VillagerAgent: A Graph-Based Multi-Agent Framework for Coordinating Complex Task Dependencies in Minecraft

Yubo Dong, Xukun Zhu, Zhengzhe Pan, Linchao Zhu†, Yi Yang
ReLER, CCAI, Zhejiang University
† Corresponding author

Abstract

In this paper, we aim to evaluate multi-agent systems against complex dependencies, including spatial, causal, and temporal constraints. First, we construct a new benchmark, named **VillagerBench**, within the Minecraft environment. VillagerBench comprises diverse tasks crafted to test various aspects of multi-agent collaboration, from workload distribution to dynamic adaptation and synchronized task execution. Second, we introduce a Directed Acyclic Graph Multi-Agent Framework (**VillagerAgent**) to resolve complex inter-agent dependencies and enhance collaborative efficiency. This solution incorporates a task decomposer that creates a directed acyclic graph (DAG) for structured task management, an agent controller for task distribution, and a state manager for tracking environmental and agent data. Our empirical evaluation on VillagerBench demonstrates that VillagerAgent outperforms the existing AgentVerse model, reducing hallucinations and improving task decomposition efficacy. The results underscore VillagerAgent's potential in advancing multi-agent collaboration, offering a scalable and generalizable solution in dynamic environments. Source code is open-source on GitHub. [1]

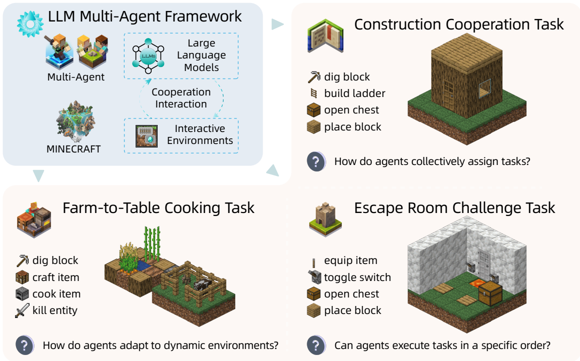

Figure 1: Minecraft Multi-Agent Benchmark (VillagerBench) is the first multi-scenario benchmark designed to evaluate the cooperative capabilities of multi-agent systems within the real-world context of Minecraft.

1 Introduction

Multi-agent collaboration using LLM is a challenging research topic that aims to enable multiple autonomous agents to coordinate their actions and achieve a common goal (Wang et al., 2023b; Xi et al., 2023; Qian et al., 2023b,a; Xie et al., 2023; Wu et al., 2023a). The collaboration process requires communication, planning, and reasoning among multiple intelligent agents. It has many applications in domains such as robotics, gaming (Wang et al., 2023a), and social simulation (Li et al., 2023).

There are increasing interests in developing multi-agent systems using LLMs. MindAgent introduces the CuisineWorld gaming scenario as a benchmark, utilizing the Collaboration Score (CoS) to measure the efficiency of collaboration (Gong et al., 2023). AgentVerse organizes its framework into four essential stages: Expert Recruitment, Collaborative Decision-Making, Action Execution, and Evaluation, thereby effectively deploying multi-agent groups that outperform a single agent (Chen et al., 2023). MetaGPT, on the other hand, employs an assembly line approach, designating specific roles to agents and efficiently breaking down complex tasks into subtasks involving many agents working together (Hong et al., 2023). However, these multi-agent collaboration models either tend to restrict agents to parallel-executable subtasks each round, even when unnecessary or bind them to a fixed pipeline and task stage, overlooking complex task dependencies. This may cause issues for tasks that need both sequential and parallel execution, thus limiting model generality and scalability (Gong et al., 2023; Chen et al., 2023; Hong et al., 2023).

In this paper, we focus on multi-agent collaboration for problem-solving with complex dependencies. These dependencies can be of different types, such as spatial dependencies that constrain the locations

[1] https://github.com/cnsdqd-dyb/VillagerAgent

Findings of the Association for Computational Linguistics ACL 2024, pages 16290–16314
August 11-16, 2024 ©2024 Association for Computational Linguistics

of the sub-tasks, causal dependencies that affect the availability and effects of the sub-tasks, and temporal dependencies that impose constraints on the timing of the sub-tasks. It is crucial to understand and manage these dependencies for effective multi-agent collaboration, enabling the agents to reason about the long-term consequences of their actions and avoid potential conflicts.

First, we introduce VillagerBench, a new multi-agent benchmark in the Minecraft environment designed for the evaluation of complex dependencies (Figure 7). Some of the multi-agent research is being tested within the Overcooked-AI (Carroll et al., 2020). Nevertheless, due to limitations in the number of agents, scenario flexibility, and task diversity, there is a desire for more comprehensive frameworks to test multi-agent cooperation. Inspired by Voyager (Wang et al., 2023a), GITM (Zhu et al., 2023), and MindAgent (Gong et al., 2023), we construct a multi-agent and multi-task evaluation framework with greater degrees of freedom using Minecraft. Minecraft offers a rich and diverse set of tasks that can be used to benchmark and evaluate multi-agent systems, such as building and farming. It allows players to explore dynamic environments that pose various challenges for multi-agent collaboration, such as resource allocation, task decomposition, and coordination. Specifically, we introduce three tasks, i.e., Construction Cooperation, Farm-to-Table Cooking and Escape Room Challenge. The Construction Cooperation task tests agents' aptitude for understanding task requirements and orchestrating team workload, focusing on the evaluation of spatial dependencies in multi-agent collaboration. The Farm-to-Table Cooking task assesses their agility in adapting to fluctuating environmental conditions, aiming to solve complex causal dependencies. The Escape Room Challenge task tests agents on their ability to execute tasks both sequentially and in parallel, requiring the reasoning of temporal dependencies and the ability to synchronize actions.

Second, we introduce a Directed Acyclic Graph Multi-Agent framework (VillagerAgent) to tackle complex dependencies in multi-agent collaborations. Each subtask is represented as a graph node in the DAG. We dynamically adjust the graph structure and the agent roles according to the environment and the agent states. VillagerAgent consists of task decomposer, agent controller, state manager and base agents. The Task Decomposer generate a Directed Acyclic Graph (DAG) of subtask nodes each round, while the Agent Controller oversees the assignment of these subtasks to the Base Agents for execution and self-reflection. Meanwhile, the State Manager is responsible for maintaining the status information of both the environment and the agents.

We quantitatively evaluate our method on VillagerBench. We demonstrate the superior performance of VillagerAgent over AgentVerse (Chen et al., 2023) by fewer hallucinations and enhancing the effectiveness of task decomposition.

2 VillagerBench Design

Our VillagerBench uses Mineflayer (PrismarineJS, 2013) to establish Agent APIs, offering a platform to examine cooperative behaviors in multi-agent systems via tasks such as construction, cooking, and escape room challenges (Figure 1).

We evaluate multi-agent systems powered by LLMs using three key metrics: **Completion (C)** that measures the average task completion rate; **Efficiency (E)** that assesses the speed of task execution and the utilization of resources; and **Balance (B)** that examines the distribution of workload among agents, with higher values indicating a more equitable assignment of tasks. Further details can be found in Appendix A.

Construction Cooperation Task: Interpretation and Allocation. Construction Cooperation task is centered around the agents' proficiency in interpreting detailed task documents and efficiently allocating the workload among team members. This task necessitates a high level of comprehension and coordination, as agents must parse the project specifications and judiciously assign sub-tasks to optimize collective performance.

Agents are provided with textual architectural blueprints that specify the positions and orientations of blocks required for construction tasks. Building materials are supplied in chests or at a material factory, where agents must mine and transport them to the building site. Further details can be found in Appendix B.1.

Farm-to-Table Cooking Task: Environmental Variability and Strategic Flexibility. In Farm-to-Table Cooking task, agents must adapt their strategies to changing environmental conditions and varying difficulty levels. They need to gather information, source ingredients either from contain-

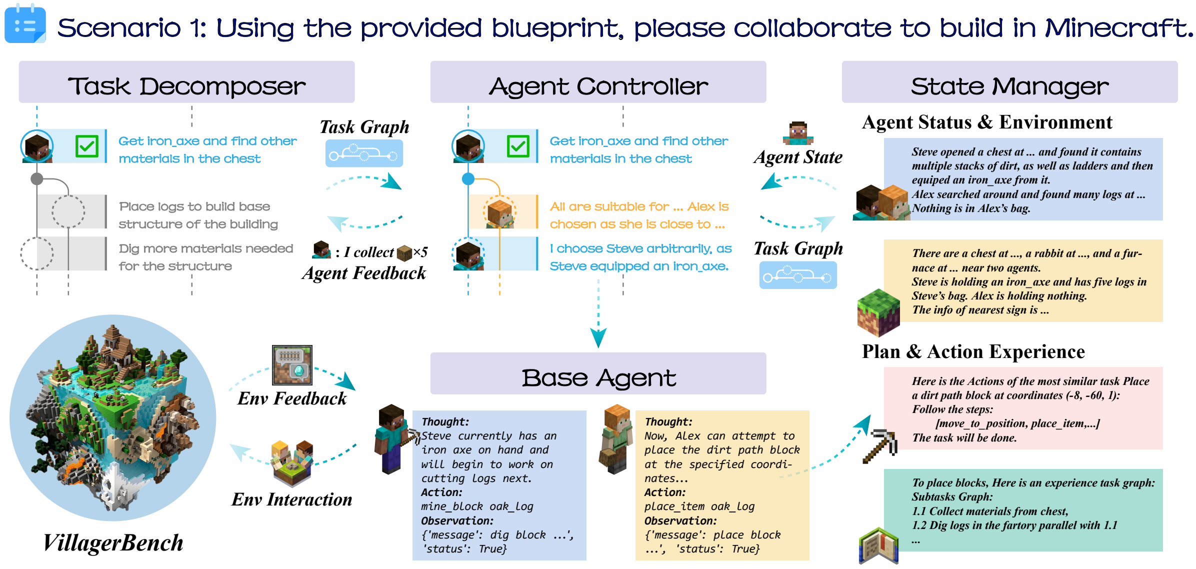

Figure 2: Overview of the VillagerAgent framework. Our framework acts as the central architecture for individual agents, enhancing their collaborative capabilities. Featuring a Task Decomposer that generates subtask DAGs, an Agent Controller for task assignment, a State Manager for status updating, and Base Agents for task execution and self-assessment.

ers or through activities like harvesting and hunting, and adjust their methods to prepare complex dishes. We simulate this by having agents act as farmers who are tasked with making **cake** and **rabbit stew** in Minecraft. These recipes are recognized for their high complexity in terms of ingredient synthesis, making them challenging targets for the task. Further details can be found in Appendix B.2.

