Planning Before Acting: Dependency-Aware LLM Agents in Minecraft

Thomas Fenno, Matthew Mitchell, Daniel S. Brown

Keywords: embodied agents, large language models, long-horizon planning, structured prompting

Summary

This paper studies long-horizon dependency management for LLM-powered agents in embodied environments. Existing LLM agents can often interpret natural-language objectives and act through tool interfaces, but they struggle when success requires satisfying ordered prerequisites across many steps. We are motivated by dependency errors, where an agent attempts a subtask before acquiring the resources, tools, or intermediate items needed to complete it.

We introduce the Structured Prompting Loop (SPL), which decomposes an objective into a Primary-Task Directed Acyclic Graph (DAG) of inventory-item dependencies, conditions that graph on the agent's current state, selects feasible subtasks, and invokes constrained failure replanning when execution fails. Across a suite of tasks within the game of Minecraft, SPL achieves higher average success, lower timeout-normalized episode duration, and fewer environment commands than the baseline agent. These preliminary results suggest that explicit dependency modeling can improve the reliability of LLM agents in open-ended embodied environments.

Contribution(s)

1. We introduce the Structured Prompting Loop (SPL), a dependency-aware planning framework for LLM agents that decomposes objectives into Primary-Task DAGs, conditions those graphs on the current environment state, and selects feasible subtasks during execution.
   Context: SPL is built as an extension of the Mindcraft/Mineflayer agent framework that enables LLM-powered agents in Minecraft (Mindcraft Bots, 2025), and models task structure primarily through inventory-item dependencies rather than arbitrary gameplay actions, locations, or social interactions.
2. We evaluate SPL against the baseline Mindcraft agent on a suite of Minecraft tasks inspired by in game advancements (Minecraft Wiki, 2026) across fifteen random world seeds, measuring task success, episode duration, and environment commands.
   Context: This is a focused preliminary evaluation rather than a large-scale benchmark. We ran each agent on 8 different tasks across 15 seeds for a total of 120 pairwise comparisons.
3. We provide empirical evidence that explicit dependency modeling can improve long-horizon embodied task execution, with SPL achieving higher average success, lower timeout-normalized episode duration, fewer environment commands, and lower aggregate LLM usage/API cost.
   Context: The results support the usefulness of dependency-aware planning on the evaluated task suite. We use total environment commands as an observable efficiency proxy for wasted execution effort, not as a direct measurement of dependency errors.

Planning Before Acting: Dependency-Aware LLM Agents in Minecraft

Thomas Fenno[1†], **Matthew Mitchell**[1,†], **Daniel S. Brown**[1]
{u1466976, u1398832, daniel.s.brown}@utah.edu

[1]**Kahlert School of Computing, University of Utah, Salt Lake City, USA**

[†] equal contribution

Abstract

Embodied Large Language Model (LLM) agents are increasingly used for high-level planning in open-ended environments such as Minecraft, but they often fail on long-horizon tasks because they attempt subtasks before satisfying required dependencies. This problem is important because dependency errors lead to failed actions, redundant work, longer runtimes, and lower task success, limiting the reliability of LLM-based agents in embodied settings. We propose a Structured Prompting Loop (SPL) that decomposes each objective into a Primary-Task directed acyclic graph of dependencies, repeatedly conditions that graph on the agent's current state, selects feasible subtasks, and invokes constrained failure-replanning when execution fails. We evaluate the SPL agent against the baseline Mindcraft agent on a suite of Minecraft tasks inspired by in-game advancements with varying dependency lengths. Across the evaluated tasks, the SPL agent achieves higher average success, lower timeout-normalized episode duration, and fewer environment commands than the baseline, with the clearest gains appearing on longer-horizon objectives that require many prerequisite resources, tools, and crafting steps. These results provide preliminary evidence that explicit dependency modeling improves long-horizon embodied task execution by helping agents choose feasible subtasks and avoid inefficient or invalid action sequences.

1 Introduction

Large language models (LLMs) have generated substantial interest as high-level planning modules for embodied agents. Because these models encode broad world knowledge and can produce structured reasoning traces, they appear well suited for tasks that require an agent to translate natural language objectives into sequences of actions. However, long-horizon interactive environments introduce a persistent challenge: agents must reason not only about what action to take next, but also about which resources, tools, intermediate states, or environmental conditions must be satisfied before later actions become feasible.

We study this problem in Minecraft because it provides a controlled but open-ended environment for evaluating long-horizon embodied task execution. Minecraft combines resource gathering, crafting, navigation, tool use, and dynamic environmental conditions, making it a useful testbed for studying whether LLM agents can manage ordered dependencies during execution. This work builds on Mindcraft (White et al., 2025), which provides an interface for creating LLM-powered agents that play Minecraft with access to low-level action functions. The base Mindcraft agents demonstrate an impressive ability to interpret natural language instructions and complete simple objectives. However, in our preliminary experiments, we observed that these agents often struggle on long-horizon tasks that require multiple ordered prerequisites. For example, an agent may attempt to mine stone

