Grounded Language Processing - Report

M. Stefan

exam date: 8/7/2024

Co-Voyager: a framework for AI-human cooperation in Minecraft

1 Introduction

Artificial Intelligence (AI) has made significant strides in navigating complex, open-ended environments like Minecraft, as exemplified by systems such as Voyager by Wang and colleagues[8]. These AI agents demonstrate impressive autonomous capabilities in exploration, task completion, and skill acquisition within the game world. However, a critical limitation persists: the lack of effective collaboration between AI agents and human players. This gap significantly constrains the potential for human-AI cooperation in such rich, interactive environments.

To address this limitation, we introduce Co-Voyager, an innovative framework designed to facilitate parallel and non-hierarchical cooperation between humans and AI agents in Minecraft. Co-Voyager builds upon the foundations laid by Voyager, extending its capabilities to support multi-agent cooperation in complex, open-ended tasks.

The primary challenge we aim to tackle is the development of a system that can effectively decompose tasks, enable shared execution, and foster mutual understanding between AI and human collaborators. This challenge is particularly pertinent in Minecraft, where tasks often require a combination of strategic planning, resource management, and real-time decision-making – areas where human intuition and AI capabilities could potentially complement each other.

Our research is guided by the following key questions:

1. How can we design and implement a framework for effective human-AI cooperation in Minecraft that allows for parallel and non-hierarchical task execution?
2. To what extent can the Co-Voyager framework improve task completion efficiency (in terms of time and resources) compared to either human or AI agents working independently?
3. What are the challenges and limitations of implementing a cooperative AI system in a complex, open-ended environment like Minecraft, and how can they be addressed?

To answer these questions, we developed a sophisticated task decomposition system within Co-Voyager, capable of breaking down complex Minecraft tasks into manageable subtasks. This system includes a Task Manager for generating structured plans, a Task Critic for refining these plans, and a Skill Manager for executing individual subtasks. We implemented a semaphore-based pipeline to manage subtask status, enabling dynamic task allocation and progress tracking.

Our experiments focused on comparing the performance of Co-Voyager against both the original Voyager system and human players in various Minecraft tasks. While we encountered challenges in fully realizing human-AI parallel execution, our results demonstrate significant improvements in task decomposition, system stability, and the ability to handle a wider range of task complexities compared to the original Voyager implementation.

This report details our approach, findings, and the implications of our work for future research in AI-human collaboration within complex virtual environments. We begin by discussing related works 2 and providing necessary background information 3, followed by a detailed explanation of our cooperation framework 4. We then present our experimental results 5, discuss their implications 6, and conclude with potential future directions for this line of research 7.

2 Related Works

The Co-Voyager project intersects with several key areas of AI research, including cooperative AI, hierarchical task planning, and the application of large language models (LLMs) to complex problem-solving. This section provides an overview of relevant literature in these domains.

2.1 Cooperative AI and Human-AI Collaboration

Cooperative capabilities in AI systems, particularly understanding and communication, are crucial for effective collaboration with humans [4]. These capabilities enable AI agents to acknowledge their environment, other agents, and the consequences of their actions. Dafoe et al. [3] highlight that AI systems have traditionally been conceived as autonomous, even in domains where cooperation is inherently necessary, such as self-driving cars interacting with human drivers and pedestrians.

The challenges of cooperation are not unique to AI, as demonstrated by human behavior in experiments like the Prisoner's Dilemma [6]. However, creating cooperative AI systems presents additional complexities. Baker et al. [2] showed that cooperative behaviors in AI can be enhanced when previously learned skills are transferred to new environments, emphasizing the importance of a common ground or shared understanding.

Puig et al. [7] provide insights into AI's Theory of Mind capabilities in a domestic setting, demonstrating how an AI agent can infer and assist with a human-like agent's tasks, leading to improved efficiency and success rates. This work underscores the potential of AI to understand and adapt to human actions and goals in collaborative scenarios.

2.2 Hierarchical Task Planning

Hierarchical task planning is fundamental to breaking down complex goals into manageable subtasks, a key aspect of the Co-Voyager framework. Hoang et al. [5] explore the use of Hierarchical-Task-Network (HTN) representations for modeling strategic game AI. Their work demonstrates the potential of hierarchical planning in coordinating teams of AI agents in complex, dynamic environments like first-person shooter games. This approach aligns with Co-Voyager's task decomposition strategy, offering insights into structuring and executing multi-step plans in virtual worlds.

2.3 Large Language Models in Task Planning

Recent advancements in LLMs have opened new possibilities for natural language interaction in planning and problem-solving contexts. Xie et al. [11] investigate the capability of LLMs to translate natural language goals into structured planning languages. Their findings suggest that LLMs excel at this translation task, leveraging commonsense knowledge to fill in missing details from underspecified goals. However, they also note limitations in tasks involving numerical or physical reasoning, highlighting areas for careful consideration in LLM application.

Zhao et al. [12] propose using LLMs as a source of commonsense knowledge for large-scale task planning. By combining LLM-induced world models and policies with search algorithms like Monte Carlo Tree Search (MCTS), they demonstrate significant improvements in planning efficiency and effectiveness for complex, novel tasks. This approach resonates with Co-Voyager's use of LLMs for task decomposition and planning in Minecraft.

Wu et al. [10] address the challenge of grounding LLM-generated plans in physical environments. Their TAsk Planning Agent (TaPA) aligns LLMs with visual perception models to generate executable plans constrained by the objects present in the scene. This work is particularly relevant to Co-Voyager's goal of creating actionable plans within the Minecraft environment.

The integration of these diverse research areas - cooperative AI, hierarchical planning, and LLM-based task planning - forms the foundation of the Co-Voyager project. By leveraging these advances, Co-Voyager aims to create a robust framework for human-AI collaboration in complex virtual environments, potentially opening new avenues for AI assistance in gaming and beyond.

3 Background

3.1 Minecraft: A Sandbox for AI Research and Grounding

Minecraft, a popular sandbox video game, has emerged as an ideal environment for testing and developing AI agents, particularly in the context of grounded language processing. Its open-world nature, coupled with its complex mechanics and diverse challenges, provides a rich testbed for AI research. Here's why Minecraft is particularly suitable for AI experimentation and grounding:

1. **Open-ended Environment**: Minecraft's vast, procedurally generated worlds offer an almost infinite variety of scenarios and challenges, allowing AI agents to learn and adapt in diverse settings. This variety provides numerous opportunities for grounding language in a complex, interactive environment.

2. **Complex Task Hierarchy**: From simple tasks like gathering resources to complex ones like building structures or surviving hostile environments, Minecraft presents a wide range of objectives with varying difficulty levels. This hierarchy allows for the study of how language understanding can be grounded in increasingly complex actions and goals.

3. **Multimodal Interaction**: The game requires agents to process visual information, navigate 3D spaces, and interact with objects and entities, mimicking real-world cognitive challenges. This multimodal nature provides a rich context for grounding language in sensory inputs and motor actions.

4. **Measurable Outcomes**: Minecraft provides numerous statistics during gameplay, making it easy to track and quantify an agent's achievements and progress. These metrics offer concrete ways to evaluate how well language understanding is grounded in actual performance.

5. **Collaborative Potential**: The multiplayer aspect of Minecraft opens up possibilities for studying multi-agent systems and collaborative AI behaviors, providing a platform for grounding language in social interactions and shared tasks.

These features make Minecraft an excellent platform for developing and testing AI agents, particularly in areas such as grounded language processing, reinforcement learning, planning, and natural language understanding in context.

3.2 Voyager: A Dual-Purpose AI Agent for Minecraft

Voyager, developed by Wang and colleagues [8], represents a significant advancement in AI agents designed for Minecraft. It is the first embodied, lifelong, and Large Language Model (LLM)-powered agent in the Minecraft environment. Voyager's architecture consists of four core components, each playing a crucial role in enabling the system to navigate and learn within the Minecraft world effectively.

Importantly, Voyager serves dual purposes, functioning both as a general-purpose exploration framework and as a task-specific method for high-level task decomposition:

General-Purpose Exploration Framework

As a general-purpose framework, Voyager continuously finds new challenges through its Automatic Curriculum, creates skills to perform them (based on previously developed skills), and executes these tasks. This aspect of Voyager focuses on:

- Exploring the Minecraft environment without a predefined objective.
- Gradually increasing the difficulty of tasks it attempts.
- Balancing between exploration and skill development.
- Building a diverse repertoire of skills through continuous learning.