Escape Room Challenge Task: Synchronization and Sequential Execution. Escape Room Challenge task tests agents' ability to work together and perform actions in a precise order, focusing on synchronization and timing. Agents must navigate environments with objects that have specific activation requirements, and success depends on their coordinated timing and teamwork.

Each room offers unique challenges that demand effective team collaboration and strategic planning. For example, a basic task may require two agents to press switches at different locations simultaneously to open a door. Further details and visual representations of each scenario can be found in Appendix B.3.

3 VillagerAgent: A Directed Acyclic Graph Multi-Agent Framework

3.1 Overview

The VillagerAgent framework comprises four main components: Task Decomposer, Agent Controller, State Manager, and Base Agents. It operates by having the Task Decomposer generate a Directed Acyclic Graph (DAG) of subtask nodes each round, based on the current state, while the Agent Controller oversees the assignment of these subtasks to the Base Agents for execution and self-reflection. Meanwhile, the State Manager is responsible for maintaining the status information of both the environment and the agents (Figure 2).

Agent Notations. We denote each base agent as A_i and the corresponding agent state as S_i. The agent state is a textual representation that recursively summarizes the agent's actions, possessions, and the entities in the surrounding environment. Each agent has an action history (H_i) that consists of the last p actions. We assume that there are k agents in the game. The agent set can be represented as $\mathbb{A} = \{A_i | i = 1, \ldots, k\}$ and the agent state set is denoted as $\mathbb{S} = \{S_i | i = 1, \ldots, k\}$

Task Notations. We model the execution dependencies of a complex task with a graph of subtasks. Each subtask node N_j is represented by a quadruple, i.e, $(T_j, D_j, \mathbb{C}_j, F_j)$. T denotes the subtask description and D represents the data from documents related to the subtask. $\mathbb{C}$ represents the assigned agents that have been selected by the Task Manager from the base agent set $\mathbb{A}$. F denotes the execution feedback. We denote the set of subtask nodes as $\mathbb{N} = \{N_j | j = 1, \ldots, m\}$ where m is the number of subtask nodes.

3.2 Task Decomposer

The Task Decomposer is responsible for managing and constructing the directed graph G. The directed graph represents the concurrency of the subtasks. In this graph, each node $v_i \in V$ corresponds to a subtask N_i, and each directed edge (v_i, v_j) signifies that subtask N_i must be completed before commencing subtask N_j. Parallel execution of subtasks is permitted when there is no direct edge dictating the execution order between them. The details of constructing the directed graph G from the set of subtasks $\mathbb{N}$ can be found in Appendix A.1.

Subtask Set Update. The Task Decomposer is also used to update the subtask set $\mathbb{N}$. Given the goal task description T_g, the relevant environment state E queried from the State Manager, the agent state set $\mathbb{S}$, and the current nodes $\mathbb{N}$, the Task Decomposer generates a set of new subtask nodes $\mathbb{N}'$.

$$\mathbb{N}' = \text{TD}(E, T_g, \mathbb{S}, \mathbb{N})$$

$$\mathbb{N} = \mathbb{N}' \cup \mathbb{N}$$

During task decomposition, the Task Decomposer adopts a zero-shot chain-of-thought (CoT) approach (Wei et al., 2023). This method is integrated into the prompt, as Figure 9 illustrates, to guide the LLM in generating responses in JSON format, specify the index of the immediate predecessor for each subtask as needed and specify JSON path expressions for each subtask, referencing the provided data D. Subsequently, each subtask node will use these JSON path expressions to query the data related to its subtask.

3.3 Agent Controller

The Agent Controller focuses on analyzing the task graph and assigning the appropriate subtask to the right agent in an efficient manner.

Ready-to-Execute Tasks Identification. The Agent Controller identifies ready-to-execute task set $\mathbb{N}_{ready}$. It checks all unexecuted tasks, where tasks with no remaining dependencies will be added to the ready-to-execute task set $\mathbb{N}_{ready}$.

Subtask Allocation. Based on the environment state E, ready-to-execute nodes $\mathbb{N}_{ready}$, and the states of the agents $\mathbb{S}$, the Agent Controller determines the allocation of agents to subtasks:

$$\text{AC}(E, \mathbb{N}_{ready}, \mathbb{A}, \mathbb{S}) \rightarrow [(A_i, N_j), \ldots]$$

In this process, the Agent Controller (AC) queries LLM to pair tasks with agents. It anticipates a JSON-formatted response containing the indices of tasks and the identifiers of the selected agents. The Agent Controller initiates the execution of tasks by the designated agents simultaneously.

3.4 State Manager

The State Manager (SM) is used to update the agent states and the environment information.

Agent State Update. SM updates the agent state based on the agent's action history H_i:

$$S_i = \text{LLM}(prompt_a, S_i, H_i).$$

where $prompt_a$ is the agent state update prompt. The agent state S_i acts as a long-term memory, in contrast to the action history H_i, which serves as short-term memory.

Environment State Retrieval. The global environment state (I) is the union of the local environment state from each agent. The local environment state of agent A_i can be obtained via the library API, i.e., Env(A_i).
Given the task description T_g, the relevant environment state E can be retrieved from the global environment state (I):

$$E = \text{LLM}(prompt_e, T_g, I).$$

where $prompt_e$ is the environment state retrieval prompt. $prompt_a, prompt_e$ can be found in Appendix 12, 13.

3.5 Base Agent Architecture

Each base agent A_i is responsible for executing its assigned subtask node N_j. The states of the agents associated with the predecessor nodes of the current node N_j in DAG can be represented as $\mathbb{S}_{\text{selected}}$. This execution results in an updated temporal action history and generates feedback:

$$(H_i, F_j) = \text{Exec}(N_j, H_i, \mathbb{S}_{\text{selected}}, E)$$

Upon execution of the subtask node N_j, two processes occur within the agent A_i:

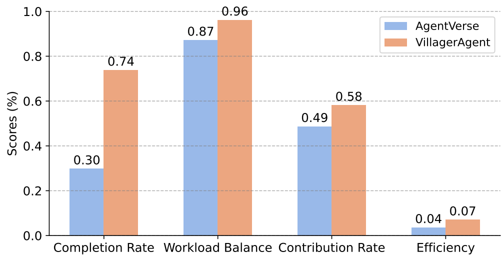

Figure 3: Comparison of VillagerAgent and AgentVerse on Farm-to-Table Cooking Task. VillagerAgent outperforms AgentVerse in Completion Rate (Chen et al., 2023).

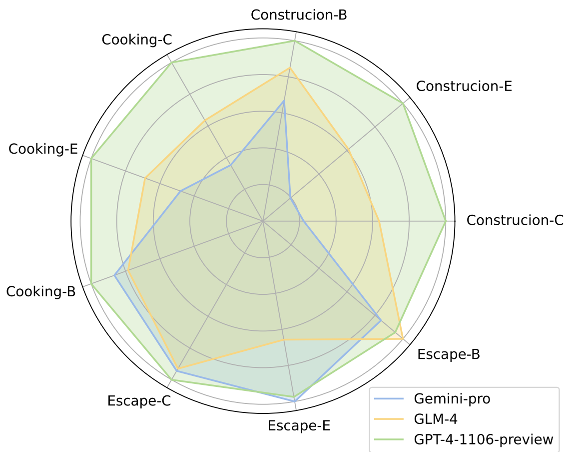

Figure 4: Comparison of LLMs on VillagerBench. We show the relative performance gap against the best in each scenario. GPT-4-1106-preview achieves higher scores across most metrics, whereas Gemini-Pro demonstrates better efficiency in the Escape Room Challenge.

ReAct Procedure. The Base Agent formulates a prompt that integrates its action history H_i, the current state of agents $\mathbb{S}_i$, the assigned subtask node N_j, and environmental data E provided by the State Manager. Utilizing the ReAct method, the agent iteratively generates actions and observations.(Yao et al., 2023) This iterative process is subject to a constraint of a maximum of 6 iterations or a total execution time limit of 120 seconds.

Self-Reflection. Upon completion of the task, the Base Agent updates the action history H_i and the task description T into a reflection prompt. LLM then generates a response that serves as feedback F_j for the subtask node N_j.

4 Experiments

LLM Capability Test. To rigorously evaluate the capabilities of LLMs, we conducted tests on the VillagerBench benchmark using the VillagerAgent framework based on three models: GPT-4-1106-preview(ope, 2023), Gemini Pro(gem, 2023), and GLM-4(Du et al., 2022). Our evaluation targeted three types of tasks: 100 Construction tasks, 100 Farm-to-table cooking tasks, and 25 Escape room challenges, each executed once. We terminate a testing round if the task execution exceeds the anticipated time frame or once the task has been successfully completed. The parameters for LLM reasoning can be found in Appendix 7.

Construction Cooperation Task. For the construction tasks ranging from 0 to 99, we deployed two agents, Alice and Bob, each equipped with essential APIs, to collaborate effectively. We intentionally omitted the requirement for agents to mine blocks from the material factory, considering the inherent complexity of the tasks. The blueprint provided to the agents is a more concise and readable format, thereby streamlining the context and facilitating more efficient task completion, as detailed in Appendix B.1.