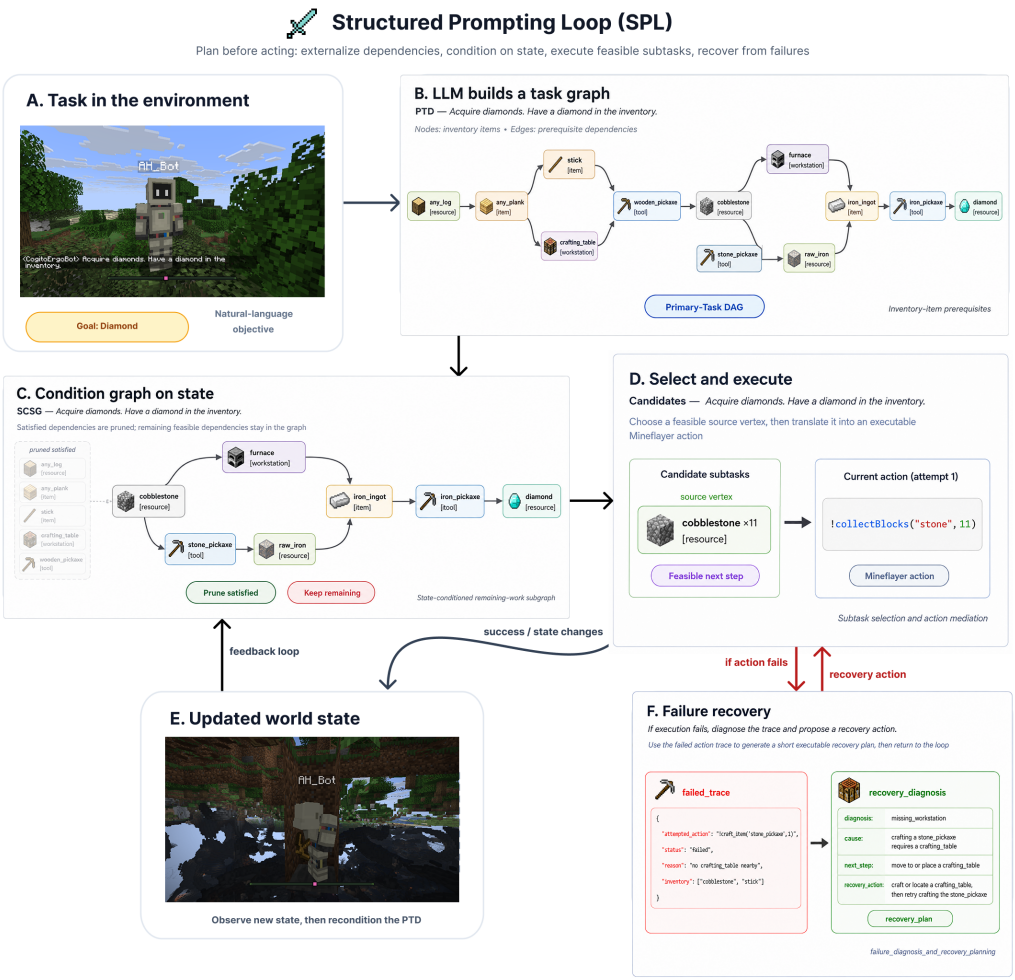


Figure 1: Overview of the Structured Prompting Loop (SPL). The agent converts a natural-language Minecraft objective into a Primary-Task DAG (PTD), conditions the graph on the current world state, selects a feasible subtask, and executes the corresponding Mineflayer action. After each action, the updated state is used to recondition the PTD and continue the loop. When execution fails, a constrained failure-replanning module diagnoses the failed trace and proposes recovery actions before returning to the main loop.

before obtaining a pickaxe, smelt iron before collecting fuel, or collect lava before crafting a bucket. We refer to these cases as *dependency errors*: attempts to complete a subtask before satisfying the items, tools, or resources required for that subtask.

These failures highlight a broader tension in embodied LLM planning. One common approach is to ask an LLM to generate a complete zero-shot plan before execution, which can help the agent reason over long-horizon dependencies but may become brittle when the environment differs from the assumptions in the initial plan (Huang et al., 2022). Another approach, exemplified by ReAct-style observe–reason–act loops (Yao et al., 2023), allows the agent to repeatedly condition its decisions on environmental feedback, but can make planning myopic when the agent selects the next action without an explicit representation of later dependencies. Open-ended agents such as Voyager (Wang et al., 2023) demonstrate the power of closed-loop prompting and self-verification, but their design is primarily aimed at continual exploration rather than reliable completion of a fixed dependency-heavy objective.

We propose a Structured Prompting Loop (SPL) that combines the strengths of these approaches. At the beginning of each task, the agent uses the LLM to construct a directed acyclic graph (DAG) representing the chain of dependencies required to complete the objective. During execution, this graph is conditioned on the current state to determine which dependencies have already been satisfied and which source vertices are feasible to attempt next. The agent then selects a feasible subtask, translates it into an executable action, observes the resulting state, and recomputes the remaining state-conditioned graph. If execution fails, a constrained LLM-based failure-replanning module uses the failed trace to generate a short recovery plan before returning control to the structured loop.

Conceptually, this framework uses an initial long-horizon dependency model to shorten the effective horizon of each execution decision. Rather than asking the LLM to choose a single next action with limited long-term structure, the agent first constructs an explicit representation of the task's prerequisite chain and then acts with respect to the currently feasible portion of that plan. This idea is related in spirit to action chunking approaches (Zhao et al., 2023), which reduce effective task horizon by predicting temporally extended action sequences. Our implementation does not directly learn or execute low-level action chunks; instead, it applies the same motivation at a higher level of abstraction by using dependency structure to convert a long-horizon objective into shorter feasible subtasks.

Our central hypothesis is that explicit dependency modeling improves the reliability of embodied LLM agents on long-horizon goals. We evaluate the SPL agent against the baseline Mindcraft agent, which follows a closed-loop observe–reason–act pattern, within Minecraft on a suite of tasks inspired by in-game Minecraft advancements (Minecraft Wiki, 2026). The results provide preliminary evidence that frontloading dependency reasoning into a structured plan, while retaining closed-loop state conditioning and failure recovery, can improve task success, reduce wasted actions, and shorten episode duration.

2 Related Work

LLM-Based Planning and Reasoning. LLMs have been studied as high-level planners for sequential decision-making tasks because they can use pretrained world knowledge to translate natural language goals into action sequences. In zero-shot planning approaches, the model receives a goal and produces a plan without task-specific training (Huang et al., 2022). More broadly, prompting methods such as chain-of-thought reasoning show that LLMs can benefit from producing intermediate reasoning steps before giving an answer (Wei et al., 2023), while planning-oriented systems use language models to generate or select action sequences for embodied agents (Ahn et al., 2022; Singh et al., 2022). These approaches show that LLMs can expose useful long-horizon structure, but they also highlight a persistent grounding problem: generated plans may be brittle when the environment changes, when intermediate actions fail, or when the plan does not map cleanly onto executable actions. In long-horizon embodied settings, this brittleness is especially problematic because small errors can compound across many dependent subtasks.