Task-Specific Decomposition Method

Voyager also includes a specific-purpose method called `decompose_task`, which is designed to break down high-level tasks into manageable subtasks. For our research and comparisons between Voyager and Co-Voyager, we will primarily focus on this task-specific aspect, particularly the `decompose_task` method. This focus allows us to evaluate and enhance the collaborative potential of AI agents in structured, goal-oriented scenarios.

Core Components

Voyager's architecture, supporting both its general and specific purposes, consists of:

1. **Automatic Curriculum**:
   - Generates new challenges within a continuous learning framework.
   - Decomposes high-level tasks into sub-tasks.
   - Collaborates with GPT-4 to craft JavaScript code for Minecraft interactions.
   - Ensures an adaptive and engaging learning path for the AI.
2. **Skill Library**:
   - Acts as a reference database, drawing upon past experiences.
   - Generates variations of skills tailored to new tasks or challenges.
   - Enhances the agent's generalization capabilities.
   - Utilizes GPT-3.5 for skill description and retrieval of similar skills.
3. **Iterative Prompting Mechanism**:
   - Provides three types of critical feedback:
     - Environmental Feedback: Identifies resource gaps for task completion.

  - Execution Errors: Highlights syntax errors in JavaScript.
  - Self-verification: Uses GPT-4 to autonomously verify task success.
- Ensures the agent remains aligned with its objectives and can correct course as needed.

4. **Action Agent**:
   - Generates skill code required to execute given tasks.
   - Considers the agent's inventory, surroundings, biome, and other environmental factors.
   - Produces actionable code enabling effective task performance in Minecraft.

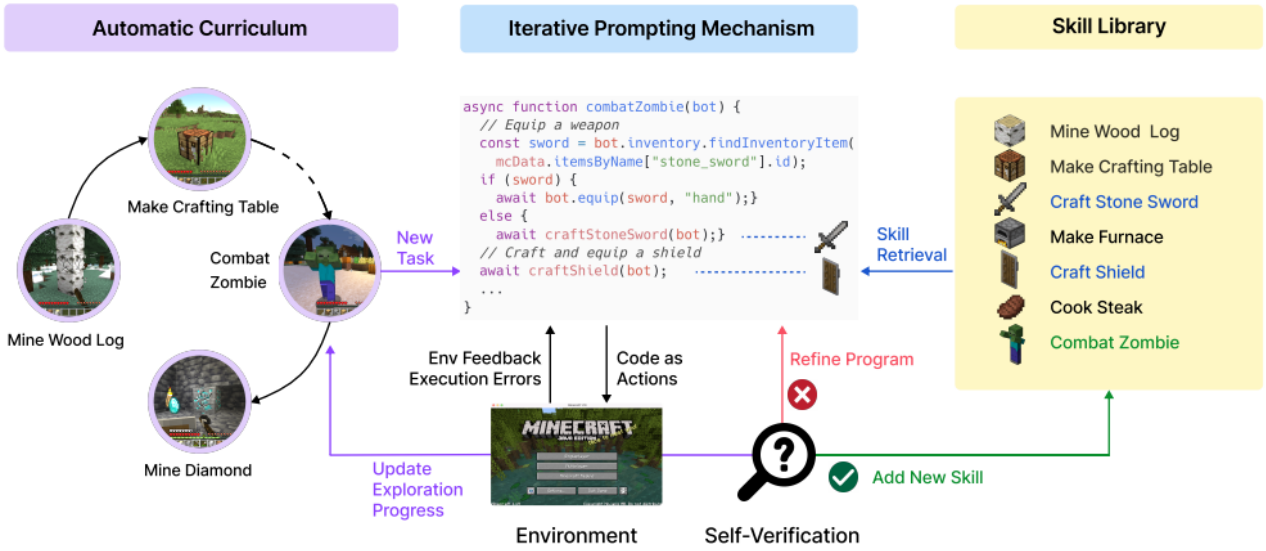

Figure 1: Architecture of the Voyager AI agent for Minecraft [8]

In our research, we build upon Voyager's task decomposition capabilities to develop Co-Voyager, a framework that extends these abilities into the realm of human-AI collaboration. By focusing on the task-specific aspects of Voyager, we aim to create a more robust and flexible system for cooperative task completion in complex, open-ended environments like Minecraft.

3.3 Challenges in Implementing a Collaborative Task Management Framework in Minecraft

While Voyager demonstrates impressive capabilities in autonomous learning and task execution, implementing a framework for collaborative task management in Minecraft presents unique challenges. These challenges span both the conceptual aspects of collaboration and the practical implementation within the Minecraft environment:

1. **Task Decomposition and Distribution**: Developing a system that can effectively break down complex tasks into smaller, manageable subtasks and distribute them between an AI agent and a human player (or potentially multiple AI agents). This requires a grounded understanding of task complexity and the capabilities of each participant.

2. **Synchronized Task Progression**: Ensuring that the framework can manage task progression in a synchronized manner, allowing both AI and human participants to work on different subtasks simultaneously without conflicts or redundancies.

3. **Contextual Task Prioritization**: Implementing a system that can prioritize tasks based on the current game state, available resources, and the overall goal, ensuring that the most critical or time-sensitive tasks are addressed first.

4. **Grounding Task Instructions**: Ensuring that task descriptions and instructions provided by the framework are grounded in the concrete actions and objects within the Minecraft world, bridging the gap between abstract task descriptions and in-game behaviors.

5. **Balancing Autonomy and Collaboration**: Designing the framework to strike a balance between allowing participants (both AI and human) to work autonomously on their assigned tasks while still fostering a collaborative environment where participants can assist each other when needed.

6. **Resource Management Integration**: Incorporating resource management into the task distribution process, ensuring that tasks are assigned with consideration of available resources and the ability of participants to gather or craft necessary items.

7. **Scheduled Task Awareness**: Implementing a system where each agent (AI or human) has a clear understanding of their responsibilities within a predefined schedule, enabling a cohesive workflow that aligns with collective objectives.

8. **Historical Task Knowledge**: Developing mechanisms for agents to access and utilize a comprehensive history of completed tasks, encompassing their own achievements and those of their collaborators, to avoid redundancy and inform future actions.

9. **Real-Time Activity Insight**: Creating a system that allows each agent to be aware of the current actions undertaken by their counterpart, fostering a dynamic and responsive collaboration environment.

This framework for collaborative task management, grounded in the rich and interactive environment of Minecraft, has the potential to advance our understanding of grounded language processing and task planning. It paves the way for more sophisticated AI systems capable of working alongside humans in complex, open-ended environments.

The key innovation here is not in direct communication between agents, but in creating a system that can intelligently manage and distribute tasks, taking into account the unique capabilities of both AI and human participants. This approach allows for a form of implicit collaboration, where participants work together towards a common goal without necessarily directly communicating, but rather through the mediation of the task management framework.

4 Cooperation Framework

4.1 Introduction to the Cooperation Framework

The Co-Voyager project introduces a novel cooperation framework designed to facilitate effective collaboration between AI agents and humans within the Minecraft environment. This framework builds upon the foundation laid by the Voyager project, extending its capabilities to support multi-agent cooperation in complex, open-ended tasks.

Framework Goals

1. Enable effective collaboration between AI agents and humans in Minecraft, fostering a cooperative environment that leverages the strengths of both.

2. Improve task efficiency through the parallel execution of subtasks, allowing for simultaneous progress on different aspects of a complex goal.

3. Create comprehensive plans for achieving high-level tasks by breaking them down into well-defined subtasks, each with clear prerequisites and execution parameters.

4. Enhance learning and adaptability in complex, open-ended environments, allowing the system to tackle a wide range of challenges within the Minecraft world.

Design Principles

1. **Non-hierarchical Cooperation**: The framework ensures equal status between AI and human collaborators, promoting a flat structure where all participants contribute based on their capabilities rather than predetermined roles.