Farm-to-Table Cooking Task. For the Farm-to-Table Cooking tasks, numbered 0 through 99. Tasks 0 to 35 are dedicated to cake-making, while tasks 36 to 99 focus on the preparation of rabbit stew. We supply cooking recipes to serve as a reference for the agents. **VillagerAgent vs. AgentVerse in Cooking**: We've transitioned AgentVerse BaseAgent from the Voyager environment (Wang et al., 2023a) to our VillagerBench BaseAgent, ensuring a fair comparison by preserving the prompt format and default settings, including the use of agent names Alice and Bob. Our modifications involve the adoption of the gpt-4-1106-preview language model, setting the temperature parameter to 0, and refining the feedback prompt to suit our ReAct Agent (Figure 16).

Escape Room Challenge Task. We've crafted 18 atom-based escape room tasks that simulate puzzle-solving scenarios for agents. Our generator constructs these tasks from the ground up, selecting appropriate atom tasks based on room attributes, required materials, and agent information, and then automatically scales them into full-fledged puzzles. The generator also ensures task feasibility by ac-

Models	Construction Task Avg. Score				Escape Challenge Avg. Score		
	C (%)	VHR (%)	E (%/min)	B (%)	C (%)	E (%/min)	B (%)
gemini-pro	8.12	13.83	0.76	63.74	69.2	**153.3**	80.35
glm-4	23.16	29.36	2.37	81.12	68.17	100.8	**95.3**
gpt-4-1106-preview	**36.45**	**49.05**	**3.88**	**95.38**	**73.29**	149.4	90.03
gpt-4-1106-preview (3-agents)	**52.17**	**61.02**	**6.26**	89.83	69.78	227.4	67.01

Table 1: GPT-4-1106-preview(ope, 2023), GLM-4(Du et al., 2022) and Gemini-Pro(gem, 2023) results on Construction Cooperation task and Escape Room Challenge Task. (The Escape Room Challenge Task updates to accommodate varying numbers of agents.)

counting for agent cooperation and item dependencies. For consistent LLM testing, we've designated seeds for each of the five difficulty levels, with 25 unique tasks in total, and set a default simultaneous item activation wait time of 30 seconds for task completion.

Benchmarking Our VillagerAgent in Overcooked-AI. We conducted tests on VillagerAgent (equipped with GPT-4) within the Overcooked environment, following the methodology used in ProAgent(Zhang et al., 2024). We analyzed the prompt tokens for each test. Consistent with the settings outlined in ProAgent, we evaluated each layout across 5 episodes, with the horizon set to 400.

Influence of Agent Quantity on Cooperative Task Execution. We analyzed how varying numbers of agents (1, 2, 4, 8) affect cooperative task performance in construction scenarios, specifically comparing the simplest task(task 0) and a complex task(task 64). Using the GPT-4-1106-preview(ope, 2023) model within the VillagerAgent framework, each task was repeated six times.

Assessing the Impact of Varied Agent Abilities on Cooperative Task Performance. We evaluate how different agent skill sets impact a complex farm-to-table cooking task (task 99 - rabbit stew preparation). With GPT-4-1106-preview(ope, 2023) as the base model, we tested two trios of agents, each set consisted of three agents: one with uniform API abilities (7 Base APIs plus SmeltingCooking, MineBlock, and AttackTarget) and another with diverse abilities (7 Base APIs with one unique additional API per agent). Each repeated six times.

4.1 Evaluation Metrics

Completion Rate (C). For each scenario, we monitor certain indicators that signify progress towards the scenario's objectives, such as blocks, ingredients or triggers. The completion rate is calculated based on the quantity of these indicators, providing a measure of how much of the scenario has been completed defined in AppendixA. The formula for calculating the completion rate is as follows:

$$\text{Completion (C)} = \frac{\text{\# Indicators Detected}}{\text{\# Total Indicators Expected}}$$

Efficiency of Completion (E). It is defined as the ratio of the task completion rate to the actual time taken by the agents. The efficiency of completion is computed as follows:

$$\text{Efficiency (E)} = \frac{\text{\# Task Completion Rate}}{\text{\# Total Execution Time}}$$

Balanced Agent Utilization Score (B). This metric assesses the distribution of workload among agents, aiming for a balanced utilization where each agent's active running time is similar. The ideal state is one where no single agent is either overburdened or underutilized.

$$\mathbf{t}' = \frac{\mathbf{t} - \min(\mathbf{t})}{\max(\mathbf{t}) - \min(\mathbf{t})} \tag{1}$$

$$\text{Balance(B)} = 1 - \sigma(\mathbf{t}') \tag{2}$$

Here, n is the number of agents, $\mathbf{t} \in \mathbb{R}^n$, $\mathbf{t}_i$ represents the active running time of agent i, and $\bar{\mathbf{t}}$ is the average active running time across all agents.

Block Placement View Hit Rate (VHR). evaluates the structural integrity and visual coherence of the construction from multiple vantage points. It is calculated as the intersection over union (IoU) of the constructed structure with the expected structure across a predefined set of viewpoints.

$$S_{vhr} = \frac{1}{V}\sum_{v=1}^{V} IoU(C_{v_{(\theta,\phi)}}, E_{v_{(\theta,\phi)}}) \tag{3}$$

Here, V is the number of viewpoints, C_v is the construction as seen from viewpoint v, and E_v is the expected view from viewpoint v.

Agent Type	Farm-to-Table Cooking Avg. Score			
	C (%)	ACR(%)	E (%/min)	B (%)
Same	**56.67**	**60.22**	**3.91**	**95.47**
Diverse	36.67	30.46	2.87	92.2

Table 2: Results of varied agent abilities on cooperative task performance on Farm-to-Table Cooking Task 99.

Models	Cooking Task Avg. Score			
	C (%)	ACR	E (%/min)	B (%)
AgentVerse gpt	29.75	48.64	3.54	87.13
VillagerAgent gemini	26.05	32.92	3.35	83.15
VillagerAgent glm	46.84	54.07	4.79	75.46
VillagerAgent gpt (2-agents)	73.75	**58.11**	6.98	**96.13**
VillagerAgent gpt (3-agents)	**85.26**	55.60	**21.90**	84.38

Table 3: Performance comparison between AgentVerse(Chen et al., 2023) and VillagerAgent on the Farm-to-Table Task. Note that gpt refers to GPT-4-1106-preview, gemini to Gemini-Pro, and glm to GLM-4

Agent Contribution Rate (ACR). quantifies the contribution of each agent in a Minecraft game based on the items they have crafted in farm-to-table cooking tasks. The specific definitions can be found in Appendix A.

4.2 Evaluation Results

GPT-4 with VillagerAgent Achieves Optimal Performance. Across the board, GPT-4-1106-preview, when integrated with VillagerAgent, consistently delivered the highest completion scores in task allocation (Figure 3), as seen in Construction, Escape Room Tasks and Farm-to-Table Cooking (Table 1, 3). It demonstrated a superior understanding of task requirements and agent management, outperforming GLM-4 and Gemini-Pro in View Hit Rate (VHR) and Agent Contribution Rate (ACR).

Gemini-Pro Excels in Efficiency for Escape Room Challenge. In the context of less complex tasks that prioritize timing and sequence, such as the Escape Room Tasks, Gemini-Pro showcased its strengths. It achieved efficiency comparable to GLM-4 and, in some cases, outperformed others due to its faster inference and response times, leading to a high-efficiency rating (Table 1).

VillagerAgent Outperforms AgentVerse. Despite both utilizing GPT-4, VillagerAgent outperforms AgentVerse in the Farm-to-Table Cooking Tasks(Figure 3), showing less hallucinatory behavior and a lower failure rate (18.2% for VillagerAgent vs. 44.4% for AgentVerse, as seen in Figure 5). Although VillagerAgent uses more tokens on average (126 vs. 107 for AgentVerse), it achieves a significantly lower Token Cost (Avg. 1.79 vs. 10.3 for AgentVerse), indicating a more efficient use of resources for higher scores, as detailed in Table 4.

Agent Collaboration and Performance Dynamics. Data analysis from Table 6 shows that VillagerAgent's task performance improves with additional agents up to a point, after which it declines. Initially, more agents contribute positively, enhancing task handling through collective capability. However, as agent numbers increase further, performance gains diminish due to issues like resource competition and increased management complexity for the LLM. The relationship between agent count and performance is thus characterized by a peak at moderate levels of collaboration, suggesting an optimal balance for system efficiency without specifying a precise range.

Diverse Abilities Hinder Coordination. The analysis of Table 2 reveals that a trio of agents with distinct extra APIs underperforms in all evaluated metrics. This underperformance is attributed to the increased complexity in coordination when agents possess different capabilities. For example, the workflow may be disrupted if one agent's task depends on the completion of another's, leading to potential bottlenecks and task failure.
Despite the lower efficiency, the diverse skill set among agents introduces a richer complexity to the task environment, paving the way for more intricate cooperative interactions. While not optimal for score maximization, this setup serves as a fertile ground for investigating advanced collaborative behaviors and strategies within our benchmark framework.