Closed-Loop Embodied Agents in Minecraft. A complementary line of work improves LLM agents by interleaving reasoning, acting, and feedback. ReAct-style agents repeatedly observe the environment, reason over the current context, and select the next action (Yao et al., 2023), while reflection-based methods use feedback to refine future outputs or decisions (Shinn et al., 2023; Madaan et al., 2023). These approaches make agents more responsive than fixed open-loop plans, but they can still encourage short-horizon decisions when the agent lacks an explicit representation of the broader dependency structure.

Minecraft has become a common environment for studying this problem because it requires navigation, resource gathering, crafting, tool use, and adaptation in an open-ended world. MineDojo provides a large-scale platform for studying generalist Minecraft agents (Fan et al., 2022), and Voyager demonstrates that LLM agents can use closed-loop prompting, self-verification, and skill discovery to explore and acquire new capabilities over extended interaction horizons (Wang et al., 2023).

We build on Mindcraft, which provides an interface for deploying LLM-powered Minecraft agents through Mineflayer action functions (White et al., 2025). Our work is motivated by the same need to connect reasoning and action, but focuses on fixed objectives where success depends on satisfying ordered task dependencies.

Memory and Long-Horizon Agent Architectures. Prior work has improved LLM agents by adding memory, retrieval, and other structured components around the base model. Generative Agents store experience traces, retrieve relevant memories, and synthesize higher-level reflections to support coherent behavior over time (Park et al., 2023), while MemoryBank adds a long-term memory mechanism that allows an LLM system to store, retrieve, and update information across extended interactions (Zhong et al., 2024). In embodied settings, Chari et al. (2025) augment the planning LLM with an experience database that supports context-aware planning, while Lică et al. (2024) organize agent memory into episodic, semantic, and procedural components. These systems suggest that LLM agents benefit from external structure that persists across decisions. Our approach follows this general motivation, but uses an explicit task-dependency graph rather than a memory database: the agent represents what must be satisfied to complete the objective, then conditions that representation on the current state during execution.

3 Problem Statement

This work studies long-horizon dependency management for embodied LLM agents. Open-loop zero-shot plans can expose useful prerequisite structure, but may become brittle when assumptions about the environment fail. Closed-loop observe–reason–act agents can adapt to feedback, but may make myopic decisions when they lack an explicit representation of the dependencies required to complete the objective. We model a long-horizon rollout as a sequence of subtasks $\tau = (\tau_1, \tau_2, \ldots, \tau_T)$, where completing the objective requires executing subtasks in an order consistent with their dependencies. A dependency error occurs when the agent attempts a subtask τ_i before satisfying one or more prerequisites required for its successful execution. As the rollout length T increases, these errors can compound, causing failed actions, redundant work, and longer episode runtimes.

The goal of this work is to study whether explicit dependency modeling can improve long-horizon task execution by helping an embodied LLM agent identify which subtasks are currently feasible, avoid invalid action sequences, and recover when execution diverges from the initial plan. Because reliable automatic annotation of dependency errors is difficult in open-ended rollouts, our evaluation uses task success, capped episode duration, and total environment commands as observable outcomes that reflect whether dependency-aware planning improves reliability and execution efficiency.

4 Method

SPL is a planning wrapper added to the baseline Mindcraft agent. SPL combines the long-horizon structure of zero-shot planning with the adaptivity of closed-loop execution. The agent first prompts the LLM to decompose the primary objective into an explicit dependency graph. During execution, this graph is repeatedly conditioned on the current state to identify feasible subtasks, which are then executed through the Mineflayer action interface. If execution fails, a constrained LLM-based recovery module uses the failed trace to generate a short recovery plan before returning control to the structured loop.

Figure 1 provides an illustration of this loop for the task objective *acquire a diamond*.

4.1 Design Principles

SPL is based on two design principles. First, long-horizon objectives can often be decomposed into shorter subtasks by identifying the prerequisites required to complete the primary objective, such as resources, tools, intermediate states, locations, or environmental preconditions. Second, this dependency model must remain connected to the agent's current state: instead of following a fixed open-loop plan, SPL repeatedly updates the remaining dependency structure so that the agent selects only from subtasks that are currently feasible.

4.2 Primary-Task DAG and State Conditioning

In our Minecraft implementation, the dependency model is instantiated as a directed acyclic graph (DAG) of inventory-item dependencies. This graph is not supplied by the benchmark or written manually for each task. Instead, SPL receives the same high-level objective string used to initialize the episode and prompts the LLM to infer the inventory items required to achieve that objective. The LLM recursively expands those requirements into prerequisite items, tools, resources, and workstations. We call the resulting graph the Primary-Task DAG (PTD), where each vertex represents an inventory item and each edge $A \rightarrow B$ indicates that item A is required to obtain item B.

During execution, SPL conditions the PTD on the current Minecraft state s to obtain a state-conditioned subgraph (SCSG). Vertices are removed when the agent's inventory satisfies the required quantity, partially satisfied vertices have their remaining requirements updated, and disconnected intermediate sinks are pruned. If the SCSG is empty, the primary task is complete. Otherwise, source vertices in the SCSG define the feasible subtasks from which the system selects the next task. The PTD construction and state-conditioning procedures are summarized here Algorithms 2, 3.

4.3 Structured Prompt Interfaces

SPL uses four structured prompt interfaces. The PTD generation prompt maps the primary objective to a JSON dependency graph. The PTD feedback prompt validates candidate graphs for structural, semantic, and quantity errors, and the refinement prompt repairs issues while preserving valid graph components. Finally, the failure-replanning prompt receives a failed execution trace and returns a bounded sequence of executable recovery actions. Each prompt uses a constrained input/output schema so that downstream deterministic components can consume the LLM output directly. Prompt interface specifications are provided in Appendix A; full prompt text and evaluation code are included in the supplementary material.

