

MineNPC-Task: Task Suite for Memory-Aware Minecraft Agents

Tamil Sudaravan Mohan Doss
Microsoft
tsudaravann@microsoft.com

Michael Xu
Microsoft Research
United States
michaelxu@microsoft.com

Sudha Rao
Microsoft Research
United States
Sudha.Rao@microsoft.com

Andrew D. Wilson
Microsoft Research
United States
awilson@microsoft.com

Balasaravanan Thoravi
Kumaravel
Microsoft Research
United States
bala.kumaravel@microsoft.com

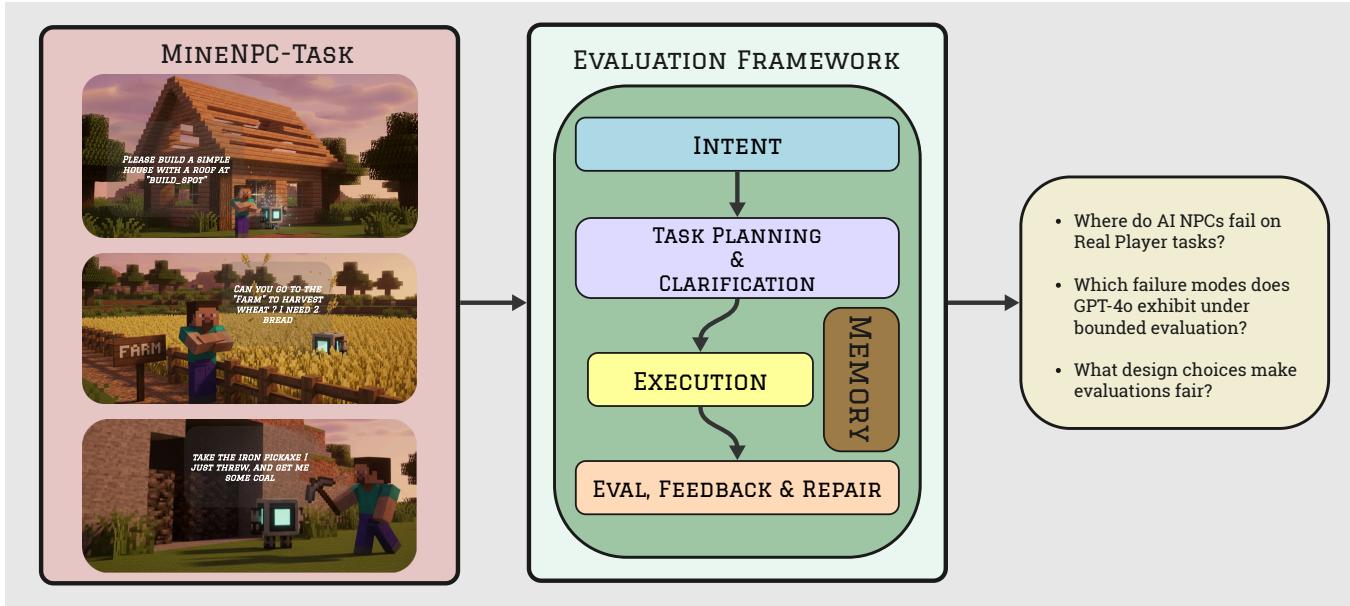


Figure 1: (a) MINENPC-TASK: Each row depicts a naturalistic, player-authored request in Minecraft, grounding evaluation in distinct phases of co-play building, farming, and mining. (b) Bounded, reproducible evaluation: a model-agnostic harness that routes intents, plans with single-turn clarification, executes via public Mineflayer APIs under a bounded-knowledge policy, and judges only from in-world evidence with lightweight memory. (c) Questions this benchmark enables: Where do AI NPCs fail on real player tasks? Which failure modes does GPT-4o exhibit under this bounded setup? Which design choices make evaluations fair?

Abstract

We present MINENPC-TASK, a user-authored benchmark and evaluation harness for testing memory-aware, mixed-initiative LLM agents in open-world *Minecraft*. Rather than relying on synthetic prompts, tasks are elicited from formative and summative co-play with expert players, normalized into parametric templates with explicit preconditions and dependency structure, and paired with machine-checkable validators under a bounded-knowledge policy that forbids out-of-world shortcuts. The harness captures plan/act/memory events—including plan previews, targeted clarifications, memory reads and writes, precondition checks, and repair attempts and reports outcomes relative to the total number of attempted subtasks, derived from in-world evidence.