Trade-off in Token Cost. To measure the relationship between task completion performance and token usage, we introduce the following formula to calculate the token cost.

$$Cost = \frac{CompletionTokens}{(Score + \varepsilon) + ActionNum} \quad (4)$$

Completion Tokens refers to the average token usage to complete each action. Action Num refers to the number of valid actions executed during the

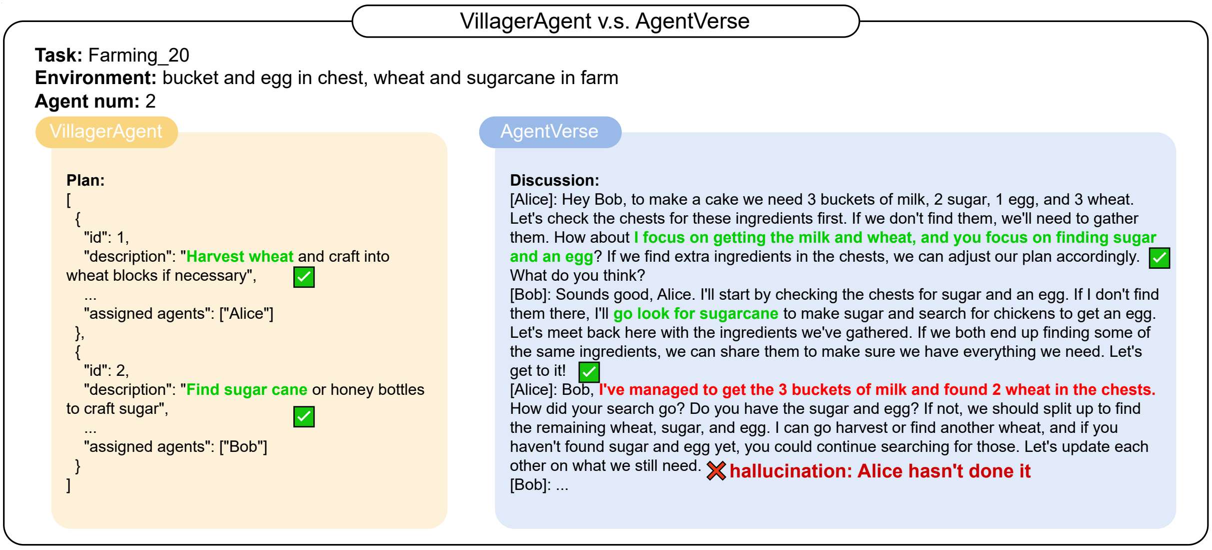

Figure 5: VillagerAgent v.s. AgentVerse(Chen et al., 2023) on Farm-to-Table Task. Hallucination exists in agent discussion stage of AgentVerse.

task. The score is the task score. We set epsilon=1 to prevent a score from dropping to zero.
We compute the Token Cost of VillagerAgent and AgentVerse on tasks of varying difficulty.

Difficulty	Framework	Tokens ↓	Cost ↓
Easy	AgentVerse	109.26	17.76
Medium	AgentVerse	108.95	6.05
Hard	AgentVerse	101.33	4.32
Average	AgentVerse	107.17	10.30
Easy	VillagerAgent	122.33	1.73
Medium	VillagerAgent	126.52	1.69
Hard	VillagerAgent	131.13	2.01
Average	VillagerAgent	126.00	1.79

Table 4: Comparison of Trade-off in Token Cost

We discovered that, although we use slightly more tokens per action compared to AgentVerse, our Token Cost is significantly lower. This indicates that the benefits we gain in terms of score outweigh the additional tokens we utilize.

4.3 In Overcooked-AI

VillagerAgent Outperforms ProAgent. In Overcooked-AI 6, our VillagerAgent (w/gpt-4), surpasses ProAgent(Zhang et al., 2024) across all five scenarios: Cramped Room, Asymmetric Advantages, Coordination Ring, Forced Coordination, and Counter Circuit. Each scenario tests different aspects of cooperative strategy and efficiency in a shared task environment. Notably excelling in the Forced Coordination scenario—a highly interdependent task requiring material sharing in confined spaces. This superior performance is attributed to our use of directed acyclic graphs for task management, enhancing efficiency in complex cooperative tasks, as detailed in Table 5.

Efficiency and Transferability of Prompts. we compare ProAgent and VillagerAgent regarding the use of prompt. The results are shown in the table below. Our framework utilized a single set of prompts to accomplish five tasks in Overcooked, whereas ProAgent employed five specific sets of prompts. Similarly, we also used a single set of prompts across three scenarios in VillagerBench. We also observe that VillagerAgent uses fewer tokens in each test, implying its lower overhead and better transferability.

5 Related Work

Minecraft Agents. Minecraft agents are intelligent programs that can perform various tasks within the Minecraft world. Recently, researchers have come to be aware of the extraordinary general planning ability of LLMs (Huang et al., 2022a). Many works (Huang et al., 2022b; Yuan et al., 2023; Wang et al., 2023c,a; Zhu et al., 2023) have leveraged LLMs for enhancing the high-level planning ability of Minecraft agents. Inner Monologue (Huang et al., 2022b) leveraged environment feedback to improve the planning ability of LLM. Voyager (Wang et al., 2023a) developed a skill library of executable code for storing and retrieving behav-

Figure 6: Overcooked-AI Scenarios

Layout	PBT	FCP	MEP	COLE	ProAgent	VillagerAgent (ours)
Cramped Room	178.8 ± 16.5	196.3 ± 16.8	185 ± 15	163.8 ± 24.1	197.3 ± 6.1	**213.3 ± 9.43**
Asymmetric Advantages	182.2 ± 27.9	185.7 ± 22.7	155.7 ± 63.9	201.3 ± 34.5	228.7 ± 23	**304 ± 8.76**
Coordination Ring	141.3 ± 28	148.8 ± 19.4	167.2 ± 22.4	168.8 ± 26.1	175.3 ± 29	**226.7 ± 18.9**
Forced Coordination	15.3 ± 17.1	44.7 ± 36.4	23.3 ± 19.8	24 ± 21.8	49.7 ± 33.1	**120 ± 16.97**
Counter Circuit	64.7 ± 45.9	58.3 ± 37.5	74.3 ± 39.1	95.5 ± 25.2	126.3 ± 32.3	**148 ± 4.38**

Table 5: Performance comparison of VillagerAgent and ProAgent across different scenarios in Overcooked-AI.

iors. The base agent in our VillagerAgent framework accounts for the states of other agents and features a modular design, enabling it to function independently or in collaboration with other agents.

MultiAgent Frameworks. MultiAgent frameworks are increasingly leveraging LLMs due to their potential in complex system development (Qian et al., 2023b,a; Xie et al., 2023; Wu et al., 2023a). CAMEL utilizes role-play to reduce hallucinations and improve collaboration (Li et al., 2023). MindAgent's CuisineWorld uses a Collaboration Score to gauge team efficiency (Gong et al., 2023). DEPS further extended this closed-loop interaction by introducing description, explainer and selector (Wang et al., 2023c). AgentVerse structures its system into recruitment, decision-making, execution, and evaluation, optimizing group performance (Chen et al., 2023). MetaGPT adopts an assembly line method, assigning roles to streamline task completion (Hong et al., 2023). However, these frameworks often face limitations in task flexibility and scalability(Gong et al., 2023; Chen et al., 2023; Hong et al., 2023). Our VillagerAgent framework improves collaborative efficiency for complex tasks by modeling task graphs.

LLM-as-Agent Benchmarks. Recent studies highlight the potential of Large Language Models (LLMs) as agents capable of tool use (Wang et al., 2023b; Xi et al., 2023). Emerging benchmarks aim to rigorously evaluate these models' performance (Liu et al., 2023; Xu et al., 2023; Carroll et al., 2020; Huang et al., 2023; Wu et al., 2023b; Ruan et al., 2023). MCU(Lin et al., 2023) discusses a method that uses atomic tasks as basic components to create a wide range of tasks (SkillForge). Our research focuses on multi-agent, enhances task complexity using Minecraft commands, introduces more intricate challenges like long-distance switch activation. The Overcooked environment is notable for coordination experiments (Carroll et al., 2020), while MAgIC focuses on assessing LLMs' cognitive and collaborative abilities in text-based multi-agent settings (Xu et al., 2023). Existing benchmarks, however, may not fully capture the capabilities of LLMs as multi-agents. Inspired by multiple single-agent studies conducted within Minecraft.(Huang et al., 2022b; Yuan et al., 2023; Wang et al., 2023c,a; Zhu et al., 2023) Our VillagerBench leverages Minecraft's API to create domains that mimic real-world tasks, facilitating multi-agent system evaluation and research advancement.

6 Conclusion

In this study, we introduce VillagerBench, a Minecraft multi-agent benchmark platform. We design three distinct scenarios within VillagerBench to evaluate collaborative tasks, aiming to assess the performance of our VillagerAgent framework. Our framework employs Directed Acyclic Graphs (DAG) to decompose tasks, enabling efficient and coordinated execution by agents. We benchmark the coordination skills of three LLMs using these metrics and demonstrate the effectiveness of our VillagerAgent framework. We also explore how agent count and capability diversity impact framework performance.

Limitations

Our VillagerAgent framework, while improving performance within the Minecraft multi-agent benchmark (VillagerBench), encounters a low overall task completion rate. This is partly due to the inherent complexity of the benchmark, which necessitates the use of a wide array of APIs, thereby enlarging the exploration space and complicating the execution of tasks, especially when agents have varied abilities.

One of the primary challenges is managing agents with varying capabilities, as it necessitates advanced coordination and balancing strategies to ensure effective teamwork. Our framework's performance diminishes when scaling beyond eight agents, suggesting issues with resource allocation and inter-agent communication efficiency. This decline could be attributed to the increased context length and the complexity of generating task graphs for a larger number of agents, analogous to a leader struggling to manage an excessive number of workers.

Challenges in Practice. If our work were to be applied in real-world settings, we anticipate facing challenges such as the complexity of processing dynamic information, variability in agent failures, and issues with the interpretability of Language Learning Models' outputs. Addressing these challenges may become one of the key directions for further research.

Acknowledgements

This work is supported by the National Science and Technology Major Project (2022ZD0117802). This work is also supported by the General Program of the National Natural Science Foundation of China (62372403).

References

2023. Gemini: A family of highly capable multimodal models.

2023. Gpt-4 technical report.

Micah Carroll, Rohin Shah, Mark K. Ho, Thomas L. Griffiths, Sanjit A. Seshia, Pieter Abbeel, and Anca Dragan. 2020. On the utility of learning about humans for human-ai coordination.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. Glm: General language model pretraining with autoregressive blank infilling.

Ran Gong, Qiuyuan Huang, Xiaojian Ma, Hoi Vo, Zane Durante, Yusuke Noda, Zilong Zheng, Song-Chun Zhu, Demetri Terzopoulos, Li Fei-Fei, and Jianfeng Gao. 2023. Mindagent: Emergent gaming interaction.