4.4 Structured Prompting Loop

The SPL begins by constructing and refining a PTD using a self-refinement loop inspired by Self-Refine (Madaan et al., 2023). During execution, the PTD is repeatedly conditioned on the current state, source vertices are converted into candidate subtasks, and one subtask is selected and mediated into an executable Minecraft command. After each action, the system recomputes the SCSG. If execution fails, the failure-replanning prompt proposes recovery actions using the available environment actions. The full SPL is defined in Algorithm 1.

5 Experimental Design

We evaluate whether explicit dependency modeling improves long-horizon Minecraft task execution. Our hypothesis is that frontloading long-horizon dependency reasoning into an explicit task graph will allow the SPL agent to make more reliable and efficient progress than the baseline Mindcraft agent, resulting in higher task success, lower timeout-normalized episode durations, fewer environment commands, and fewer LLM calls and lower API costs, especially on objectives with longer chains of prerequisite resources, tools, and crafting steps. To test this, we compare our SPL agent

Algorithm 1 Structured Prompting Loop

Require: Primary task T
Require: Environment $\mathcal{E}$
Require: Large language model LLM
Require: PTD prompt set $P_{\text{ptd}} = \{P_{\text{gen}}, P_{\text{val}}, P_{\text{ref}}\}$
Require: Failure-replanner prompt P_{fr}
Require: Max PTD refinement rounds n

```
G ← SelfRefinePTD(LLM, T, P_ptd, n)
repeat
    s ← GetState(E)
    G_s ← ComputeSCSG(G, s)
    if G_s = ∅ then
        return
    V_src ← GetSourceVertices(G_s)
    task ← SelectNextTask(V_src, G, s)
    status ← ExecuteTask(task, E)
    if status = fail then
        status ← RecoverFailedTask(LLM, P_fr, task, E, G)
until status = fail
```

against the baseline Mindcraft LLM-powered agent provided in the Mindcraft repository (Mindcraft Bots, 2025). The baseline condition uses the default Mindcraft agent profile, Andy. This baseline is a closed-loop LLM agent: at each decision point, the agent observes the current game state and task context, reasons through the LLM prompt, selects a next action from the available Mineflayer action interface, executes that action, and then conditions future decisions on the updated observation. Unlike SPL, the baseline does not construct a persistent Primary-Task DAG, validate or refine an explicit dependency graph, restrict action selection to currently feasible graph-source vertices, or invoke the same graph-conditioned failure-replanning module.

For each evaluation episode, the benchmark provides the agent only with a high-level natural-language task objective, such as obtain a diamond or acquire an iron ingot. The benchmark does not provide a ground-truth dependency graph, ordered subtask list, recipe chain, or hand-written plan. For SPL, the Primary-Task DAG, candidate subtasks, graph validation feedback, and recovery plans are produced internally by the agent through the structured LLM prompt interfaces described in Section 4.3. The deterministic components of SPL then condition the generated graph on the agent's observed Minecraft state and select feasible source vertices during execution. Both SPL and the baseline Mindcraft agent use the same task objective, Minecraft environment, world seed, spawn location, LLM backend, and available Mineflayer action interface; the main difference is that SPL adds an explicit generated dependency representation and state-conditioned subtask-selection loop.

5.1 Environment

Experiments are run on a vanilla Minecraft Java Edition 1.21.6 server. We test a suite of advancement-style Minecraft tasks with varying dependency lengths, including objectives such as acquiring hardware, crafting multiple tools, obtaining diamonds, collecting lava, and completing armor- or interaction-based advancements. Each agent is evaluated on each task across fifteen random world seeds. For each seed-task pair, both agents are run from the same world seed and spawn location to make the comparison as consistent as possible.

Each world is set to survival mode and peaceful difficulty. Thus, agents must gather resources, craft tools, and satisfy task prerequisites from an initially empty inventory, but do not need to handle hostile mobs. The spawn radius is set to 0 so that agents begin at the same coordinates for a given seed. All agent conditions use the same LLM model to isolate the effect of the agent architecture rather than model choice. Task descriptions are provided in Appendix D.

Figure 2: Example Minecraft environments used in the evaluation. The benchmark spans multiple randomly generated worlds, exposing agents to variation in terrain, nearby resources, navigation difficulty, and access to task-relevant blocks.

5.2 Evaluation Criteria

For each agent, task, and seed, we log task success, episode duration, and total environment commands. Task success is a binary indicator of whether the agent completed the objective before the task-specific timeout. Episode duration is reported as the percentage of the task-specific timeout used: $\tilde{d} = 100 \cdot d/T_{\text{task}}$, with failed runs assigned 100%. Total environment commands count the executable Mineflayer/Mindcraft actions issued by the agent during the episode. We use total environment commands as an indirect efficiency measure rather than as a direct dependency-error count. Poor dependency reasoning can cause an agent to issue extra commands by attempting infeasible subtasks, repeating failed actions, searching for resources before acquiring required tools, or recovering from poorly ordered execution. However, command count is not specific to dependency errors: it can also reflect navigation difficulty, environmental search, successful long-horizon execution, or early failure. We therefore interpret command count together with task success and timeout-normalized episode duration.

We report per-task averages across the fifteen seeds for each agent. The main comparison focuses on success rate, average percentage of task timeout used, and environment commands on average. We also report aggregate LLM API usage for each agent, including total API cost, total tokens, and total LLM requests. These values were captured from the OpenAI usage dashboard for the corresponding evaluation batches. These metrics correspond to the hypothesized reliability and efficiency benefits of frontloaded dependency reasoning: higher task completion, faster execution, fewer environment commands, and reduced LLM usage/API cost. We also log failure-replanning invocations to quantify how often SPL requires recovery after normal subtask execution fails.