As an initial snapshot, we instantiate the framework with GPT-4o and evaluate 216 subtasks across 8 experienced players. We

observe recurring breakdown patterns in code execution, inventory/tool handling, referencing, and navigation, alongside recoveries supported by mixed-initiative clarifications and lightweight memory. Participants rated interaction quality and interface usability positively, while highlighting the need for stronger memory persistence across tasks. We release the complete task suite, validators, logs, and harness to support transparent, reproducible evaluation of future memory-aware embodied agents.

1 Introduction

Building truly capable AI companions for open-world games requires more than one-shot instruction following, it requires agents that can *plan*, *clarify*, and *remember* in order to operate effectively in dynamic, long-horizon environments. However, existing evaluation practices fall short: many rely on overly prescriptive prompts

or grant agents privileged access to hidden environment state, artificially inflating their apparent competence and undermining fair comparison across models. This highlights the need for a *transparent, reproducible benchmark* that captures the pressures of mixed-initiative interaction [15], without granting unfair advantages. Recent efforts, such as the Minecraft Universe (MCU) benchmark [1, 2], move in this direction by introducing scalable, composable tasks in open-world environments and emphasizing repeatable, human-aligned evaluation.

We introduce MINENPC-TASK, a practical benchmark for evaluating memory-aware, mixed-initiative LLM agents in *Minecraft*. Rather than synthetic prompts, tasks are elicited from expert co-play: during our formative and evaluation sessions, experienced players issued real requests, which we normalized into compact templates with explicit preconditions and a small set of slot parameters, and paired with simple machine-checkable validators. The benchmark runs inside a Mineflayer envelope so perception and action are limited to public, in-game APIs; a bounded-knowledge policy forbids admin commands, global map introspection, and scans beyond loaded chunks. The aim is straightforward: a clean, reusable setup that others can run, swap in different models, and obtain comparable numbers, while avoiding hidden shortcuts that have complicated evaluation in prior *Minecraft*-based agents [8, 13, 35].

To execute tasks reproducibly we use a model-agnostic evaluation framework (Section 4). The agent presents a brief plan preview that breaks the request into a handful of subtasks, asks a targeted clarifying question only when a slot is unbound (for example, a tool variant or a search radius), acts through Mineflayer skills, and is judged solely on in-world evidence drawn from inventory and equipment deltas, position changes, nearby entities and blocks within loaded chunks, and recent chat. This mixed-initiative pattern aligns with recent agent designs that combine planning with lightweight memory and reflection while keeping behavior legible to human partners [10, 22, 26]. Short scenario walkthroughs (Section 5) illustrate how planning, clarification, and reuse of prior context work together.

As an initial snapshot, we instantiate the framework with GPT-4o and run live co-play with 8 experienced players (Section 6). Across 44 user-authored tasks and 216 subtasks, we observe 71 subtask failures (approximately 33%) and report where things went wrong, including code execution, inventory handling, referencing, and navigation, alongside common recoveries such as clarifying a slot, simplifying a goal, or constraining location. Participants rated interaction quality and interface usability positively, with mixed views on personalization and completion, consistent with the need for stronger memory scaffolding. We intentionally omit ablations or model comparisons. The intent is to provide a compact, user-authored benchmark that others can extend. For context, our focus complements broader surveys on AI in games and adaptive interaction [25, 39] and ongoing efforts to evaluate reasoning in live games [11].

The following are the contribution of this work:

- **MINENPC-TASK suite:** a user-authored benchmark for *Minecraft*, normalized into templates with explicit preconditions and paired with simple, machine-checkable validators under a bounded-knowledge policy. See Appendix E and Section 3.

Compared with prior *Minecraft* agents and datasets, we emphasize human-elicited goals, public-API constraints, and validator-backed judging [8, 13, 35].

- **Evaluation framework:** a model-agnostic procedure that enforces plan previews and single-turn clarifications when required, constrains perception and action to Mineflayer APIs, and produces validator-backed outcomes suitable for controlled comparisons across LLMs. See Section 4. The design follows calls for transparent, reproducible evaluation in interactive systems [11, 12].
- **Empirical snapshot with GPT-4o:** co-play results from 8 experts over 44 tasks, comprising 216 subtasks with 71 failures (approximately 33%), along with observations about where mixed-initiative interaction and lightweight memory help—and where brittleness remains. See Section 6.