Sirui Hong, Mingchen Zhuge, Jonathan 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. 2023. Metagpt: Meta programming for a multi-agent collaborative framework.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. Benchmarking large language models as ai research agents.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022a. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. 2022b. Inner monologue: Embodied reasoning through planning with language models.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Haowei Lin, Zihao Wang, Jianzhu Ma, and Yitao Liang. 2023. Mcu: A task-centric framework for open-ended agent evaluation in minecraft.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv: 2308.03688*.

PrismarineJS. 2013. Prismarinejs/mineflayer: Create minecraft bots with a powerful, stable, and high level javascript api.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023a. Communicative agents for software development.

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2023b. Experiential co-learning of software-developing agents.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2023. Identifying the risks of lm agents with an lm-emulated sandbox.

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

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2023b. A survey on large language model based autonomous agents.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

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. 2023a. Autogen: Enabling next-gen llm applications via multi-agent conversation.

Yue Wu, Xuan Tang, Tom M. Mitchell, and Yuanzhi Li. 2023b. Smartplay: A benchmark for llms as intelligent agents.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. 2023. The rise and potential of large language model based agents: A survey.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. 2023. Openagents: An open platform for language agents in the wild.

Lin Xu, Zhiyuan Hu, Daquan Zhou, Hongyu Ren, Zhen Dong, Kurt Keutzer, See Kiong Ng, and Jiashi Feng. 2023. Magic: Benchmarking large language model powered multi-agent in cognition, adaptability, rationality and collaboration. *arXiv preprint arXiv: 2311.08562*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Haoqi Yuan, Chi Zhang, Hongcheng Wang, Feiyang Xie, Penglin Cai, Hao Dong, and Zongqing Lu. 2023. Skill reinforcement learning and planning for open-world long-horizon tasks.

Ceyao Zhang, Kaijie Yang, Siyi Hu, Zihao Wang, Guanghe Li, Yihang Sun, Cheng Zhang, Zhaowei Zhang, Anji Liu, Song-Chun Zhu, Xiaojun Chang, Junge Zhang, Feng Yin, Yitao Liang, and Yaodong Yang. 2024. Proagent: Building proactive cooperative agents with large language models.

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. 2023. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory.

A Metrics

A.1 Task Node Graph relevant algorithm

Convert subtask node set to Graph. Since LLMs are autoregressive, their outputs for subtasks often exhibit causal relationships. Leveraging this, we can assume that a given prompt suggests subsequent subtasks depend on or run concurrently with earlier ones, forming the basis for transforming them into a graph.

Task Decomposer construct graph using algorithm 1 to connect nodes representing subtasks:

1. Initialize the graph G with an empty set of vertices V, an empty set of edges E and the input list of subtask nodes L containing $N_1, N_2, \ldots, N_n$.
2. Iterate over each node N_i in the list L, where i ranges from 1 to n. Then add the current node N_i to the vertex set V.
3. Check if the current node N_i has predecessor nodes $P(N_i)$:
   - If N_i has predecessors, for each predecessor node p_j, add an edge from p_j to N_i to the edge set E.
   - If N_i does not have predecessors and $i > 1$, implying it may share predecessors with the previous node N_{i-1}, for each predecessor of N_{i-1}, p_k, add an edge from p_k to N_i to the edge set E.
4. Repeat steps 2 and 3 until all nodes in the list have been processed.

A.2 Construction Task Complete Rate (C)

Construction Task Complete Rate. quantifies the alignment of the constructed structure with the provided blueprint. It is defined as the ratio of correctly placed blocks to the total number of blocks specified by the blueprint. A higher C indicates a closer match to the intended design, reflecting the agents' ability to accurately interpret and execute the construction plan.

$$C = \frac{|P_{(x,y,z,\theta,\phi)} \cap B_{(x,y,z,\theta,\phi)}|}{|B_{(x,y,z,\theta,\phi)}|} \tag{5}$$

Here, P represents the set of placed blocks, and B represents the set of blocks in the blueprint. θ denotes facing and ϕ denotes axis.

Algorithm 1 Convert Task List to Graph

```
G ← (V, E) with V ← ∅, E ← ∅
L ← [N_1, N_2, ..., N_n]                      ▷ Input list
for i ← 1 to n do
    V ← V ∪ {N_i}                  ▷ Add element as a node
    if P(N_i) ≠ ∅ then
        for all p_j ∈ P(N_i) do
            E ← E ∪ {(p_j, N_i)}   ▷ Add edges from predecessors
        end for
    else if i > 1 then
        for all p_k ∈ P(N_{i−1}) do
            E ← E ∪ {(p_k, N_i)}   ▷ Share predecessors with previous element
        end for
    end if
end for
```

Algorithm 2 Find Ready-to-Execute Tasks

```
Require: G = (V, E)        ▷ Task graph with nodes and edges
Require: S ⊆ V             ▷ Set of successfully executed tasks
Require: U ⊆ V             ▷ Set of unexecuted tasks
R ← ∅                      ▷ Result set of ready-to-execute tasks
for all N_i ∈ U do
    P(N_i) ← {p_j | (p_j, N_i) ∈ E}    ▷ Find predecessors of N_i
    if P(N_i) = ∅ or P(N_i) ⊆ S then
        R ← R ∪ {N_i}      ▷ Add if no predecessors or all predecessors executed
    end if
end for
return R
```

Figure 7: Live demonstration of agents performing tasks in VillagerBench scenarios.

A.3 Construction Dependency Complexity (D)

$$D = \sum_{i=1}^{B} \left(\frac{1}{EP_i} + W_h(H_i - G) \right) + D_i \quad (6)$$

Here, EP represents the effective path of one block to place through the nearby blocks, B is the number of blocks, H is the height of the block, G is the ground height, and D is the block dig score if this block needs to be dug from the factory.

A.4 Farm-to-Table Cooking Completion Rate

Completion Rate (C) quantifies the level of task completion based on the materials acquired and the actions performed:

$$C = \sum_{i=1}^{n} S_{\text{raw}_i} + \sum_{j=1}^{m} S_{\text{action}_j} \quad (7)$$

Here, S_{raw_i} is the score of the i-th raw material and S_{action_j} is the score for the j-th action that contributes to task progress.

A.5 Farm-to-Table Agent Contribution Rate

Agent Contribution Rate (ACR). The contribution score for each agent with respect to a specific material is defined as follows:
The overall ACR for the task is then calculated by aggregating the contributions of all agents for all required materials:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (I_i - I_{avg})^2} \quad (8)$$

The cooperation level can then be calculated as:

$$S_{cc} = \left(1 - \frac{\sigma - \sigma_{min}}{\sigma_{max} - \sigma_{min}} \right) \quad (9)$$

Here, n is the number of agents, $\mathbf{I} \in \mathbb{R}^n$, I_i is the contribution of item agent i provides, and then we standardize the score.

A.6 Farm-to-Table Dependency Complexity

Farm-to-Table Cooking Dependency Complexity (D).

$$D = \sum_{i=1}^{n} m_i \times d_i \quad (10)$$

where m_i represents the direct materials required for crafting the target food item, and d_i denotes the number of processing steps required to obtain or synthesize the material m_i within the context of the task.
In this formulation, m_i is the quantity of each direct material, and d_i reflects the depth of the dependency chain for each material, indicating the complexity of the process needed to acquire it. The product of m_i and d_i for each material is summed to yield the overall dependency complexity of the cooking task.

A.7 Escape Room Challenge Completion Rate

Completion Rate (C).

$$C = \frac{\sum_{i=1}^{n} \left(\frac{c_i}{m} \times S_i \right)}{\sum_{i=1}^{n} S_i} \quad (11)$$

Here, n is the number of tasks, c_i is the number of conditions that have been met for task i, and S_i is the score obtained for task i.

Config	Construction Avg. Score			
	C (%)	VHR(%)	E (%/min)	B (%)
$\text{Task}_0$1p	**100**	**100**	12.96	-
$\text{Task}_0$2p	**100**	**100**	**17.75**	**93.09**
$\text{Task}_0$4p	**100**	**100**	17.41	81.64
$\text{Task}_0$8p	66.63	63.33	12.45	55.67
Task_{64}1p	35.25	36.25	1.92	-
Task_{64}2p	41.67	35.62	2.34	**90.77**
Task_{64}4p	**46.67**	**39.38**	**3.28**	88.91
Task_{64}8p	30.21	33.33	2.27	74.09

Table 6: Evaluation on task execution efficiency with different agent quantities. The Balanced Agent Utilization Score (B) is inapplicable for a single-player scenario.

A.8 Escape Room Challenge Dependency Complexity (D)

The Escape Room Challenge Dependency Complexity (D) is calculated recursively using a breadth-first search approach, starting from the exit. The complexity of each room is determined by the number of conditions that must be met to pass through it. The complexity for the entire challenge is the cumulative sum of the complexities of all rooms encountered during the search. The formula for calculating the dependency complexity (D) is as follows:

$$D = \sum_{i=1}^{n} c_i \tag{12}$$

where c_i represents the complexity of room i, which is the number of conditions required to pass that room. The sum is taken over all rooms n that are encountered in the breadth-first search from the exit to the entrance of the escape room challenge. This approach ensures that the overall complexity reflects the dependencies and requirements of each room within the context of the escape scenario.

B Task Illustrations

B.1 Construction with Blueprints

Task Description. In this task, participants are required to work collaboratively to construct a structure in the game Minecraft, following the provided blueprint. The participants have access to two chests: one chest contains a variety of building materials, while the other chest, located within the factory, contains tools. However, the tools are not necessary for the completion of this task. The objective is to accurately replicate the blueprint in the game environment, and the task is considered complete once the structure matches the blueprint specifications.