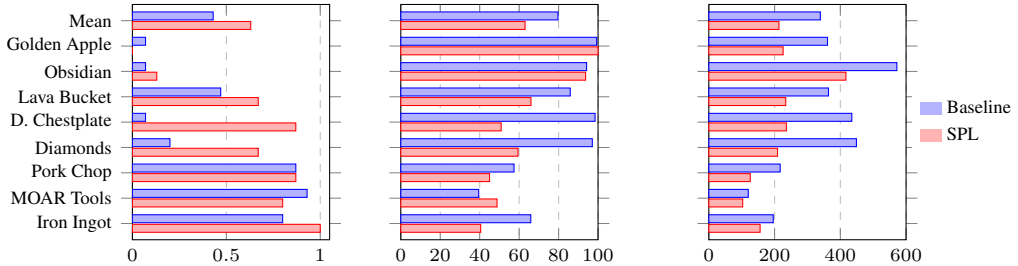
Because each SPL run is paired with a baseline run on the same task and world seed, we use paired statistical tests. We set $\alpha = 0.05$ before running the tests. For binary task success, we use a two-sided exact McNemar test over paired task-seed outcomes, which compares cases where only one of the two agents succeeds. For the aggregate task-level metrics, we use one-sided Wilcoxon signed-rank tests over paired per-task values. For task success, our hypothesized direction is that SPL will produce more SPL-only successes than baseline-only successes. For the one-sided Wilcoxon tests, the directional alternatives match our efficiency hypotheses: SPL should use a lower percentage of the task timeout and fewer environment commands than the baseline.

6 Results

We average results for each agent across fifteen seeds for each task and report suite-level means over the displayed tasks. As shown in Table 1, the SPL agent outperforms the baseline Mindcraft agent on the main aggregate metrics. Mean success rate increases from 0.43 for the baseline to 0.63 for SPL, while mean timeout used decreases from 79.7% to 63.0%. The SPL agent also uses fewer

Table 1: Per-task performance comparison between the baseline Mindcraft agent and the SPL agent. The first row displays averages over the 8 tasks across 15 seeds.

	Success Rate		Avg. Timeout Used (%)		Avg. Commands	
	Baseline	**SPL**	**Baseline**	**SPL**	**Baseline**	**SPL**
Task	**0.43**	**0.63**	**79.7**	**63.0**	**339.25**	**213.38**
Iron Ingot	0.80	1.00	65.9	40.5	196	156
MOAR Tools	0.93	0.80	39.5	48.8	120	103
Pork Chop	0.87	0.87	57.4	45.0	217	126
Diamonds	0.20	0.67	97.1	59.4	449	209
Diamond Chestplate	0.07	0.87	98.5	50.9	435	236
Lava Bucket	0.47	0.67	85.9	66.0	364	234
Obsidian	0.07	0.13	94.2	93.7	572	417
Golden Apple	0.07	0.00	99.3	100.0	361	226



(a) Success rate. (b) Timeout used (%). (c) Environment commands.

Figure 3: Per-task comparison between the baseline agent and the SPL agent. Agents are compared on success rate, average percentage of task timeout used, and average total environment commands per task.

environment commands on average, decreasing the mean from 339.25 commands per run for the baseline to 213.38 commands per run.

The clearest gains appear on longer-horizon tasks with many prerequisite steps. On `Diamonds`, the SPL agent achieves a success rate of 0.67, compared to 0.20 for the baseline. The SPL agent also completes the task faster on average, reducing percentage of timeout used from 97.1% to 59.4%, while also reducing the average number of environment commands required during the episode from 449 to 209 (Figures 3a, 3b, and 3c). A similar pattern appears on `Diamond Chestplate`, where SPL improves success rate from 0.07 to 0.87 and reduces average timeout used from 98.5% to 50.9%.

The paired tests support the aggregate trends in Table 1. For binary task success, the exact McNemar test found 34 SPL-only successes compared to 11 baseline-only successes across the paired task-seed outcomes, with 41 pairs where both agents succeeded and 34 pairs where both failed. This difference was statistically significant using the exact two-sided McNemar test ($p = 0.0008$). At the task-aggregate level, one-sided Wilcoxon signed-rank tests also found significant improvements for SPL: lower percentage of the task timeout ($W = 5.0$, $p = 0.0391$) and fewer total commands ($W = 1.0$, $p = 0.0195$). These paired tests provide preliminary statistical evidence that SPL improves task reliability and environment-level efficiency, while the aggregate API logs provide descriptive evidence that SPL also reduces LLM usage and API cost. Failure replanning was invoked in 38 of 120 SPL episodes, indicating that recovery was used in a substantial fraction of runs where normal subtask execution encountered a failure before the episode completed or timed out.

SPL also used substantially fewer LLM resources. Across the evaluation batch, the baseline used 5,031 LLM requests, 12,561,211 tokens, and $66.48 in API cost, while SPL used 420 LLM requests, 1,930,256 tokens, and $22.64 in API cost. This corresponds to reductions of 91.7% in LLM requests, 84.6% in tokens, and 65.9% in API cost. These usage totals support the hypothesis that frontloaded dependency reasoning can reduce downstream LLM usage while improving task success.

These results are consistent with our hypothesis that explicit dependency modeling is most useful when task completion requires satisfying a long chain of prerequisites. Obtaining diamonds, lava, or diamond armor requires the agent to gather resources, craft tools, acquire fuel or workstations, and execute subtasks in a valid order. Without an explicit representation of these dependencies, the baseline agent may spend more time and commands on infeasible, redundant, or poorly ordered actions. SPL, by contrast, uses the PTD and state-conditioned subgraph to select feasible subtasks, allowing it to make more reliable progress toward the final objective.

The weaker tasks suggest that SPL is currently limited less by high-level dependency ordering than by missing or incorrect grounding between the dependency graph and the action interface. `Obsidian` and `Golden Apple` each fail when a required dependency is either incorrectly modeled or not mapped to a reliable low-level behavior: water-bucket use on lava is mishandled and apples are treated as invalid block-search targets. These failures suggest that SPL is strongest when tasks can be represented as inventory-item dependency chains with valid action mappings, but future versions need richer dependency types for source grounding, interaction preconditions, temporal constraints, and non-inventory actions.