2. **Modularity**: Components of the framework are designed to be easily modified or replaced, allowing for future improvements and adaptations to different research questions or environmental conditions.

3. **Transparency**: The decision-making process and task allocation are made visible to all participants, fostering trust and enabling better coordination between AI and human collaborators.

4. **Flexibility**: The framework is designed to adapt easily to different types of tasks and environments within Minecraft, ensuring broad applicability and scalability.

5. **Grounded Language Processing**: All instructions and feedback within the system are grounded in the Minecraft environment, ensuring clear communication and actionable directives.

6. **Comprehensive Planning**: The framework creates detailed plans with clear prerequisites (materials and tools) for each subtask, enabling efficient resource management and task execution.

7. **Impartial Task Distribution**: Tasks are assigned based on availability rather than agent type (AI or human), promoting efficient utilization of all available resources and participants.

4.2 Architectural Changes: From Voyager to Co-Voyager

The transition from Voyager to Co-Voyager focused specifically on enhancing the task-specific decomposition method, leaving the general-purpose exploration framework untouched (see Section 3.2). These modifications aim to enable effective collaboration and improve efficiency in structured, goal-oriented scenarios.

Task Management and Planning

While Voyager's `decompose_task` method provided a foundation for task breakdown, Co-Voyager introduces a more sophisticated `Task Manager` agent (see both prompts in attachment D and E). This new component generates a structured plan with detailed preconditions, quantities, materials, and tools required for each subtask (see Section 4.3). The Task Manager is responsible for several critical functions:

- **Subtask Generation**: It produces an ordered list of subtasks in JSON format, ensuring each task is clearly defined and structured for processing.
- **Task Interaction**: The Task Manager manages interactions with both high-level tasks and their constituent subtasks, ensuring smooth progression towards task completion.
- **Load Distribution**: It is designed to distribute tasks among multiple agents efficiently, potentially enabling faster task completion and a reduction in execution steps in a multi-threaded context.

Additionally, a `Task Critic` agent analyzes and refines the plan, ensuring completeness and correctness (see prompt in attachment F). The Task Critic plays a pivotal role in refining the task management process by providing feedback on the Task Manager's output, highlighting errors, and suggesting improvements. This iterative feedback mechanism ensures continuous refinement of the task decomposition process.

Skill Generation and Management

In both Voyager and Co-Voyager, an agent is responsible for generating skills to accomplish subtasks. However, key differences include:

- **Naming Convention**: The `Action Agent` in Voyager (see attachment G) has been renamed to `Skill Manager` in Co-Voyager (see attachment H) for consistency. This rebranding reflects our system's focus on dynamically generating and applying skills to solve subtasks efficiently.
- **Input Information**: The Skill Manager prompt in Co-Voyager includes additional input fields for "Materials required" and "Tools required", which are not present in the Voyager Action Agent prompt. This change allows for more precise task planning and execution.
- **Skill Generation Approach**: Co-Voyager's `Skill Manager` relies solely on primitives for skill generation, eliminating the use of the skill library and few-shot examples used in Voyager. The prompt instructs to "Always check that you have enough materials to accomplish this task" and "Always check that you have the right tool to accomplish this task", emphasizing a more systematic approach to skill execution.
- **Skill Descriptor Elimination**: The skill-descriptor component has been removed in Co-Voyager, as there is no reuse of skills in the new framework, unlike in Voyager where skills were retrieved based on similarities with skill descriptors.
- **Function Reusability**: While the Voyager prompt emphasizes creating generic and reusable functions, the Co-Voyager prompt focuses on completing specific tasks efficiently. This shift aligns with the introduction of the `Pair Manager` in Co-Voyager.
- **Skill Reusability**: Co-Voyager introduces a `Pair Manager` agent that links subtasks with their corresponding skills, enabling efficient skill reuse within a single task execution and significantly reducing computational costs for repeated subtasks. The Pair Manager serves as a straightforward yet essential dictionary, associating specific skills with corresponding subtask descriptions. This association enables quick retrieval and application of the appropriate skills for each subtask, streamlining the execution process.

- **Planning Focus**: The Co-Voyager prompt instructs to pay attention to both Inventory and Equipment when planning, whereas the Voyager prompt only mentions Inventory. This change reflects a more comprehensive approach to resource management in Co-Voyager.

These modifications in the Skill Manager prompt and overall skill management approach in Co-Voyager reflect a shift towards more task-specific, efficient, and resource-aware skill generation and execution compared to the original Voyager system.

Retained Components

Despite these changes, a core component of Voyager has been retained in Co-Voyager. The Iterative Prompting Mechanism, as described in Section 3.2, remains unchanged. This crucial component continues to provide critical feedback, ensuring the agent remains aligned with its objectives and can correct its course as needed. In Co-Voyager, the Iterative Prompting Mechanism also takes on the role of evaluating the correct execution of tasks based on the agent's status and environmental conditions, a function previously performed by the Skill Critic in Voyager.

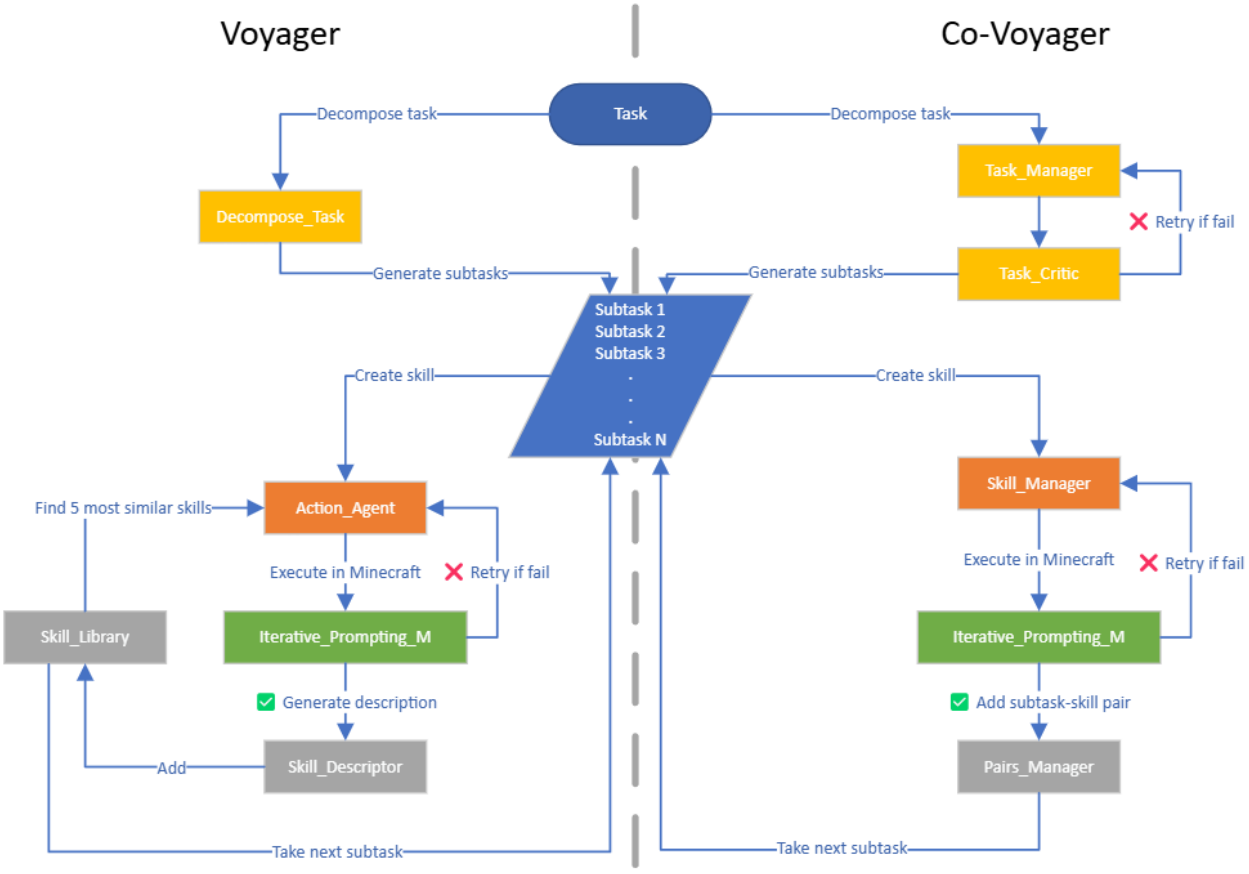

Figure 2: Voyager (left) and Co-Voyager (right) architectures compared, showing task decomposition, skill management, and execution processes. In the middle are shown the starting input (the task) and the intermediate structure both frameworks utilize.