Given APIs. The following APIs are provided to facilitate the construction process within the game. These functions allow the agent to interact with the game world, such as placing and fetching blocks, navigating to specific locations:

```
Agent.placeBlock
Agent.fetchContainerContents
Agent.MineBlock
Agent.scanNearbyEntities
Agent.equipItem
Agent.navigateTo
Agent.withdrawItem
Agent.dismantleDirtLadder
Agent.erectDirtLadder
Agent.handoverBlock
```

Blueprint. The blueprint specifies the exact materials and their respective positions required to construct the structure. Each line in the blueprint represents a different component of the structure, detailing the type of material, its orientation, and the coordinates where it should be placed. The following is the blueprint that must be followed to complete the task:

```
"task_24": [
    "[material:grass_block facing: None
     positions:[start:[-9 -60 -1] end:...",
    "[material:oak_trapdoor facing:E
       positions:[[-8 -60 -1] [-8 -60 0]]
     material:oak_trapdoor facing:S ...]",
    "[material:oak_trapdoor facing:W
     positions:[[-10 -60 -1] [-10 -60 0]]",
    "[material:oak_trapdoor facing:N
       position:[-9 -60 -2]]",
    "[material:oxeye_daisy facing: None
       position:[-9 -59 0]]",
    "[material:poppy facing: None
       position:[-9 -59 -1]]",
    "[material:dandelion facing: None
       position:[-9 -59 1]]"
],
```

B.2 Farm-to-Table Cooking

Given APIs. The following APIs are available to assist participants in interacting with the virtual environment, which includes fetching contents from containers, mining blocks, scanning nearby entities, equipping items, cooking, navigating, withdrawing items, crafting, attacking targets, using items on entities, and transferring blocks:

```
Agent.fetchContainerContents
Agent.MineBlock
Agent.scanNearbyEntities
Agent.equipItem
Agent.SmeltingCooking
Agent.navigateTo
Agent.withdrawItem
Agent.craftBlock
Agent.attackTarget
Agent.UseItemOnEntity
Agent.handoverBlock
```

Recipes. The recipes detail the specific ingredients and quantities needed to craft the food items. Below is the recipe for crafting rabbit stew, which requires a combination of baked potato, cooked rabbit, a bowl, a carrot, and a brown mushroom:


```
{
    "result": {
        "name": "rabbit_stew",
        "count": 1
    },
    "ingredients": [
        {
            "name": "baked_potato",
            "count": 1
        },
        {
            "name": "cooked_rabbit",
            "count": 1
        },
        {
            "name": "bowl",
            "count": 1
        },
        {
            "name": "carrot",
            "count": 1
        },
        {
            "name": "brown_mushroom",
            "count": 1
        }
    ]
}
```


B.3 Escape Room

Task Description. Agents, you are presented with a cooperative multi-stage escape challenge. Each room, measuring 10x10, demands teamwork to decipher puzzles and navigate through impediments. It is important to note that agents may find themselves in separate rooms, where direct collaboration is not feasible. Despite these circumstances, it is imperative to utilize individual strengths and work collectively to advance. Successful completion of a task in one room will result in transportation to the subsequent room or will clear the path to proceed by foot. The rooms are arranged along the z-axis, with their centers spaced 10 units apart. The ultimate goal is to reach the exit located at coordinates (130, -60, -140). Communication, adaptation, and teamwork are essential to escape. We wish you the best of luck!

Given APIs. The following APIs are provided to assist agents in interacting with the environment, which includes placing and fetching blocks, mining, scanning nearby entities, equipping items, navigating, withdrawing items, toggling actions, and transferring blocks:

```
Agent.placeBlock
Agent.fetchContainerContents
Agent.MineBlock
Agent.scanNearbyEntities
Agent.equipItem
Agent.navigateTo
Agent.withdrawItem
Agent.ToggleAction
Agent.handoverBlock
```

Room Sign Hints. The escape room challenge provides hints through signs placed within each room. Agents can read the nearby sign text to gain clues for solving the room's puzzle. One such hint is as follows:

```
Step on all the pressure plates at the
same time to clear the stone blocks and
open the trapdoors for escape.

In each room the agent can get nearby
sign text. Around you, the key activated
blocks are: a oak_pressure_plate block
set at position [130, -60, 131] powered.
You have done the task in this room.

Move to x=130, y=-60, z=137 to continue.
You are at task room [130, -60, 131].
```

C Experiment Configuration

C.1 Context Length

Throughout the testing process, the total length of context tokens does not exceed 4,000, and the length of the subsequent text does not exceed 1,024 tokens. The configurations for the tests are as (Table 7)

D Qualitative Analysis

Within the AgentVerse framework, during the discussion phase, Alice exhibits clear hallucinations in the first round, mistakenly believing that she has already searched the chest and generated fictitious feedback. Based on this fabricated feedback, our provided BaseAgent Alice infers that she can hand over the bucket to Bob to complete the subsequent tasks. However, the bucket has not actually been collected. This process illustrates how hallucinations in AgentVerse can gradually escalate and impact the stability of the entire decision-making process. (Figure 8)
Our approach, VillagerAgent, employs centralized decision control and correctly generates sub-tasks such as collecting wheat and finding sugar during the Task Graph generation process by the Task Decomposer, issuing instructions for parallel execution.

E VillagerBench API Library

E.1 Movement and Navigation

scanNearbyEntities: Search for specific items or creatures within a radius.
navigateTo: Move to a specific coordinate location.
navigateToPlayer: Move to another player's location.
erectDirtLadder: Build a dirt ladder at a specified location to reach higher places.
dismantleDirtLadder: Dismantle a dirt ladder at a specified location.
layDirtBeam: Place a dirt beam from one position to another.
removeDirtBeam: Remove a dirt beam.

E.2 Combat and Interaction

attackTarget: Attack the nearest entity with a specific name.
UseItemOnEntity: Use a specific item on a specific entity.
talkTo: Talk to an entity.
handoverBlock: Hand over an item to another player.

E.3 Item Management

equipItem: Equip a specific item to a designated slot.
tossItem: Toss a specific amount of items.
withdrawItem: Withdraw items from a container.
storeItem: Store items in a container.
openContainer: Open the nearest container.
closeContainer: Close a container.
fetchContainerContents: Fetch details of specific items in a container.

E.4 Production and Crafting

MineBlock: Mine a block at a specific location.
placeBlock: Place a block at a specific location.
craftBlock: Craft items at a crafting table.
SmeltingCooking: Cook or smelt items in a furnace.
enchantItem: Enchant items at an enchanting table.
repairItem: Repair items at an anvil.
trade: Trade items with a villager.

E.5 Life Skills

sleep: Go to sleep.
wake: Wake up.
eat: Eat food.
drink: Drink a beverage.
wear: Wear an item in a specific slot.

E.6 Other Actions

ToggleAction: Operate a door, lever, or button.
get_entity_info: Get information about an entity.
get_environment_info: Get information about the environment.
performMovement: Perform actions like jump, move forward, move backward, turn left, turn right.
lookAt: Look at someone or something.
startFishing: Start fishing.
stopFishing: Stop fishing.
read: Read a book or sign.
readPage: Read a specific page of a book.
write: Write on a writable book or sign.

Model	Total Tokens	Output Tokens	Temperature	Other Defaults
GPT-4-1106-preview	128,000	4,096	0	Default
Gemini-Pro	30,720	2,048	0	Default
GLM-4	128,000	> 1,024	0.01	Default

Table 7: Configuration of models used in the experiment.

F VillagerBench Scenario Examples

Here we present live demonstrations of two agents performing an escape room challenge, three agents executing a farm-to-table cooking task, and four agents engaged in a construction task. (Figure 7)

G Prompts

G.1 Task Decomposer

The Task Decomposer utilizes template 9 and template 10 in VillagerBench.

G.2 Agent Controller

Template 11 is used for the Agent Controller.

G.3 State Manager

The State Manager employs the Agent State Summary template 12 and the Environment Summary template 13.

G.4 Base Agent

The Base Agent uses the Execution template 14 and the Reflect template 15.

G.5 AgentVerse Prompt

The configuration for AgentVerse is defined in template 16.

G.6 Task Decompose Prompt (Overcooked-AI)

The Decompose Prompt for the Overcooked-AI Benchmark is outlined in template 17.

VillagerAgent v.s. AgentVerse

Task: Farming_20
Environment: bucket and egg in chest, wheat and sugarcane in farm
Agent num: 2

VillagerAgent

Plan:

```
[
  {
    "id": 1,
    "description": "Harvest wheat and craft into wheat blocks if necessary",
    "milestones": [
      "Navigate to wheat at [45, -59, 129] and [45, -59, 131]",
      "Harvest a total of 3 wheat",
      "Craft wheat into wheat blocks if less than 3 wheat is harvested"
    ],
    "retrieval paths": [
      "~/meta-data/ingredients/3"
    ],
    "required subtasks": [],
    "assigned agents": ["Alice"]
  },
  {
    "id": 2,
    "description": "Find sugar cane or honey bottles to craft sugar",
    "milestones": [
      "Scan for sugar cane or honey bottles in the environment or chests",
      "Navigate to the location of sugar cane or honey bottles",
      "Collect or withdraw 2 sugar canes or honey bottles",
      "Craft 2 sugars from the collected items"
    ],
    "retrieval paths": [
      "~/meta-data/ingredients/1",
      "~/meta-data/ingredients/2"
    ],
    "required subtasks": [],
    "assigned agents": ["Bob"]
  }
]
```

Action:
[...]

AgentVerse

Discussion:
[Alice]: Hey Bob, to make a cake we need 3 buckets of milk, 2 sugar, 1 egg, and 3 wheat. Let's check the chests for these ingredients first. If we don't find them, we'll need to gather them. How about **I focus on getting the milk and wheat, and you focus on finding sugar and an egg**? If we find extra ingredients in the chests, we can adjust our plan accordingly. What do you think?
[Bob]: Sounds good, Alice. I'll start by checking the chests for sugar and an egg. If I don't find them there, I'll **go look for sugarcane** to make sugar and search for chickens to get an egg. Let's meet back here with the ingredients we've gathered. If we both end up finding some of the same ingredients, we can share them to make sure we have everything we need. Let's get to it!
[Alice]: Bob, **I've managed to get the 3 buckets of milk and found 2 wheat in the chests.** How did your search go? Do you have the sugar and egg? If not, we should split up to find the remaining wheat, sugar, and egg. I can go harvest or find another wheat, and if you haven't found sugar and egg yet, you could continue searching for those. Let's update each other on what we still need. **hallucination: Alice hasn't done it**
[Bob]: ...