7 Future Work

The current SPL agent models task structure primarily through inventory-item dependencies. This representation works well for objectives that require resource gathering, crafting, smelting, and tool progression, but it does not fully capture broader gameplay actions such as shooting a bow, trading with villagers, navigating to a structure, or performing temporally extended interactions. Future work should extend the PTD representation to include action dependencies, location dependencies, and interaction preconditions while preserving the benefits of state-conditioned execution.

A second direction is to evaluate SPL in multi-agent settings, where agents must divide labor, share resources, communicate progress, and avoid redundant work. A shared dependency graph could support subtask selection based on each agent's inventory, local environment, and assigned role. Finally, future evaluations should use a larger task suite, more seeds, repeated trials per seed, stronger planning baselines, and ablations without PTD validation or failure replanning. Direct dependency-error annotation would also allow future work to separate dependency failures from other sources of command inefficiency, such as navigation, search, and low-level action grounding.

8 Conclusion

Embodied LLM agents are promising because they can use language models to translate high-level goals into actions, but long-horizon tasks reveal an important weakness: reasoning about what to do next is not enough when success depends on satisfying ordered prerequisites over time. This paper studied that problem through the Structured Prompting Loop (SPL), which uses an explicit dependency representation to connect long-horizon planning with closed-loop execution. Our results suggest that LLM agents become more reliable when they externalize task structure before acting, then repeatedly condition that structure on the current environment to select feasible subtasks.

More broadly, this work supports the idea that structured intermediate representations can help bridge the gap between flexible language-based reasoning and reliable action in interactive environments. The Minecraft implementation tested here is only one instantiation of this idea. Similar dependency-aware planning could be useful for other tool-using or embodied agents, including robots, web agents, software-development agents, and multi-agent systems where progress depends

on ordered subtasks, resource constraints, or shared state. The most exciting direction for future work is to move beyond inventory-item dependencies toward richer models of action preconditions, locations, temporal constraints, interaction requirements, and collaboration between agents. This would allow LLM agents not only to plan before acting, but to maintain and revise explicit task structure as the world changes.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J. Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022. URL http://arxiv.org/abs/2204.01691.

Anirudh Chari, Suraj Reddy, Aditya Tiwari, Richard Lian, and Brian Zhou. MINDSTORES: Memory-informed neural decision synthesis for task-oriented reinforcement in embodied systems, 2025. URL http://arxiv.org/abs/2501.19318.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. MineDojo: Building open-ended embodied agents with internet-scale knowledge, 2022. URL https://arxiv.org/abs/2206.08853v2.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022. URL https://arxiv.org/abs/2201.07207v2.

Mircea Lică, Ojas Shirekar, Baptiste Colle, and Chirag Raman. MindForge: Empowering embodied agents with theory of mind for lifelong collaborative learning, 2024. URL http://arxiv.org/abs/2411.12977. version: 2.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

Mindcraft Bots. Mindcraft. https://github.com/mindcraft-bots/mindcraft, 2025. Accessed: 2026-04-29.

Minecraft Wiki. Advancement. https://minecraft.wiki/w/Advancement, 2026. Accessed: 2026-04-29.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL http://arxiv.org/abs/2304.03442.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL http://arxiv.org/abs/2303.11366.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models, 2022. URL https://arxiv.org/abs/2209.11302v1.

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. URL http://arxiv.org/abs/2305.16291.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL http://arxiv.org/abs/2201.11903.

Isadora White, Kolby Nottingham, Ayush Maniar, Max Robinson, Hansen Lillemark, Mehul Maheshwari, Lianhui Qin, and Prithviraj Ammanabrolu. Collaborating action by action: A multi-agent LLM framework for embodied reasoning, 2025. URL http://arxiv.org/abs/2504.17950.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023. URL http://arxiv.org/abs/2210.03629.

Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware, 2023. URL https://arxiv.org/abs/2304.13705v1.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. 38(17):19724–19731, 2024. ISSN 2374-3468. DOI: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

Appendix

A Prompt Interface Specifications

This appendix summarizes the structured prompt interfaces used by the SPL agent. Each prompt constrains the LLM to operate over a narrow stage of the planning or recovery process and to return machine-readable JSON that can be consumed by the deterministic components of the system.

A.1 Primary-Task DAG Generation

The PTD generation prompt converts a natural-language Minecraft objective into a JSON dependency graph. The input is a primary objective string O, such as *smelt an iron ingot*, *obtain a diamond*, or *cook a porkchop*. The output is a directed acyclic graph G whose vertices are inventory items and whose edges represent prerequisite relationships.

The prompt requires the LLM to return a JSON object with four top-level fields: `objective`, `sinks`, `vertices`, and `edges`. Each vertex includes an item identifier, quantity, item type, and acquisition dependency. Each edge specifies a prerequisite item, target item, dependency type, quantity, and whether the prerequisite is consumed. The prompt restricts vertices to inventory items rather than actions, locations, or game events.

Field	Description
`objective`	Original natural-language task objective.
`sinks`	Inventory items that directly satisfy the objective.
`vertices`	Required inventory items, tools, resources, or workstations.
`edges`	Directed prerequisite relationships between inventory items.

Table 2: Required top-level fields for the PTD JSON output.

The generation prompt also defines valid vertex and edge types. Vertex types include `resource`, `item`, `tool`, and `workstation`. Edge types include `crafting_input`, `smelting_input`, `fuel_input`, `item_dependency`, `tool_dependency`, and `workstation_dependency`. For an edge $A \rightarrow B$, item A is required to obtain item B. Consumed edges represent ingredients or fuel, while non-consumed edges represent reusable requirements such as tools or workstations.

Finally, the prompt specifies graph-validity and quantity rules. The graph must be acyclic, every edge endpoint must refer to an existing vertex, every sink must have no outgoing edges, duplicate edges are disallowed, and all item ids must use lowercase `snake_case`. Quantity rules require the graph to respect Minecraft recipe batch sizes and to aggregate shared intermediate requirements when the same item is consumed by multiple downstream vertices.