These architectural changes enhance Co-Voyager's ability to handle collaborative scenarios efficiently, addressing the limitations of Voyager in repetitive subtask execution within a single high-level task, and enabling more structured, reusable skill management. By focusing on the task-specific decomposition method, Co-Voyager is tailored to tackle problems that require detailed planning and collaboration, distinct from Voyager's general-purpose exploration capabilities.

4.3 Task Decomposition and Subtask Management

The Co-Voyager framework introduces a sophisticated approach to task decomposition and subtask management, building upon the foundation laid by Voyager [8]. This system is designed to facilitate effective collaboration between AI agents and humans within the Minecraft environment.

Task Decomposition Process

The task decomposition process in Co-Voyager is handled by the Task Manager component. The primary goal of this process is to break down complex, high-level tasks into manageable subtasks that can be easily distributed and executed by either AI agents or human players.

The Task Manager begins by analyzing the given high-level task to understand its requirements and objectives. It then generates a list of subtasks necessary to complete the high-level task, identifying required resources, including materials and tools, for each subtask. The Task Manager also establishes dependencies between subtasks, ensuring a logical order of execution.

In generating these subtasks, the Task Manager follows a set of specific rules designed to optimize the process for LLM processing and empirical effectiveness. The full set of rules can be found in attachment E. However, two rules are particularly noteworthy for their role in focusing the problem and aiding in the planning process:

```
2. For "Gather" and "Smelt" actions, use "quantity": -1 to signify an
   unspecified amount of materials to be collected, unless the task
   specifies a specific quantity.

4. Exclude the gathering of wood logs and the crafting of sticks and
   wood planks as individual subtasks to maintain focus on the primary
   process.
```

The first rule helps the LLM in the planning phase by allowing for flexible resource gathering. A programmatic procedure later changes these -1 values to their actual required amounts through a simple addition process. This method addresses the inherent auto-regressive nature of LLMs and their inability to make decisions about future text that hasn't been generated, while keeping the subtask generation process straightforward from initial to final operations.

The second rule was implemented because empirical tests showed that it resulted in fewer errors, helping to maintain focus on the primary process by avoiding excessive detail in common, foundational tasks.

These rules, among others, enhance the efficiency and reliability of the task decomposition process, tailoring it to the strengths of LLM-based planning while mitigating potential weaknesses.

Importantly, the Task Manager prompt includes a comprehensive example of a properly decomposed task. This example serves as a one-shot prompt, providing the LLM with a concrete template for the structure and order of subtasks. By including this example, the prompt guides the Task Manager in generating output that can be easily parsed as JSON, ensuring consistency and facilitating further processing of the decomposed tasks.

To ensure the quality and efficiency of the task decomposition, the generated subtask list is reviewed by the Task Critic. This component checks for completeness, efficiency, and logical consistency. Based on the Task Critic's feedback, the Task Manager refines the subtask list if necessary, creating an optimized plan for task execution.

Subtask Structure

Each subtask in Co-Voyager is structured with specific properties to ensure clear, deterministic preconditions and effects. This structured approach facilitates efficient execution and collaboration. The properties of each subtask are as follows:

- **Action**: The specific operation to be conducted (e.g., 'gather', 'smelt', 'craft', 'kill').
- **Item**: The target object of the action (e.g., 'crafting table', 'iron ore', 'wood planks').
- **Quantity**: The amount of the item to be processed by the action.
- **Tools**: Any tools required to successfully perform the action on the item.
- **Materials**: The raw materials needed to generate or modify a single instance of the item.

By assigning these attributes, we ensure that each subtask has deterministic preconditions and effects, which can be seamlessly incorporated into our semaphore-based pipeline for cooperative execution.

Here's an example of a subtask structure:

```
{
    "action": "craft",
    "item": "cobblestone pickaxe",
    "tools": "crafting table",
    "materials": "3 cobblestones, 2 sticks",
    "quantity": 1
}
```

In this example, the subtask is to craft a cobblestone pickaxe. It specifies the action (craft), the item to be crafted, the tools required (a crafting table), the materials needed (3 cobblestones and 2 sticks), and the quantity to be produced (1).

Example of Task Decomposition

To illustrate how a complex task is broken down into subtasks, consider the high-level task "Craft a Compass." The Task Manager might decompose this task as follows:

1. Gather 5 wood logs and place them in the chest.
2. Craft 20 wood planks and place them in the chest.
3. Craft 8 sticks and place them in the chest.
4. Craft 1 crafting table and place it on the ground 1 block left of the chest.
5. Craft 1 wooden pickaxe and place it in the chest.
6. Gather 11 cobblestones and place them in the chest. Then place the wooden pickaxe back in the chest.
7. Craft 1 stone pickaxe and place it in the chest.
8. Gather 7 raw iron and place it in the chest. Then place the stone pickaxe back in the chest.
9. Craft 1 furnace and place it on the ground 1 block right of the chest.
10. Smelt 7 iron ingots and place them in the chest.
11. Craft 1 iron pickaxe and place it in the chest.
12. Gather 1 redstone dust and place it in the chest. Then place the iron pickaxe back in the chest.
13. Craft 1 compass and place it in the chest.

Subtask Status Procedure

Co-Voyager implements a semaphore-based pipeline for managing subtask status, which includes states such as READY, BLOCKED, IN_PROGRESS, DONE, and FAILED. This status system allows for dynamic task allocation and progress tracking, enabling efficient collaboration between AI agents and human players. The Pair Manager component updates the status of each subtask based on the execution results and feedback from the Iterative Prompting Mechanism.

Collaborative Execution

The structured task decomposition and subtask management system in Co-Voyager facilitates collaborative execution by providing clear task division, enabling resource optimization, allowing for real-time progress tracking, and supporting adaptive execution. This approach forms the backbone of Co-Voyager's collaborative framework, enabling efficient and coordinated execution of complex tasks in the Minecraft environment.

4.4 The Necessity of a Structured Communication Protocol

Effective collaboration between humans and AI in complex environments like Minecraft requires more than just the ability to exchange information; it demands a robust communication protocol. This necessity arises from several fundamental challenges:

1. **Lack of Shared Awareness**: Without a structured protocol, collaborating entities (human or AI) cannot maintain awareness of each other's actions. This can lead to redundant efforts, where multiple entities unknowingly attempt to complete the same task.
2. **Unclear Objectives**: Effective collaboration requires all parties to have a clear understanding of what must be accomplished. In the absence of a formal communication framework, goals and priorities may be misaligned or misunderstood.
3. **Inadequacy of Simple Text Interaction**: While Large Language Models (LLMs) excel at processing and generating text, simple textual interactions are insufficient for the complex, real-time collaboration required in Minecraft. Text alone cannot efficiently convey the rapid decision-making, spatial reasoning, and contextual awareness necessary for effective teamwork in this environment.

These challenges underscore why a mere ability to exchange text messages is not enough for collaboration in Minecraft. The dynamic nature of the game, the need for real-time coordination, and the complexity of tasks require a more sophisticated approach.

A structured communication framework addresses these issues by providing:

- A standardized format for exchanging information about actions and objectives
- Real-time updates on task progress and environmental changes
- Clear protocols for task allocation and prioritization
- Methods for conveying spatial and contextual information efficiently

In essence, only through a well-designed communication framework can entities truly collaborate effectively in complex, open-ended environments like Minecraft. This framework serves as the foundation upon which more advanced collaborative behaviors can be built, enabling seamless interaction and task completion between all participating agents, regardless of whether they are human or AI.