2 Related Work

Benchmarks and evaluation for embodied agents. A long line of embodied benchmarks has standardized APIs, sensors, and success criteria for instruction following, navigation, and manipulation. Suites that span vision, language, and action such as ALFRED [28], TEACH [21], and EmbodiedQA [6] probe multistep execution under natural language, often requiring dialog to bind underspecified slots. Platform efforts including Habitat, iTHOR, and ProcTHOR provide large scale scene generation and reproducible perception and action loops [7, 14, 27], while BEHAVIOR and VirtualHome emphasize program like decompositions of everyday activities and structured evaluation protocols [23, 30]. Text centric environments and curricula such as TextWorld, ALFWORLD, ScienceWorld, and BabyAI offer controllable abstractions for reasoning, exploration, and sample efficiency [4, 5, 29, 36]. Beyond simulators, LLM agent evaluations study open ended tool use and web interaction (Agent-Bench, WebArena) [18, 41], and analyze reasoning in live games and across modalities [11, 19, 39]. Collectively, these efforts advanced reproducibility and coverage, while differing in reliance on scripted versus user authored goals and on privileged versus in situ evidence for judging.

LLM agents and NPCs in games *Minecraft* has served as a versatile laboratory for grounded learning, planning, and human-AI teaming. Malmo established a widely used interface and experimental substrate [9, 13]. LLM-driven systems leverage internet-scale priors for broad competence: MineDojo aggregates cross-modal knowledge and goals [8]; Voyager demonstrates autonomous skill acquisition and reuse in open-ended play [35]; STEVE-1 targets text-to-behavior generation [17]; and Ghost in the *Minecraft* augments agents with text-based knowledge and memory [42]. Parallel threads examine runtime code generation for gameplay and NPC behaviors [12, 34] and conversational NPCs that assist with scripted quest structures [24]. Outside *Minecraft*, game-based evaluations continue to investigate agent reasoning, adaptation, and spectator-facing dynamics across genres and tasks [3, 39].

Mixed-initiative planning, clarification, and memory. Mixed-initiative pipelines and prompting strategies interleave thinking with acting, emphasizing targeted questions when parameters are unbound and short, legible plans. Dialog-driven instruction following foregrounds clarification for slot binding (e.g., TEACH) [21], and