A.2 PTD Validation and Refinement

The PTD validation and refinement prompts implement a self-refinement loop for improving candidate dependency graphs before execution. The validation prompt takes the original objective and a candidate PTD as input. It audits the graph for downstream-critical defects that would make the graph invalid, incomplete, or misleading for execution. The validator returns a structured JSON report with a `pass` or `fail` verdict, a list of issues, a list of possible issues, and a short summary.

The validator prioritizes errors that affect downstream execution, including malformed JSON, missing fields, invalid graph structure, incorrect sink nodes, missing prerequisites, incorrect edge types, incorrect item classifications, missing smelting dependencies, and quantity errors that affect executability. The validator is instructed to avoid rejecting a graph solely because another valid graph could also satisfy the objective.

When validation fails, the refinement prompt receives the original objective, the current candidate graph, and the validator output. It repairs only the issues identified by the validator while preserving unrelated valid vertices, edges, quantities, and sinks. If a repair changes consumed demand or graph structure, the refinement prompt recomputes only the impacted quantities and returns a corrected JSON graph.

Stage	Role in PTD self-refinement
Generation	Produces an initial JSON dependency graph from the objective.
Validation	Audits the graph for material structural, semantic, and quantity errors.
Refinement	Repairs issues while preserving valid graph components.

Table 3: Prompt stages used to construct and refine the Primary-Task DAG.

A.3 Failure-Replanning

The failure-replanning prompt is used when normal subtask execution fails. Its input consists of a failed execution trace, previous recovery diagnoses, and the available action interface. The failed trace records the selected subtask, the attempted environment actions, the observed states, the action results, and the final state reached after failure. The replanner treats the final state of the trace as the starting point for recovery.

The prompt constrains the LLM to diagnose the likely blocker and return a JSON object containing a short diagnosis and an ordered list of executable recovery actions. Recovery actions must use only the provided action interface; the replanner is not allowed to invent new actions or output abstract pseudo-actions. This constraint allows the recovery plan to be passed back to the execution system without requiring free-form interpretation.

The output schema contains two fields: `diagnosis`, a short natural language explanation of the likely failure, and `actions`, an ordered list of bounded executable actions. If a recovery attempt fails, its diagnosis is stored and supplied to the next failure-replanning call so that the system can avoid repeating unsuccessful recovery strategies.

B Example SPL Execution

This appendix provides an example of how SPL conditions a Primary-Task DAG on the current Minecraft state, selects a feasible subtask, and recovers from a failed execution attempt.

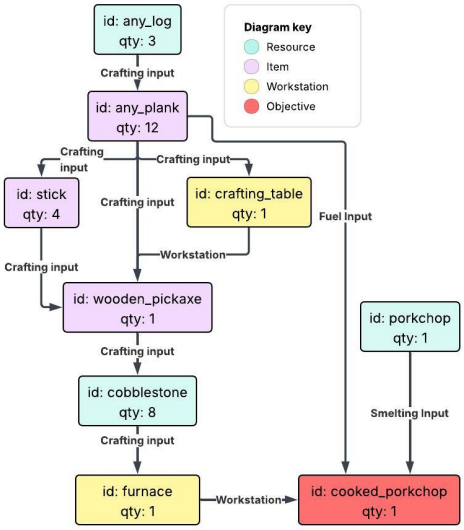
B.1 Example State-Conditioned Execution

Figure 4 shows one iteration of the structured prompting loop for the objective *cook a porkchop*. The initial PTD contains the inventory-item dependencies required to obtain and cook a porkchop, including wood, sticks, a wooden pickaxe, cobblestone, a furnace, raw porkchop, and fuel. The current state shows that the agent has already collected some wooden materials, so the state-conditioning procedure removes satisfied dependencies from the graph.

The resulting state-conditioned subgraph contains only the remaining work needed to complete the objective. Source vertices in this subgraph become candidate subtasks. In this example, the system selects a currently feasible crafting subtask and mediates it into an executable Minecraft action. After the action is executed, the SPL recomputes the state-conditioned subgraph using the updated state and continues until the objective is complete or execution fails.

B.2 Example Failure Recovery

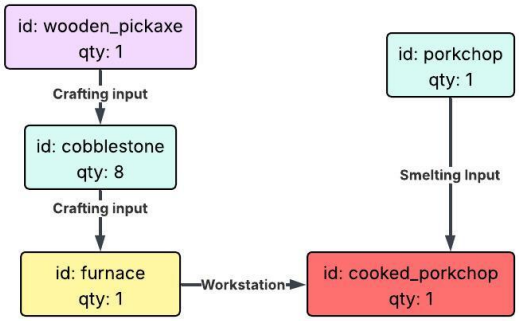
Failure replanning is triggered when the selected subtask cannot be completed by the normal execution routine. Table 4 summarizes a recovery episode for the objective *cook a porkchop*. The system



(a) Primary-Task DAG


```
"state": {
    "inventory": {
      "spruce_planks": 6,
      "stick": 4,
      "crafting_table": 1,
      "spruce_log": 1
    },
    "craftable_items": [
      "spruce_planks",
      "crafting_table",
      "wooden_sword",
      "wooden_pickaxe",
      "wooden_axe",
      "stick"
    ],
    "nearby_blocks": [
      "spruce_log",
      "sand",
      "dirt",
      "stone",
      "grass_block",
    ],
    "nearby_entities": {
      "mobs": []
    }
    }
```


(b) Current state



(c) State-conditioned subgraph


```
"candidates": [
  {
    "id": "porkchop",
    "qty": 1,
    "item_type": "resource",
    "satisfied_inputs": [],
    "source_hint": "pig",
    "source_kind": "mob"
  },
  {
    "id": "wooden_pickaxe",
    "qty": 1,
    "item_type": "tool",
    "satisfied_inputs": [
    {
      "item": "stick",
      "qty": 2,
      "type": "crafting_input",
      "consumed": true
    },
    {
      "item": "any_plank",
      "qty": 3,
      "type": "crafting_input",
      "consumed": true
    },
    {
      "item": "crafting_table",
      "qty": 1,
      "type": "workstation",
      "consumed": false
    }
    ],
    "source_hint": null,
    "source_kind": "block"
  }
  ]
```


(d) Candidate subtasks


```
{"action": "!craftRecipe(\"wooden_pickaxe\", 1)"}
```


(e) Selected task and mediated action

Figure 4: Example structured loop iteration for the objective *cook a porkchop*. The PTD is conditioned on the current state to produce the SCSG. Source vertices become candidate subtasks, and the selected task is mediated into an executable action.

initially selects the subtask of obtaining raw porkchop, but the search routine fails because no pig is found nearby. The failure replanner then uses the failed trace and previous recovery diagnoses to propose bounded recovery actions.

