mathcal{F}_{i,t}(u)$ exposes a stale version, missing verified state, write conflict, or unresolved receipt **then**
    Produce a bounded revision $a_{i,t}^{\text{rev}}$ using $\mathcal{F}_{i,t}(u)$, and encode it as $\tilde{u}$.
else
    Set $\tilde{u} \leftarrow u$.
end if
if $G_{\text{coh}}(i, \tilde{u}) G_{\text{rec}}(i, \tilde{u}) = 0$ **then**
    Reject $\tilde{u}$ and return an execution receipt.
end if
Acquire write ownership for $\mathcal{W}(\tilde{u})$ and execute $\tilde{u}$.
Verify the outcome from environment observation e_{t+1}.
if the outcome is verified **then**
    Commit verified writes to L3, advance versions, invalidate stale L1/L2 copies, and refresh writer-side caches.
else
    Re-observe affected states, synchronize verified changes, and create a failure receipt.
end if
Release write ownership and store the receipt for later decisions.

Table 6: Cooking task success rate (%) by collaboration scale.

Method	Scale	GPT-5.6	Gemini	DeepSeek
Baseline	2-agent	64.0	56.0	64.0
StructuredCache	2-agent	84.0	80.0	80.0
Baseline	3-agent	56.0	52.0	60.0
StructuredCache	3-agent	64.0	60.0	68.0

Table 7: Construction task success rate (%) by collaboration scale.

Method	Scale	GPT-5.6	Gemini	DeepSeek
Baseline	2-agent	37.3	37.4	38.3
StructuredCache	2-agent	38.6	38.4	39.6
Baseline	3-agent	36.3	33.8	32.8
StructuredCache	3-agent	38.2	36.8	37.9

A.9 Additional Behavioral Analysis

To inspect whether StructuredCache changes execution behavior, we additionally compute process-level analysis from execution logs. These metrics are diagnostic rather than repeated-seed statistical estimates.

A.9.1 Metric Definitions

Long-task subset. Long tasks are selected within the same task family and agent scale by taking the top $\lceil N/3 \rceil$ tasks under a fixed task-length proxy. For Crafting, the proxy is the task-definition dependency depth. For Cooking, the proxy is the total number of explicit recipe steps for the target items. Ties are broken by the fixed order in the task file and not by experimental outcomes.

Invalid action request rate. We count task actions that can change location or environment state, including navigation, collection, attack, transfer, crafting, smelting, and facility operations. The in-

valid action request rate is

$$R_{\text{invalid}} = \frac{N_{\text{failure}}}{N_{\text{success}} + N_{\text{failure}}}. \tag{22}$$

Parameter errors, insufficient resources, missing targets, navigation failures, handover failures, and StructuredCache pre-execution rejections of stale or repeated candidates are counted as failures. Requests with undefined, truncated, or unclassifiable receipts are excluded from this rate but remain included in total action-request counts.

Duplicate action rate. A physical action candidate is duplicate if it has the same agent, command name, and parameters as the previous physical action candidate from the same agent. Read-only queries and dialogue do not break the physical-action sequence, while any different physical command does.

Route correctness. Route correctness measures whether task-relevant search or navigation choices remain within the set of semantically valid resources, entities, or facilities for the task. The allowed set is derived from the task file, Minecraft crafting and smelting dependencies, legal fuel choices, and task-specific alternatives such as apple drops. The metric checks semantic path consistency, not shortest-path optimality.

Route execution success. Route execution success is computed over `goToCoordinates`, `goToPlayer`, `goToRememberedPlace`, `searchForBlock`, and `searchForEntity`. Receipts containing reached or already-close-enough markers are counted as successful; missing targets, parameter errors, and navigation failures are counted as failures.

Item transfer success. A transfer is counted as successful only when the execution receipt explicitly confirms that the recipient received the item. Reaching the recipient without a receipt-level item transfer does not count as handover success.

Production action success. A production action is counted as successful when a crafting or smelting receipt explicitly reports the expected produced item. Insufficient recipe resources, unavailable facilities, parameter errors, and pre-execution blocking are counted as failures.

Average action requests. Average action requests divide all task-action requests by the number of tasks in the corresponding evaluation set. Pre-execution rejections and unclassifiable receipts are retained in this total.

A.9.2 Three-Agent Cooking Analysis

We analyze execution-level behaviors in the three-agent cooking setting to examine how StructuredCache affects coordination efficiency, shown in Table 8. StructuredCache reduces invalid and duplicate actions, improves route correctness, item transfer, and production success, and lowers the average number of action requests by 23.9%.

Table 8: Additional action-level analysis for three-agent cooking tasks.

Metric	Baseline	StructuredCache	Difference
Invalid action request rate (%)	29.6	24.2	-5.5 pp
Duplicate action rate (%)	5.4	3.7	-1.7 pp
Route correctness (%)	72.0	88.0	+16.0 pp
Action-level on-route choices (%)	95.2	98.3	+3.1 pp
Route execution success (%)	67.3	69.8	+2.5 pp
Item transfer success (%)	65.0	79.0	+14.0 pp
Production action success (%)	59.8	70.1	+10.3 pp
Average action requests per task	33.12	25.20	-23.9%

A.9.3 Long-Task Subset

We evaluate tasks to examine performance under more challenging long-horizon settings, as reported in Table 9. StructuredCache substantially improves success on three-agent crafting and cooking tasks, while showing mixed results in the two-agent settings. We further compare the median

completion time of successful episodes on the long-task subset to assess execution efficiency, presented in Table 9. StructuredCache consistently reduces completion time across all four settings, with particularly clear gains in the three-agent tasks.

Table 9: Task success and median completion time on the longest top-third subset.

		Task Success (%)		Median Time (s)	
Setting	Subset Size	Baseline	StructuredCache	Baseline	StructuredCache
2-agent Crafting	6/17	66.7	33.3	273.231	184.120
3-agent Crafting	9/25	11.1	66.7	301.175	201.728
2-agent Cooking	9/25	77.8	77.8	200.168	185.895
3-agent Cooking	9/25	66.7	88.9	273.031	204.437

A.9.4 Long-Task Cooking Action Analysis

We analyze cooking tasks under 2- and 3-agent settings. As shown in Table 10, StructuredCache improves routing and transfer efficiency while reducing action requests, supporting more efficient long-horizon coordination.

Table 10: Action-level analysis on the longest nine cooking tasks for each agent scale.

Scale	Metric	Baseline	StructuredCache	Difference
2-agent	Invalid action request rate (%)	23.3	13.8	-9.5 pp
2-agent	Duplicate action rate (%)	5.6	2.6	-3.0 pp
2-agent	Route correctness (%)	100.0	100.0	0.0 pp
2-agent	Route execution success (%)	70.1	80.9	+10.8 pp
2-agent	Item transfer success (%)	76.0	75.0	-1.0 pp
2-agent	Production action success (%)	73.3	95.0	+21.7 pp
2-agent	Average action requests per task	20.33	17.44	-14.2%
3-agent	Invalid action request rate (%)	25.8	29.1	+3.3 pp
3-agent	Duplicate action rate (%)	3.9	5.0	+1.1 pp
3-agent	Route correctness (%)	66.7	88.9	+22.2 pp
3-agent	Route execution success (%)	66.7	69.7	+3.0 pp
3-agent	Item transfer success (%)	67.6	71.4	+3.8 pp
3-agent	Production action success (%)	95.2	47.6	-47.6 pp
3-agent	Average action requests per task	31.78	25.67	-19.2%