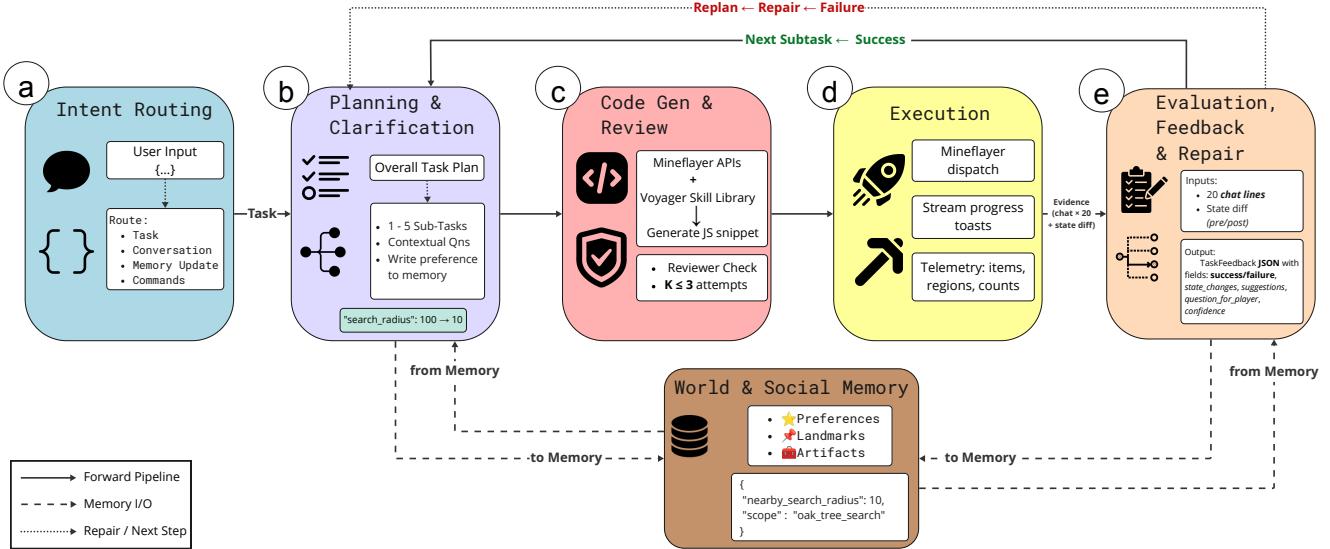


Figure 2: Plan-Clarify-Act-Judge: our model-agnostic evaluation framework. (a) *Intent routing* parses chat into {intent, slots, confidence}. (b) *Planning and clarification* compiles a short plan (3–5 steps); if a required slot is missing, the agent issues a single, contextual question. (c) *Code generation and review* synthesizes a small JavaScript snippet against Mineflayer APIs and a skill library; a lightweight reviewer caps retries ($K \leq 3$). (d) *Execution* dispatches approved code and streams concise progress updates. (e) *Evaluation and bounded repair* reads recent chat and state deltas to emit TaskFeedback; on success the harness advances to the next subtask, and on failure it offers a bounded repair and partial replan. Dashed arrows denote reads/writes to *memory* (landmarks, artifacts, preferences, commitments).

prompting methods such as ReAct integrate reasoning traces with tool calls [40]. Foundational agent models formalize state, intentions, and cooperation [38], while cognitive distinctions between episodic and semantic memory inform how agents might store and retrieve context [31, 32]. Recent LLM systems investigate experience distillation and dynamic memory consolidation for situated tasks [10, 26]. Interaction-design perspectives highlight dynamic grounding and constructive negotiation for aligning human-agent work [33], and studies of theory-of-mind cues examine coordination in multi-agent settings [16]. Complementary literatures on NPC believability, social presence, and anthropomorphism provide additional frames for evaluating expectations in game worlds [20, 37].

3 Benchmark Setting and MINENPC-TASK

We instantiate open-world co-play in *Minecraft* using a Mineflayer client so that actions are constrained to public, in-game interfaces. Concretely, the agent observes chat, its own inventory and equipment, and nearby entities/blocks within currently loaded chunks (a practical proxy for line of sight). Actions are issued through high-level skills like *navigate*, *mine*, *craft*, *place*, *interact* implemented atop Mineflayer APIs. A bounded-knowledge policy forbids privileged capabilities (e.g., `/give`, `/teleport`), global map/seed introspection, and scans beyond loaded chunks; runs that violate this policy are invalidated.

Task source and structure. The MINENPC-TASK suite is derived from goals observed during expert co-play rather than from synthetic prompts. From these observations we specify a light-weight *task template* that the evaluation framework instantiates at

run time. Given a user goal, the framework compiles a short plan (typically 3–5 subtasks) and represents each subtask as a compact record with five fields: name, dependencies, required parameters, a single targeted clarifying question issued only if a required parameter is missing, and a success criterion. Completion is judged by simple, machine-checkable validators that consume only bounded, in-world evidence—inventory/equipment and position deltas, nearby entities/blocks within loaded chunks, and a short window of recent chat—and return a pass/fail with a brief rationale. (Prompts and template examples appear in the Appendix.)

Coverage. The MINENPC-TASK suite covers the everyday goals we observed most frequently: *resource acquisition and logistics* (gather–craft chains, inventory constraints); *navigation and retrieval* (recalling named landmarks and fetching/returning items); *tooling and crafting* (multi-step recipes, tool selection with durability checks); *construction* (layout-constrained builds using existing facilities); *combat and safety* (simple loadout preconditions); and *continuity/mixed-initiative* cases where an underspecified slot is bound via a brief clarification. Section 4 details how the evaluation framework enforces this template end-to-end and produces validator-backed outcomes.

4 Evaluation Framework

We now describe the evaluation framework that turns the constraints and task templates into a reproducible harness for testing LLM agents in *Minecraft* via Mineflayer. Figure 2 summarizes the pipeline. The framework is model-agnostic: an LLM proposes a short plan, asks at most one targeted clarifying question when a required parameter is unbound, generates Mineflayer-compatible