Stage	Observed issue	Recovery behavior
Initial failure	The agent searches for a pig at increasing radii, but no pig is found nearby.	The failed trace records the attempted searches, nearby blocks and entities, inventory state, and final failure reason.
Attempt 1	The agent appears to be searching from an area where pigs are unlikely to spawn.	The replanner moves the agent toward grass and searches again, but no pig is found.
Attempt 2	The search center has shifted, but the agent still fails to find a pig.	The replanner moves toward a nearby land feature and searches again, but the attempt still fails.
Attempt 3	Previous attempts did not move the agent far enough from the original search area.	The replanner sends the agent to a new coordinate region, searches again, finds a pig, and attacks it.
Post-recovery	The agent obtains raw porkchop.	The SPL recomputes the state-conditioned subgraph, removes the satisfied porkchop dependency, and resumes execution toward cooked porkchop.

Table 4: Condensed example of failure replanning for the objective *cook a porkchop*.

This example illustrates how the failure-replanning module differs from the main dependency loop. The PTD and SCSG determine which subtask is currently feasible, while failure replanning handles local execution problems such as search failure, navigation failure, missing nearby entities, or incorrect grounding of an abstract dependency. Once recovery changes the environment state, control returns to the main SPL loop.

C Algorithms

Algorithm 2 Primary-Task DAG Construction

Require: Primary objective O
Ensure: Primary-Task DAG G
  $G \leftarrow$ INITIALIZEGRAPH(O)
  $F \leftarrow$ INTERPRETOBJECTIVE(O)
  for all $v \in F$ **do**
    ADDSINKVERTEX(G, v)
  while $F \neq \emptyset$ **do**
    $v \leftarrow$ POP(F)
    for all $u \in$ DEPENDENCIES(v) **do**
      ADDVERTEXIFMISSING(G, u)
      ADDEDGE(G, u, v)
      if DEPENDENCIES$(u) \neq \emptyset$ **then**
        PUSH(F, u)
  return G

Algorithm 3 State-Conditioned Subgraph

Require: Primary-task DAG G, state s
Ensure: State-conditioned remaining-work subgraph.
  $V_{\text{sat}} \leftarrow \textsc{SatisfiedNodes}(G, s)$ ▷ vertices whose required quantities are satisfied
  if $\textsc{GetSinks}(G) \subseteq V_{\text{sat}}$ **then**
    return $\emptyset$
  $G' \leftarrow \textsc{UpdateRemainingQuantities}(G, s)$
  $G' \leftarrow G' \setminus V_{\text{sat}}$
  repeat
    $V_{\text{gd}} \leftarrow \textsc{GetSinks}(G') \setminus \textsc{GetSinks}(G)$ ▷ goal-disconnected vertices
    $G' \leftarrow G' \setminus V_{\text{gd}}$
    $V_{\text{sat}} \leftarrow \textsc{SatisfiedNodes}(G', s)$
    $G' \leftarrow \textsc{UpdateRemainingQuantities}(G' \setminus V_{\text{sat}}, s)$
  until $V_{\text{sat}} = \emptyset \wedge V_{\text{gd}} = \emptyset$
  return G'

Algorithm 4 PTD Self-Refinement

Require: Primary task T
Require: Large language model LLM
Require: Prompt set $P_{\text{ptd}} = \{P_{\text{gen}}, P_{\text{val}}, P_{\text{ref}}\}$
Require: Max refinement rounds n
Ensure: Refined Primary-Task DAG G
  $G \leftarrow \textsc{GeneratePTD}(\text{LLM}, T, P_{\text{gen}})$
  for $i = 0$ to n **do**
    $F \leftarrow \textsc{ValidatePTD}(\text{LLM}, T, G, P_{\text{val}})$
    if F passes **then**
      return G
    if $i < n$ **then**
      $G \leftarrow \textsc{RefinePTD}(\text{LLM}, T, G, F, P_{\text{ref}})$
  return G

D Task Descriptions

We created eight inventory based-tasks based on in-game Minecraft advancements. Tasks had variable timeout lengths based on the complexity of the objective.

1. **Task ID:** acquire_hardware; **Name:** Iron Ingot; **Goal:** Smelt an Iron Ingot. Have an iron ingot in the inventory.; **Timeout (s):** 900
2. **Task ID:** hot_stuff; **Name:** Lava Bucket; **Goal:** Fill a Bucket with lava. Have a lava bucket in the inventory.; **Timeout (s):** 1200
3. **Task ID:** diamonds; **Name:** Diamonds; **Goal:** Acquire diamonds. Have a diamond in the inventory.; **Timeout (s):** 1200
4. **Task ID:** moar_tools; **Name:** MOAR Tools; **Goal:** Construct one pickaxe, one shovel, one axe, and one hoe with the same material.; **Timeout (s):** 600
5. **Task ID:** pork_chop; **Name:** Pork Chop; **Goal:** Cook a Pork Chop.; **Timeout (s):** 900
6. **Task ID:** cover_me_in_diamonds; **Name:** Diamond Chestplate; **Goal:** Acquire a diamond chestplate. Have a diamond chestplate in the inventory.; **Timeout (s):** 1500
7. **Task ID:** ice_bucket_challenge; **Name:** Obsidian; **Goal:** Obtain a block of Obsidian. Have a block of obsidian in the inventory.; **Timeout (s):** 1800

8. **Task ID:** oh_shiny; **Name:** Golden Apple; **Goal:** Craft a golden apple. Have a golden apple in the inventory.; **Timeout (s):** 1500