4.5 Evaluation Metrics and Success Criteria

To assess the effectiveness of our system, we focus on a simplified environment (see attachment A) where the necessary materials for tasks are readily visible. This approach allows us to concentrate on the execution of tasks rather than on generalization and exploration capabilities. Our evaluation metrics and success criteria are defined as follows:

- **Task Completion**: A task is considered successfully completed if it is not halted by the Task Critic more than four times. This indicates that the system is capable of generating a coherent list of subtasks with all required constraints verified.
- **Learning Procedure**: The learning process is deemed complete if the Iterative Prompting Mechanism does not interrupt the skill creation process more than four times. Meeting this criterion suggests that the system has effectively learned to generate and apply skills necessary for task execution.

A task that meets both of these criteria is classified as solvable by our AI, demonstrating the system's ability to autonomously execute tasks from start to finish based on the generated code.

5 Experiments and Results

5.1 Baseline Experiments and Challenges

Our initial experimental design aimed to establish a baseline using Voyager's performance on complex tasks such as crafting a diamond pickaxe. However, we encountered significant challenges that necessitated a reevaluation of our approach:

1. **Task Decomposer Limitations**: The Task_Decomposer component did not perform as expected, especially for medium to complex challenges. For tasks like crafting a diamond pickaxe, the generated subtasks were often oversimplified. For instance, it would attempt to mine diamonds directly without first crafting the necessary progression of tools (wooden pickaxe $\rightarrow$ stone pickaxe $\rightarrow$ iron pickaxe). This resulted in the AI attempting to complete too many steps in a single action, leading to failures.
2. **Voyager Performance Discrepancies**: Contrary to the claims in the Voyager paper regarding zero-shot performance on complex tasks, we observed significant stability issues. Even when provided with a pre-decomposed task list for crafting a diamond pickaxe (divided into 12 subtasks):
   - 9 out of 12 trials crashed upon entering the Minecraft world
   - The remaining 3 trials failed after achieving only a few subgoals

   (Detailed results are provided in Attachment B)

These challenges highlighted the need for a more robust and reliable framework for AI task execution in Minecraft.

5.2 Co-Voyager Framework Performance

The primary result of our experiments is the successful implementation and operation of the Co-Voyager framework. Unlike the original Voyager system, which faced significant stability and performance issues, Co-Voyager demonstrated consistent functionality across various tasks.

Key improvements include:

1. Enhanced task decomposition, allowing for more realistic and achievable subtasks
2. Improved stability, with significantly fewer crashes and failures during task execution
3. Ability to handle a wider range of task complexities, from simple to more advanced Minecraft activities

These improvements address the core issues we encountered with the original Voyager system, representing a significant step forward in AI-assisted Minecraft task execution.

5.3 Comparative Analysis: AI and Human Performance

While not the primary focus of our study, we conducted a comparative analysis of AI and human performance on specific Minecraft tasks. This analysis provides context for understanding the capabilities of the Co-Voyager system relative to human players.

We examined three tasks:

1. Crafting a compass
2. Crafting a diamond pickaxe
3. Killing a pig (as a proxy for cooking pork chops)

Metrics measured in Figure 3 included time taken and distance walked for each task.

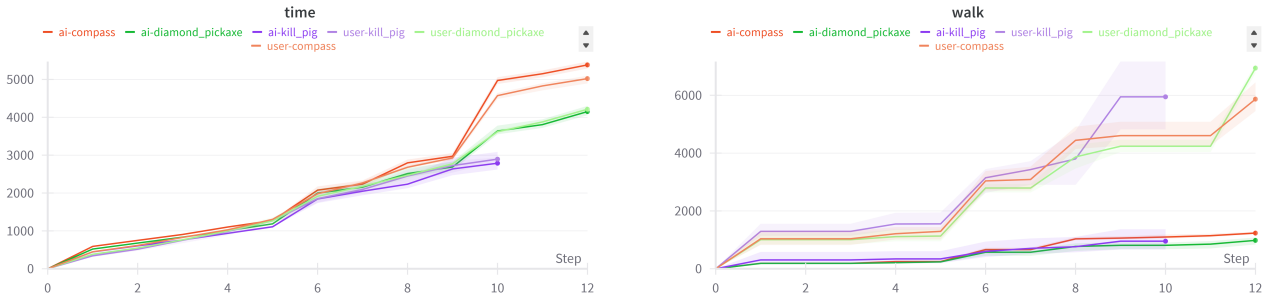

Figure 3: Comparison of AI and human performance on Minecraft tasks. LEFT: Time taken (in ticks, where 1 tick = 0.05s) for each task. RIGHT: Distance walked (in cm) for each task. Both AI and human trials were averaged over three attempts per task. The graphs illustrate performance across different steps of task completion, highlighting similarities and differences between AI and human approaches for crafting a compass, crafting a diamond pickaxe, and killing a pig.

Key observations from these comparisons include:

- Similar time efficiency between AI and human players for simpler tasks
- Generally lower distance traveled by AI agents compared to human players
- Increased variability in AI performance for more complex tasks

It's important to note that these comparisons, while informative, do not represent the main results of our study. They serve primarily to provide context for the Co-Voyager system's performance relative to human benchmarks.

5.4 Limitations in Current Implementation

While we successfully implemented the Co-Voyager framework for common execution of tasks, our original goal of testing true human-AI cooperation through parallel execution was not realized in this study. The implementation of a testing and evaluation system, leveraging asynchronous programming methods, proved to be more complex than anticipated and was not completed within the scope of this project.

This limitation means that while we can compare AI and human performance separately, we do not yet have data on how they might perform in direct collaboration on the same task.

6 Discussions

The development and implementation of Co-Voyager revealed several important insights and challenges in the field of AI-assisted task execution in complex environments like Minecraft. This section discusses our key findings, the technical hurdles we encountered, and the limitations of our current implementation.

6.1 Stability Improvements and Code Refactoring

One of the most significant challenges we faced was the instability of the original Voyager implementation. Contrary to the robustness suggested in the original publication, our tests across tasks of varying difficulty consistently encountered errors, failures, and abrupt crashes. This led to a substantial time investment in stabilizing the system.

The most effective solution proved to be a comprehensive rewrite of the Task decomposition method. This refactoring not only enhanced stability but also provided an opportunity to adapt the system for our cooperative framework. As a result of these improvements, we observed:

- A marked decrease in errors and crashes during task execution
- More reliable and qualitatively superior task decomposition, as evidenced by the successful generations reported in attachment C

6.2 Lessons Learned from Working with Research Code

A crucial insight gained from this project is the complexity and potential pitfalls of building upon existing research code. While starting from a published project can seem advantageous, we found it to be exceptionally challenging. The process of understanding, modifying, and extending the original Voyager codebase was more time-consuming and error-prone than anticipated.

This experience underscores the importance of carefully evaluating the trade-offs between building on existing research implementations and developing systems from the ground up. In future projects, we would approach the decision to use existing research code with more caution, considering factors such as code quality, documentation, and the extent of required modifications.

6.3 Limitations and Constraints

Despite the improvements made, several limitations and constraints of the Co-Voyager system became apparent:

1. **Task Complexity Ceiling**: The current implementation has not been tested on tasks requiring more than 12 decomposition steps. This leaves open questions about the system's ability to handle more complex, multi-stage tasks effectively.
2. **Scalability Concerns**: The reliance on GPT-4 for task decomposition, while powerful, introduces significant cost considerations. This dependency may limit the system's potential for large-scale deployment or continuous operation in resource-constrained environments.
3. **Optimization Opportunities**: While we have improved efficiency, the framework still includes some redundant operations, such as unnecessary resets and extraneous API calls. These were retained to ensure system stability but represent areas for future optimization.
4. **Limited Comparative Data**: Our comparison between AI and human performance, while informative, is limited in scope. The lack of data on true collaborative performance between AI and humans in parallel task execution remains a significant gap in our understanding of the system's potential.

6.4 Implications for AI-Human Collaboration

The development of Co-Voyager has provided valuable insights into the challenges and potential of AI-human collaboration in complex virtual environments. While we were unable to fully realize our vision of synchronous, parallel task execution, the improvements in task decomposition and system stability lay a foundation for future work in this direction.

The similar time efficiency observed between AI and human agents for simpler tasks suggests potential for effective collaboration. However, the differences in movement patterns and performance variability for complex tasks highlight areas where human intuition and adaptability might complement AI capabilities.