Action:
[...]

Discussion:
[Alice]: Bob, great work on finding the sugar cane and the egg. Since you've got the sugar cane, could you please craft the 2 additional sugar we need? Also, since you found **iron ingots**, please craft the third bucket. I'll go ahead and collect the 3 wheat from the farm now. Once we have all the ingredients, we can start making the cake. Let me know once you've crafted the sugar and the bucket. **hallucination: no iron ingots in chest**
[Bob]: Alice, I'm on it. I'll craft the 2 additional sugar from the sugar cane and then use the iron ingots to craft the third bucket. After that, I'll have all the items we need from my side. I'll wait for you to collect the wheat, and then we can combine our ingredients to make the cake. I'll let you know as soon as I've finished crafting. Let's keep up the good work!
[Alice]: ...
[Bob]: ...

Action:
[...]

```
{
  "action": "handoverBlock",
  "action_input": {
  "item_name": "milk_bucket",
  "item_count": 3,
   ...
  }
}
```

Observation: {'message': "**Bob don't have enough milk_bucket in inventory**", 'status': False}

Figure 8: VillagerAgent v.s. AgentVerse: The hallucination in AgentVerse began at the discussion stage and eventually extended its influence to the execution stage.

Task Decompose Prompt

SYSTEM

Your current mission is to leader all the players and execute a set of specified tasks within the Minecraft environment.
--- Background Information ---
Our system manages the task as a Directed Acyclic Graph (DAG).
In this turn, you need to decompose the tasks and arrange them in chronological order. Next turn we will analyse your result json to a graph.

A subtask-structure has the following json component:
{
"id": int, id of the subtask start from 1,
"description": string, description of the subtask, more detail than a name, for example, place block need position and facing, craft or collect items need the number of items.
"milestones": list[string]. Make it detailed and specific,
"retrieval paths": list[string], [~/...] task data is a dict or list, please give the relative path to the data, for example, if the data useful is {"c": 1} dict is {"meta-data": {"blueprint": [{"c": 1},]}}, the retrieval path is "~/meta-data/blueprint/0",
"required subtasks": list[int], if this subtask is directly prerequisite for other subtasks before it, list the subtask id here.
"candidate agents": list[string], name of agents. dispatch the subtask to the agents.
}

*** Important Notice ***
- The system do not allow agents communicate with each other, so you need to make sure the subtasks are independent.
- Sub-task Dispatch: Post decomposition, the next step is to distribute the sub-tasks amongst yourselves. This will require further communication, where you consider each player's skills, resources, and availability. Ensure the dispatch facilitates smooth, ** parallel ** execution.
- Task Decomposition: These sub-tasks should be small, specific, and executable with MineFlayer code, as you will be using MineFlayer to play MineCraft. The task decomposition will not be a one-time process but an iterative one. At regular intervals during playing the game, agents will be paused and you will plan again based on their progress. You'll propose new sub-tasks that respond to the current circumstances. So you don't need to plan far ahead, but make sure your proposed sub-tasks are small, simple and achievable, to ensure smooth progression. Each sub-task should contribute to the completion of the overall task. That means, the number of sub-tasks should no more than numbers of agents. When necessary, the sub-tasks can be identical for faster task accomplishment. Be specific for the sub-tasks, for example, make sure to specify how many materials are needed.
- In Minecraft, item can be put in agent's inventory, chest, or on the ground. You can use the item in agent's inventory or chest, but you can not use the item on the ground unless you dig it up first.
- The block at lower place should be placed first, and the block at higher place should be placed later. [x,-60,z] is the lowest place. For example, if a task is placing block at x -57 z, then y -60, -59 and -58 should be placed first and in order.
- Integration and Finalization: In some tasks, you will need to integrate your individual efforts. For example, when crafting complicated stuff that require various materials, after collecting them, you need to consolidate all the materials with one of players.
- You can stop to generate the subtask-structure json if you think the task need the information from the environment, and you can not get the information from the environment now.

USER

This is not the first time you are handling the task, so you should give part of decompose subtask-structure json feedback. Here is the query:
"""
the environment information around:
{env}

The high-level task:
{task}

Agent ability: (This is just telling you what the agent can do in one step, subtask should be harder than one step)
{agent_ability}
"""
Your response should exclusively include the identified sub-task or the next step intended for the agent to execute.
So, {num} subtasks is the maximum number of subtasks you can give.
Response should contain a list of subtask-structure JSON.

Figure 9: Task Decomposer Prompt Template

Redecompose Prompt

SYSTEM

Your current mission is to leader all the players and execute a set of specified tasks within the Minecraft environment.
--- Background Information ---
Our system manages the task as a Directed Acyclic Graph (DAG).
In this turn, you need to decompose the tasks and arrange them in chronological order. Next turn we will analyse your result json to a graph.

A subtask-structure has the following json component:
{
"id": int, id of the subtask start from 1,
"description": string, description of the subtask, more detail than a name, for example, place block need position and facing, craft or collect items need the number of items.
"milestones": list[string]. Make it detailed and specific,
"retrieval paths": list[string], [~/...] task data is a dict or list, please give the relative path to the data, for example, if the data useful is {"c": 1} dict is {"meta-data": {"blueprint": [{"c": 1},]}}, the retrieval path is "~/meta-data/blueprint/0",
"required subtasks": list[int], if this subtask is directly prerequisite for other subtasks before it, list the subtask id here.
"candidate agents": list[string], name of agents. dispatch the subtask to the agents.
}

*** Important Notice ***
- The system do not allow agents communicate with each other, so you need to make sure the subtasks are independent.
- Sub-task Dispatch: Post decomposition, the next step is to distribute the sub-tasks amongst yourselves. This will require further communication, where you consider each player's skills, resources, and availability. Ensure the dispatch facilitates smooth, ** parallel ** execution.
- Task Decomposition: These sub-tasks should be small, specific, and executable with MineFlayer code, as you will be using MineFlayer to play MineCraft. The task decomposition will not be a one-time process but an iterative one. At regular intervals during playing the game, agents will be paused and you will plan again based on their progress. You'll propose new sub-tasks that respond to the current circumstances. So you don't need to plan far ahead, but make sure your proposed sub-tasks are small, simple and achievable, to ensure smooth progression. Each sub-task should contribute to the completion of the overall task. That means, the number of sub-tasks should no more than numbers of agents. When necessary, the sub-tasks can be identical for faster task accomplishment. Be specific for the sub-tasks, for example, make sure to specify how many materials are needed.
- In Minecraft, item can be put in agent's inventory, chest, or on the ground. You can use the item in agent's inventory or chest, but you can not use the item on the ground unless you dig it up first.
- The block at lower place should be placed first, and the block at higher place should be placed later. [x,-60,z] is the lowest place. For example, if a task is placing block at x -57 z, then y -60, -59 and -58 should be placed first and in order.
- Integration and Finalization: In some tasks, you will need to integrate your individual efforts. For example, when crafting complicated stuff that require various materials, after collecting them, you need to consolidate all the materials with one of players.
- You can stop to generate the subtask-structure json if you think the task need the information from the environment, and you can not get the information from the environment now.

USER

This is not the first time you are handling the task, so you should give a decompose subtask-structure json feedback. Here is the query:
"""
the environment information around:
{env}

agent state:
{agent_state}

success previous subtask tracking:
{success_previous_subtask}

failure previous subtask tracking:
{failure_previous_subtask}

Agent ability: (This is just telling you what the agent can do in one step, subtask should be harder than one step)
{agent_ability}

The high-level task
{task}
"""
Your response should exclusively include the identified sub-task or the next step intended for the agent to execute.
So, {num} subtasks is the maximum number of subtasks you can give.
Response should contain a list of subtask-structure JSON.

Figure 10: Task REDecompose Prompt Template

Controller Prompt

SYSTEM

You are the Global Controller for Minecraft game agents. Your task is to assign tasks to agents. Create a plan that assigns tasks to suitable agents and return a list of task-assignment JSON objects.

USER

Background Information:

Your objective is to select tasks and allocate them to appropriate agents based on specific criteria. Each task requires a set number of agents for completion, as indicated by the task's "number." Only agents listed as candidates for a task are eligible to perform it. It's crucial to ensure that no agent is assigned to more than one task at any given time.

When assigning tasks, consider the following factors:

1. **Agent's Current State:** This includes the agent's location, items in possession, health status, etc.
2. **Task Requirements:** Necessary items, task location, and other specific needs.
3. **Agent's Experience:** Previous tasks completed and overall performance history.
4. **Agent's Abilities:** Skills and capabilities relevant to the task.

Resources Provided:

- **Minecraft Game Environment:** `{env}`
- **Agent Experience Records:** `{experience}`
- **Current Agent States:** `{agent state}`
- **List of Available Agents:** `{free agent}`
- **List of Tasks:** `{tasks}`

Assignment Objective:

You are to match tasks with suitable agents from the available list and produce a series of task-assignment JSON objects. The JSON format should be as follows:

```json
{
	"reason": "Explanation of the selection process, detailing why the agent is fit for the task based on their current state and held items.",
	"task_id": "The ID of the selected task.",
	"agent": "Names of agents assigned to the task."
}
```

Key Instructions:

- Provide a step-by-step reasoning for each task assignment.
- Ensure each task is assigned to the exact number of agents required, with all agents being from the task's candidate list.
- Aim to minimize the number of unassigned agents, adhering to the rules stated above.

Response Format:

Submit your response as a list of task-assignment JSON objects.