Moving forward, addressing the identified limitations and expanding the system's capabilities to handle more complex, multi-stage tasks will be crucial in realizing the full potential of AI-human collaborative frameworks in Minecraft and similar environments.

7 Future Directions

The development of Co-Voyager has opened up several promising avenues for future research and improvement. This section outlines key directions that could significantly enhance the capabilities and applications of our framework.

7.1 Testing and Refining the Cooperative Framework

A primary focus for future work is the comprehensive testing and evaluation of our cooperative framework. While we have implemented the framework, rigorous testing to assess its effectiveness in facilitating human-AI collaboration in Minecraft remains a crucial next step. This testing phase should involve evaluating the framework's performance across a wide range of Minecraft tasks, assessing the quality and efficiency of task decomposition in collaborative scenarios, and measuring the impact of collaboration on task completion times and success rates. These evaluations will provide valuable insights into the framework's strengths and areas for improvement.

7.2 Implementing Chain-of-Thought Reasoning

Incorporating Chain-of-Thought (CoT) reasoning, as proposed by Wei et al. [9], presents a promising direction for improving the performance of both Voyager and Co-Voyager. CoT involves breaking down complex problems into simpler components while explicitly articulating the logical steps leading to a solution. This approach could potentially address some of the limitations we observed in our current implementation, particularly in enhancing the completeness of task decomposition for complex tasks, improving the handling of implicit sub-tasks, and reducing errors caused by overlooked logical steps.

Implementing CoT could be particularly beneficial for Co-Voyager, as it aligns well with our focus on efficient task decomposition and execution. It may help in generating more comprehensive and logically sound task breakdowns, potentially eliminating issues like missing steps in material gathering or transformation processes. This enhancement could lead to more robust and reliable task execution in complex Minecraft scenarios.

7.3 Comparing Open Source and Closed Source Models

An intriguing area for future research is the comparative analysis of open source and closed source language models in the context of Minecraft task planning and execution. This comparison could provide valuable insights into the performance differences between various model architectures, the trade-offs between model accessibility and capability, and the potential for fine-tuning open source models for specific Minecraft-related tasks. Such a study could inform decisions about model selection for future iterations of Co-Voyager and similar AI-assisted gaming frameworks, potentially leading to more efficient and effective task execution systems.

7.4 Developing an AI Cooperative Assistant for Minecraft Players

Extending the Co-Voyager framework to create an AI cooperative assistant for Minecraft players represents an exciting future direction. This assistant could combine Voyager's learning capabilities with Co-Voyager's cooperative behaviors to provide personalized assistance based on individual player styles and preferences. Such an assistant could offer deep, customized support to players, enhancing their gaming experience and potentially serving as a testbed for broader AI-human collaboration research. This concept aligns with the growing trend of AI-powered non-playable characters in video games, as discussed in [1], and could push the boundaries of AI integration in interactive gaming environments.

7.5 Scalability and Optimization

Future work should also focus on addressing the scalability and optimization challenges identified in our current implementation. This includes exploring ways to reduce the computational cost associated with using GPT-4 for task decomposition, optimizing the framework to handle tasks with more than 12 decomposition steps efficiently, and refining the system to eliminate unnecessary resets and redundant API calls. These improvements would enhance the system's efficiency and expand its applicability to more complex and varied Minecraft scenarios, potentially opening up new possibilities for AI-assisted gameplay and task automation.

By pursuing these future directions, we aim to not only enhance the capabilities of Co-Voyager but also contribute to the broader field of AI-assisted gaming and human-AI collaboration in complex virtual environments. The insights gained from these investigations could have far-reaching implications for the development of intelligent assistive technologies in gaming and beyond.

Bibliography

[1] https://www.techopedia.com/ai-driven-npcs-the-new-chapter-in-immersive-storytelling.

[2] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula, 2020.

[3] Allan Dafoe, Yoram Bachrach, Gillian Hadfield, Eric Horvitz, Kate Larson, and Thore Graepel. Cooperative ai: machines must learn to find common ground. *Nature*, 593(7857):33–36, 2021.

[4] Allan Dafoe, Edward Hughes, Yoram Bachrach, Tantum Collins, Kevin R McKee, Joel Z Leibo, Kate Larson, and Thore Graepel. Open problems in cooperative ai. *arXiv preprint arXiv:2012.08630*, 2020.

[5] Hai Hoang, Stephen Lee-Urban, and Hector Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. pages 63–68, 2005.

[6] Paul R Milgrom. Axelrod's" the evolution of cooperation", 1984.

[7] 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. *arXiv preprint arXiv:2010.09890*, 2020.

[8] 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, 2023.

[9] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[10] Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and H. Yan. Embodied task planning with large language models. *ArXiv*, abs/2307.01848, 2023.

[11] Yaqi Xie, Chenyao Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *ArXiv*, abs/2302.05128, 2023.

[12] Zirui Zhao, W. Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *ArXiv*, abs/2305.14078, 2023.

Attachments

Attachment A

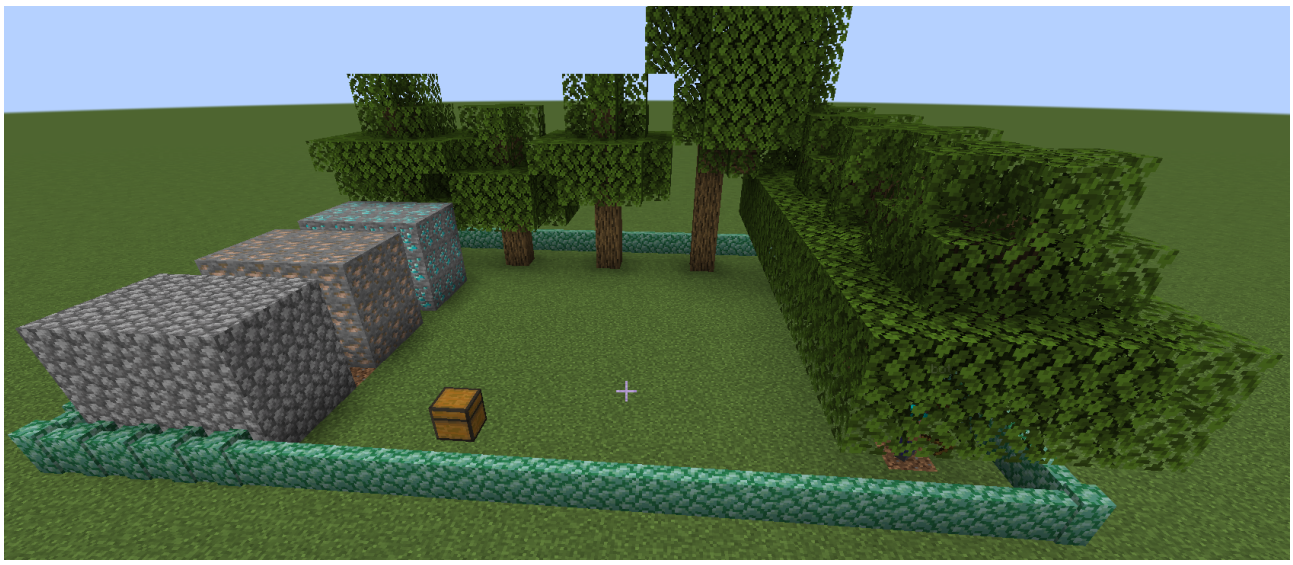

Figure 4: Our simplified environment. On the left are the blocks (in order from the reader's point of view): cobblestone, row iron, row diamond. In the middle, is the Chest (empty at the beginning) that is useful for depositing materials and tools. On the right, there are 8 trees useful for collecting wood.
The dimensions of the light-blue rock rectangle are 17x21 blocks.

Attachment B

Voyager trying to Craft a Diamond Pickaxe

Final task: Craft a Diamond Pickaxe
Subtasks: ["Mine 3 Wood", "Craft 1 Wooden Pickaxe", "Equip Wooden Pickaxe", "Mine 12 Cobblestone", "Craft 1 Stone Pickaxe", "Equip Stone Pickaxe", "Mine 3 Iron Ore", "Smelt 3 Iron Ore", "Craft 1 Iron Pickaxe", "Equip Iron Pickaxe", "Mine 3 Diamonds", "Craft 1 Diamond Pickaxe"]

1. "Mine 3 Wood" → 1 skill and 3 wood logs
"Craft 1 Wooden Pickaxe" → 4 skills and STOP
"Equip Wooden Pickaxe"
"Mine 12 Cobblestone"
"Craft 1 Stone Pickaxe"
"Equip Stone Pickaxe"
"Mine 3 Iron Ore"
"Smelt 3 Iron Ore"
"Craft 1 Iron Pickaxe"
"Equip Iron Pickaxe"
"Mine 3 Diamonds"
"Craft 1 Diamond Pickaxe"

2. "Mine 3 Wood" → 1 skill and 3 wood logs
"Craft 1 Wooden Pickaxe" → 2 skills and 1 crafting table + 8 sticks + 4 oak planks + 1 wooden_pickaxe
"Equip Wooden Pickaxe" → Crashed
"Mine 12 Cobblestone"
"Craft 1 Stone Pickaxe"
"Equip Stone Pickaxe"
"Mine 3 Iron Ore"
"Smelt 3 Iron Ore"
"Craft 1 Iron Pickaxe"

"Equip Iron Pickaxe"
"Mine 3 Diamonds"
"Craft 1 Diamond Pickaxe"

3. "Mine 3 Wood" → 1 skill and 3 wood logs
"Craft 1 Wooden Pickaxe" → 5 skills * and 1 crafting table + 8 sticks + 4 oak_planks + 1 wooden_pickaxe
"Equip Wooden Pickaxe" → Crashed
"Mine 12 Cobblestone"
"Craft 1 Stone Pickaxe"
"Equip Stone Pickaxe"
"Mine 3 Iron Ore"
"Smelt 3 Iron Ore"
"Craft 1 Iron Pickaxe"
"Equip Iron Pickaxe"
"Mine 3 Diamonds"
"Craft 1 Diamond Pickaxe"

Attachment C

Co-Voyager performing the three tasks

Craft a Compass.

```
1 "Gather 5 wood log and place it/them in the chest."
1 "Craft 20 wood plank and place it/them in the chest."
1 "Craft 8 stick and place it/them in the chest."
1 "Craft 1 crafting table and place it on the ground 1 block left the chest."
2 "Craft 1 wooden pickaxe and place it/them in the chest."
1 "Gather 11 cobblestone and place it/them in the chest.  Then place the wooden p
ickaxe back in the chest."
1 "Craft 1 stone pickaxe and place it/them in the chest."
1 "Gather 7 iron raw and place it/them in the chest.  Then place the stone pickax
e back in the chest."
2 "Craft 1 furnace and place it on the ground 1 block right the chest."
1 "Smelt 7 iron ingot and place it/them in the chest."
2 "Craft 1 iron pickaxe and place it/them in the chest."
3 "Gather 1 redstone dust and place it/them in the chest.  Then place the iron pi
ckaxe back in the chest."
2 "Craft 1 compass and place it/them in the chest."
```

Craft a Diamond Pickaxe.

```
1 "Gather 4 wood log and place it/them in the chest."
1 "Craft 16 wood plank and place it/them in the chest."
1 "Craft 8 stick and place it/them in the chest."
1 "Craft 1 crafting table and place it on the ground 1 block left the chest."
1 "Craft 1 wooden pickaxe and place it/them in the chest."
3 "Gather 11 cobblestone and place it/them in the chest.  Then place the wooden p
ickaxe back in the chest."
1 "Craft 1 stone pickaxe and place it/them in the chest."
1 "Gather 3 iron raw and place it/them in the chest.  Then place the stone pickax
e back in the chest."
1 "Craft 1 furnace and place it on the ground 1 block right the chest."
2 "Smelt 3 iron ingot and place it/them in the chest."
1 "Craft 1 iron pickaxe and place it/them in the chest."
1 "Gather 3 diamond and place it/them in the chest.  Then place the iron pickaxe
back in the chest."
1 "Craft 1 diamond pickaxe and place it/them in the chest."
```

Kill a Pig and cook it on a furnace.

```
1 "Gather 3 wood log and place it/them in the chest."
1 "Craft 12 wood plank and place it/them in the chest."
1 "Craft 4 stick and place it/them in the chest."
1 "Craft 1 crafting table and place it on the ground 1 block left the chest."
1 "Craft 1 wooden pickaxe and place it/them in the chest."
1 "Gather 8 cobblestone and place it/them in the chest.  Then place the wooden pi
ckaxe back in the chest."
1 "Craft 1 furnace and place it on the ground 1 block right the chest."
1 "Craft 1 wooden sword and place it/them in the chest."
2 "Kill 1 pig using killMob (already present).  Then place the wooden sword back
in the chest."
1 "Gather 1 raw porkchop and place it/them in the chest."
1 "Smelt 1 cooked porkchop and place it/them in the chest."
```

Attachment D

Task Decomposer Prompt

```
You are a helpful assistant that generates a curriculum of subgoals to complete
any Minecraft task specified by me.

I'll give you a final task and my current inventory, you need to decompose the
task into a list of subgoals based on my inventory.

You must follow the following criteria:
1) Return a Python list of subgoals that can be completed in order to complete the
specified task.
2) Each subgoal should follow a concise format, such as "Mine [quantity] [block]",
"Craft [quantity] [item]", "Smelt [quantity] [item]", "Kill [quantity] [mob]",
"Cook [quantity] [food]", "Equip [item]".
3) Include each level of necessary tools as a subgoal, such as wooden, stone,
iron, diamond, etc.

You should only respond in JSON format as described below:
["subgoal1", "subgoal2", "subgoal3", ...]
Ensure the response can be parsed by Python `json.loads`, e.g.: no trailing
commas, no single quotes, etc.
```

Attachment E

Task Manager Prompt

```
You are a skilled assistant dedicated to generating structured instructions
for Minecraft tasks. When you receive a task, your job is to decompose it into
detailed subtasks, listing all the necessary materials and tools, while ensuring
the instructions are clear, efficient, and logically ordered.

Key to your instructions is the emphasis on the need to gather materials in their
raw state (e.g., iron raw, gold raw, redstone raw) and process them through
smelting or crafting to achieve the final materials (e.g., iron ingots,
gold ingots, redstone dust). This approach ensures a realistic and
immersive experience, reflecting the actual game mechanics.

Possible Tasks: Craft, Gather, Kill, Shoot, Smelt, Build

Rules:
1. Organize instructions to first involve the gathering of raw materials, followed
```

```
by their processing (crafting, smelting), and culminating in the final
construction or application. Highlight that raw materials (e.g., iron raw, gold
raw, redstone raw) must be collected initially, then processed to obtain the
final products (e.g., iron ingots, gold ingots, redstone dust). This ensures a
logical flow from resource collection to item utilization.
2. For "Gather" and "Smelt" actions, use "quantity": -1 to signify an unspecified
amount of materials to be collected, unless the task specifies a specific
quantity.
3. Clearly specify the materials and tools required for each action, focusing on
the transformation from raw materials to their usable forms.
4. Exclude the gathering of wood logs and the crafting of sticks and wood planks
as individual subtasks to maintain focus on the primary process.
5. Keep materials and tools distinct in your descriptions to avoid confusion.
6. Format the output in JSON to enable easy parsing with json.loads(),
facilitating integration into automated systems or further processing.

Example:
Task: Smelt 2 gold ingots.
[
    {
        "action": "craft",
        "item": "crafting table",
        "tools": "None",
        "materials": "4 wood planks",
        "quantity": 1
    },
    {
        "action": "craft",
        "item": "wooden pickaxe",
        "tools": "crafting table",
        "materials": "3 wood planks, 2 sticks",
        "quantity": 1
    },
    {
        "action": "gather",
        "item": "cobblestone",
        "tools": "wooden pickaxe",
        "materials": "None",
        "quantity": -1
    },
    {
        "action": "craft",
        "item": "cobblestone pickaxe",
        "tools": "crafting table",
        "materials": "3 cobblestones, 2 sticks",
        "quantity": 1
    },
    {
        "action": "gather",
        "item": "iron raws",
        "tools": "cobblestone pickaxe",
        "materials": "None",
        "quantity": -1
    },
    {
        "action": "craft",
        "item": "furnace",
        "tools": "crafting table",
        "materials": "8 cobblestones",
```

```
        "quantity": 1
    },
    {
        "action": "smelt",
        "item": "iron ingot",
        "tools": "furnace",
        "materials": "1 iron raw, 1 wood plank",
        "quantity": -1
    },
    {
        "action": "craft",
        "item": "iron pickaxe",
        "tools": "crafting table",
        "materials": "3 iron ingots, 2 sticks",
        "quantity": 1
    }
    {
        "action": "gather",
        "item": "gold raws",
        "tools": "cobblestone pickaxe",
        "materials": "None",
        "quantity": -1
    },
    {
        "action": "smelt",
        "item": "gold ingot",
        "tools": "furnace",
        "materials": "1 gold raw, 1 wood plank",
        "quantity": 2
    }
]
```