Figure 11: Agent Controller Prompt Template

Agent State Update Prompt

SYSTEM

You are a helpful assistant in Minecraft.

USER

You are {name}. Your task is to create a concise running summary of actions and information results in the provided text, focusing on key and potentially important information to remember.

You will receive the current summary and the your latest actions. Combine them, adding relevant key information from the latest development in 1st person past tense and keeping the summary concise.
The subject of the sentence should be {name}.

Summary So Far:
{summary_so_far}

Latest Development:
{latest_development}

Your Summary:

Figure 12: State Manager Agent State Update Prompt

Environment Summary Prompt(one shot)

SYSTEM

You are a helpful assistant in Minecraft.
Based on the environment info and the task, extract the key information and summarize the environment info in a concise and informative way.
You should focus on the entities, blocks, and creatures in the environment, and provide a summary of the environment info.

USER

The environment info:
{"person_info": [{"name": "Tom", "position": [-1, -59, 1], "held_items": {"spruce_planks": 1}}], "blocks_info": [{"spruce_planks": [-3, -60, 0]}, {"grass_block": [-2, -61, 0]}, {"chest": [-4, -60, 0], "facing": "W"}, {"oak Log": [-3, -61, 0]}, {"birch_slab": [-3, -60, -1]}, {"birch_slab": [-3, -60, 1]}, {"dirt": [-2, -62, 0]}, {"grass_block": [-2, -61, -1]}, {"grass_block": [-2, -61, 1]}, {"crafting_table": [-4, -60, -1]}, {"facing": "W", "furnace": [-4, -60, 1]}, {"stone_pressure_plate": [-3, -60, 2]}, {"juggle_button": [-3, -60, 3]}], "time": "sunrise"},
nearby_entities': [{'Alice': [42, -59, 125], 'other_entity': 'Alice'}, {'pig': [-3, -59, 0]}, {'pig': [-3, -59, 2]}]
*** The task *** : cook meat in the Minecraft.

The summary of the environment info:
Entity: Tom is located at position [-1, -59, 1] and is holding one spruce plank, Alice is located at position [42, -59, 125].
Blocks: a chest at [-4, -60, 0] facing west, a furnace at [-4, -60, 1] and other bloces.
Creatures: two pigs at [-3, -59, 0] and [-3, -59, 2].
Interactive-Items: a stone pressure plate at [-3, -60, 2], a juggle button at [-3, -60, 3].

The environment info:
{environment_info}
*** The task *** : {task}.
Return with Entity, Blocks, Creatures and Interactive-Items, and give all these position of these blocks and entities like chest, crafting table, furnace, animals, and plants.

Figure 13: State Manager Environment Summary Prompt

Agent Prompt

*** The relevant data of task(not environment data)***
{relevant_data}
*** Other agents team with you ***
{other_agents}
*** {agent_name}'s state ***
{agent_state}
*** The agent's actions in the last time segment partially ***
{agent_action_list}
*** environment ***
{env}
*** The minecraft knowledge card ***
{minecraft_knowledge_card}
*** The task description ***
====================
*** Task ***
{task_description}
*** milestone ***
{milestone_description}

At least two Action before the Final Answer.

Figure 14: Base Agent Execution Prompt

Reflect Prompt

SYSTEM

You are in a Minecraft world. You are a agent player. You need to use the action history compared with the task description and the milestone description to check whether the task is completed.
The check-strucutre
{
    "reasoning": str, # the reasoning process
    "summary": str, # the summary of the vital information of action history with detailed position number and other parameters, which not included in task description.
    "task_status": bool, # whether the task is completed
}

USER

Now you have tried to complete the task.
The task description is:
{task_description}

The milestone description is:
{milestone_description}

The action history is:
{state}
{action_history}

Please check whether the task is completed and return a check-strucutre json.

Figure 15: Base Agent Reflect Prompt

AgentVerse Config

YAML TEMPLATE

```yaml
prompts:
  prompt: &prompt |-
    # Role Description
    You are an experienced MineCraft player. ${role_description}

    Your current mission is to team up with other players and execute a set of specified tasks within the Minecraft environment.

    # Requirements
    It is essential that you effectively coordinate with other players to ensure the successful completion of tasks in a highly efficient manner. This collaboration should be achieved through the following steps:

    - Communication: Engage in open dialogue, discussing the specifics of the high-level task to make the goal more specific.

    - Task Decomposition: After understanding the task in its entirety, you guys need to decompose the high-level task into smaller, manageable sub-tasks. These sub-tasks should be small, specific, and executable with MineFlayer code, as you will be using MineFlayer to play MineCraft. The task decomposition will not be a one-time process but an iterative one. At regular intervals during playing the game, you'll be paused and should discuss with others again based on your progress. You'll propose new sub-tasks that respond to the current circumstances. So you don't need to plan far ahead, but make sure your proposed sub-tasks are small, simple and achievable, to ensure smooth progression. Each sub-task should contribute to the completion of the overall task, and each of you should take one subtask. That means, the number of sub-tasks should be 2. When necessary, the two sub-tasks can be identical for faster task accomplishment. You don't need to always agree with the decomposition proposed by other players. You can propose a more reasonable one when you find the decomposition not good. Be specific for the sub-tasks, for example, make sure to specify how many materials are needed.

    - Sub-task Dispatch: Post decomposition, the next step is to distribute the sub-tasks amongst yourselves. This will require further communication, where you consider each player's skills, resources, and availability. Ensure the dispatch facilitates smooth, ** parallel ** execution.

    - Integration and Finalization: In some tasks, you will need to integrate your individual efforts. For example, when crafting complicated stuff that require various materials, after collecting them, you need to consolidate all the materials with one of you. For these specific tasks, it is essential to discuss who should drop their items in inventory and who should collect them to reach the final goal. For other tasks that can be done completely parallal, this step can be ignored.

    # Task Description
    The high-level task: ${goal}

    # Relevant Recipes
    {{recipe}}

    # Reminder
    Remember, the key to achieving high efficiency as a group is maintaining a constant line of communication, cooperation, and coordination throughout the entire process. Now you should discuss with the other player. There will be 4 rounds for you guys to discuss the sub-tasks and the assignment at discussion phase. ** DO NOT imagine that you have achieved anything that is not mentioned in the chat history or have obtained anything that does not in your inventory. ** What will you, ${agent_name}, say now? Your response should only contain the words of ${agent_name}.

    # Chat History
    Below is the chat history among players:
    [Before Game Start. Discussion Phase.]
    ${chat_history}

    ${env_description}
    [${agent_name}]:
  # - Progress Monitoring and Sub-task Update: After you have made some progress, you can inform other players what you have achieved, and discuss whether there's a need for sub-task re-assignment or update based on the changing circumstances. Do not imagine that you have achieved something that is not mentioned in the chat history before game start.
  summarization_prompt: &sum_prompt |-
    Please review the following chat conversation and identify the specific latest sub-task or the next step that ${agent_name} needs to accomplish.

    # Chat Conversation
    ${chat_history}

    # Response Guidelines
    Your response should exclusively include the identified sub-task or the next step intended for ${agent_name}. Ensure that you are only extracting the sub-task or next step designated to ${agent_name}, excluding tasks assigned to other participants. Keep your response succinct and to the point.
    For instance, "Gather 3 wood for making pickaxes", "Kill 3 cows", "Drop 4 sticks", "Pickup 4 sticks dropped by xxx". Remember to add the quantifier and other important information discussed in the conversation.
...
```

Figure 16: AgentVerse Config

Decompose Prompt (Overcooked-AI)

SYSTEM

Your current mission is to leader all the players and execute a set of specified tasks within the Overcooked environment.
--- Background Information ---
Our system manages the task as a Directed Acyclic Graph (DAG).
In this turn, you need to decompose the tasks and arrange them in chronological order. Next turn we will analyse your result json to a graph.

A subtask-structure has the following json component:

```
{
    "id": int, id of the subtask start from 1,
    "action": string, name of the action,
    "required subtasks": list[int], if this subtask is directly prerequisite for other subtasks before it, list the subtask id here.
    "assigned_agent": str, name of agent. dispatch the subtask to the agent.
}
```

*** Important Notice ***
- The system do not allow agents communicate with each other, so you need to make sure the subtasks are independent.
- Sub-task Dispatch: Post decomposition, the next step is to distribute the sub-tasks amongst yourselves. This will require further communication, where you consider each player's skills, resources, and availability. Ensure the dispatch facilitates smooth, ** parallel ** execution.
- You can stop to generate the subtask-structure json if you think the task need the information from the environment, and you can not get the information from the environment now.
- The pot is considered full when there are three onions in it.
- There will be dispensers, pot, serving loc and counters in the kitchen.
  - Dispensers and counters are where agent could pick up objects. For example, agent can pick up onions from the onion dispenser.
- The soup will only start cooking when there are three onions in one pot.
- The team needs to put three onions in a pot to start cooking. Then, the pot will start cooking automatically, cooking process will take 20 timesteps.
- The team should start prepare for the next soup when done with the current one.
- agent cannot use move actions and don't use the location information in the observation.
- agent can only pickup and hold one thing at a time. To put the thing down agent is holding , agent should use place_obj_on_counter.
- Pay very close attention to player's states and kitchen states. For example, when agent is holding an onion, agent cannot pickup_onion. When agent has soup in hand, agent should deliver_soup instantly.
- agent can only pickup dish when a soup is ready.

USER

This is not the first time you are handling the task, so you should give part of decompose subtask-structure json feedback. Here is the query:
"""
the environment information around:
{env}

The high-level task:
{task}

Agent actions: (This is telling you what the agent can do in one step)
{agent_ability}
"""
Your response should exclusively include the identified sub-task or the next step intended for the agent to execute.
So, Each player (Player 0 & Player 1) should have one subtask, you should feedback {num} subtask-structure JSON.
Response should contain a list of subtask-structure JSON.

Figure 17: Decompose Prompt (Overcooked-AI)