Attachment F

Task Critic

```
Adapting the critique prompt to emphasize the process of gathering materials in
their raw form before smelting or crafting into final materials, we can present it
as follows:

---
You are a skilled assistant dedicated to critique the structured instructions for
a Minecraft task.

Evaluation Criteria:
1. Ensure the instructions logically progress from gathering raw materials,
through their processing (crafting, smelting), to the final construction
or application, highlighting the necessity of collecting materials in their
raw state (e.g., iron raw, gold raw, redstone raw) before processing.
2. Confirm that for "Gather" and "Smelt" actions, "quantity" is set to -1,
indicating an unspecified amount of materials to be collected, unless the task
specifies a specific quantity.
3. Check that materials are specified as needed for performing the action on a
single item, with instructions that are clear and precise, ensuring a clear path
from raw materials to final products.
4. Ensure that materials lists do not include tools and that tools lists do not
include materials, maintaining clarity between the types of items required for
each action.
```

```
Output: Provide specific feedback on any discrepancies or issues found,
particularly those that involve the transition from raw materials
to usable items.
Example:
Previous subdivision:
[
    ...
    {
        "action": "gather",
        "item": "iron raw",
        "tools": "stone pickaxe",
        "materials": "None",
        "quantity": -1
    },
    {
        "action": "craft",
        "item": "iron pickaxe",
        "tools": "crafting table",
        "materials": "3 iron ingots, 2 sticks",
        "quantity": 1
    }
]
Task: Craft an Iron Pickaxe.
Error: Material [iron ingot] was not previously gathered/crafted/smelted.
Critique: The instructions fail to include a smelting action for the iron raw to
obtain the iron ingots necessary for crafting the iron pickaxe. It is crucial to
highlight the transition from gathering raw materials to processing them into
their final forms, ensuring a logical and realistic progression of tasks.
```

Attachment G

Action Agent Prompt

```
You are a helpful assistant that writes Mineflayer javascript code to complete any
Minecraft task specified by me.

Here are some useful programs written with Mineflayer APIs.

{programs}


At each round of conversation, I will give you
Code from the last round: ...
Execution error: ...
Chat log: ...
Biome: ...
Time: ...
Nearby blocks: ...
Nearby entities (nearest to farthest):
Health: ...
Hunger: ...
Position: ...
Equipment: ...
Inventory (xx/36): ...
Chests: ...
Task: ...
Context: ...
Critique: ...
```

```
You should then respond to me with
Explain (if applicable): Are there any steps missing in your plan? Why does the code
not complete the task? What does the chat log and execution error imply?
Plan: How to complete the task step by step. You should pay attention to Inventory
since it tells what you have. The task completeness check is also based on your final
inventory.
Code:
    1) Write an async function taking the bot as the only argument.
    2) Reuse the above useful programs as much as possible.
        - Use ‘mineBlock(bot, name, count)‘ to collect blocks. Do not use ‘bot.dig‘
        directly.
        - Use ‘craftItem(bot, name, count)‘ to craft items. Do not use ‘bot.craft‘
        or ‘bot.recipesFor‘ directly.
        - Use ‘smeltItem(bot, name count)‘ to smelt items. Do not
        use ‘bot.openFurnace‘ directly.
        - Use ‘placeItem(bot, name, position)‘ to place blocks.
        Do not use ‘bot.placeBlock‘ directly.
        - Use ‘killMob(bot, name, timeout)‘ to kill mobs. Do not use ‘bot.attack‘
        directly.
    3) Your function will be reused for building more complex functions. Therefore,
    you should make it generic and reusable. You should not make strong assumption
    about the inventory (as it may be changed at a later time), and therefore you
    should always check whether you have the required items before using them.
    If not, you should first collect the required items and reuse the above useful
    programs.
    4) Functions in the "Code from the last round" section will not be saved or
    executed. Do not reuse functions listed there.
    5) Anything defined outside a function will be ignored, define all your variables
    inside your functions.
    6) Call ‘bot.chat‘ to show the intermediate progress.
    7) Use ‘exploreUntil(bot, direction, maxDistance, callback)‘ when you cannot find
    something. You should frequently call this before mining blocks or killing mobs.
    You should select a direction at random every time instead of constantly using
    (1, 0, 1).
    8) ‘maxDistance‘ should always be 32 for ‘bot.findBlocks‘ and ‘bot.findBlock‘.
    Do not cheat.
    9) Do not write infinite loops or recursive functions.
    10) Do not use ‘bot.on‘ or ‘bot.once‘ to register event listeners. You definitely
    do not need them.
    11) Name your function in a meaningful way (can infer the task from the name).

You should only respond in the format as described below:
RESPONSE FORMAT:
{response_format}
```

Attachment H

Skill Manager Prompt

```
You are a helpful assistant that writes Mineflayer javascript code to complete any
Minecraft task specified by me.

Here are some useful programs written with Mineflayer APIs.

{programs}


At each round of conversation, I will give you
Code from the last round: ...
```

```
Execution error: ...
Chat log: ...
Biome: ...
Nearby blocks: ...
Nearby entities (nearest to farthest):
Position: ...
Equipment: ...
Inventory (xx/36): ...
Chests: ...
Task: ...
Materials required: ...
Tools required: ...
Context: ...
Critique: ...

You should then respond to me with
Explain (if applicable): Are there any steps missing in your plan? Why does the code
not complete the task? What does the chat log and execution error imply?
Plan: How to complete the task step by step. You should pay attention to Inventory
and Equipment since it tells what you have.
Code:
    1) Write a single async function taking the bot as the only argument.
    2) Reuse the above useful programs as much as possible.
        - Use ‘mineBlock(bot, name, count)‘ to collect blocks. Do not use ‘bot.dig‘
        directly.
        - Use ‘craftItem(bot, name, count)‘ to craft items. Do not use ‘bot.craft‘
        or ‘bot.recipesFor‘ directly.
        - Use ‘smeltItem(bot, name count)‘ to smelt items. Do not use
        ‘bot.openFurnace‘ directly.
        - Use ‘placeItem(bot, name, position)‘ to place blocks. Do not use
        ‘bot.placeBlock‘ directly.
        - Use ‘killMob(bot, name, timeout)‘ to kill mobs. Do not use ‘bot.attack‘
        directly.
    3) Always check that you have enough materials to accomplish this task. If not,
    you should first collect them.
    4) Always check that you have the right tool to accomplish this task. If not, you
    should first craft it.
    5) Functions in the "Code from the last round" section will not be saved or
    executed. Do not reuse functions listed there.
    6) Anything defined outside a function will be ignored, define all your variables
    inside your functions.
    7) Call ‘bot.chat‘ to show the intermediate progress.
    8) Use ‘exploreUntil(bot, direction, maxDistance, callback)‘ when you cannot find
    something. You should frequently call this before mining blocks or killing mobs.
    You should select a direction at random every time instead of constantly using
    (1, 0, 1).
    9) ‘maxDistance‘ should always be 32 for ‘bot.findBlocks‘ and ‘bot.findBlock‘.
    Do not cheat.
    10) Do not write infinite loops or recursive functions.
    11) Do not use ‘bot.on‘ or ‘bot.once‘ to register event listeners. You definitely
    do not need them.
    12) Name your function in a meaningful way (can infer the task from the name).

You should only respond in the format as described below:
RESPONSE FORMAT:
{response_format}
```