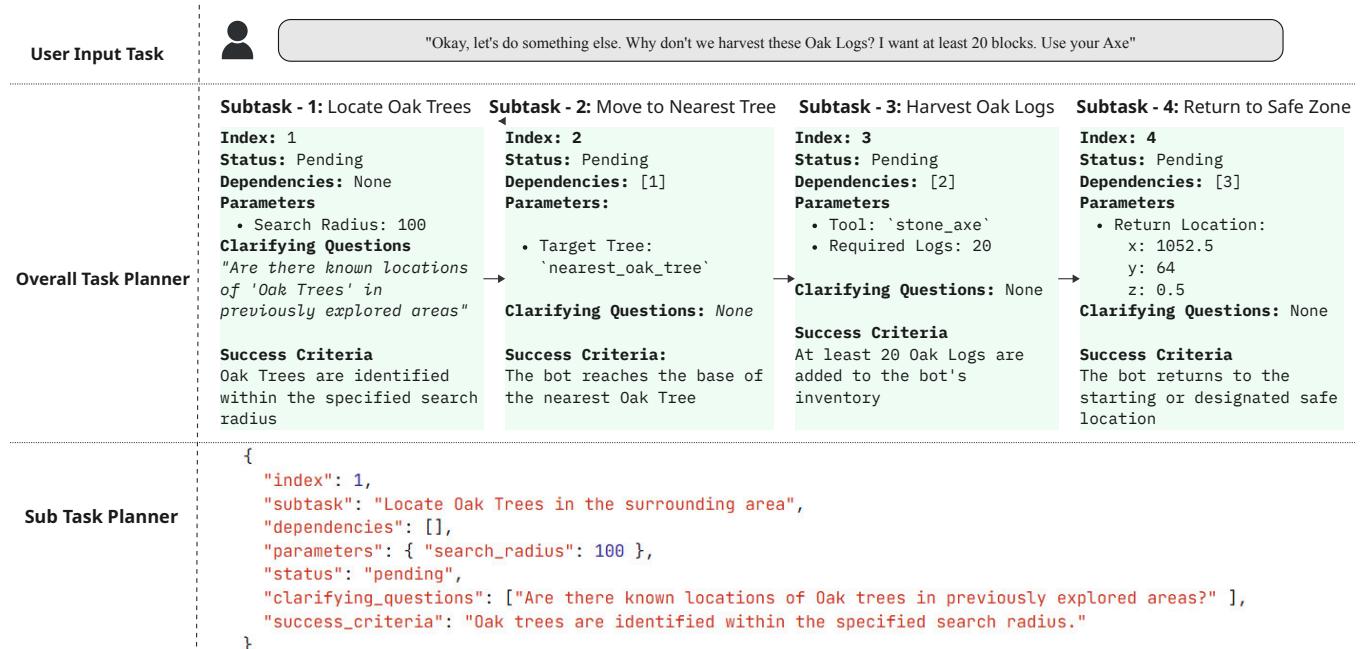


Figure 3: Planning surfaces for “harvest oak logs.” (a) Chat request routed to task(request). (b) Short, legible plan with dependencies. (c) Subtask record with defaults (e.g., search_radius=100) and its clarifying question.

code to act, and is judged from bounded, in-world evidence. In this paper we instantiate the framework with GPT-4o; Section 6 reports that snapshot.

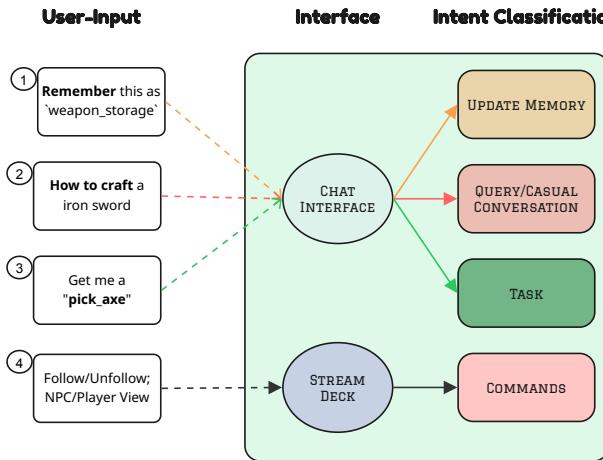


Figure 4: Routing and planning. Ingress via chat or control buttons is parsed into intents (*update memory, conversation, task, control*). Tasks flow to a planner that decomposes requests and binds missing slots with a single, contextual question when needed.

Observation and action contract. Perception and control are constrained to public, in-game interfaces exposed by Mineflayer. The agent can read recent chat, its inventory and equipment, and nearby entities and blocks within currently loaded chunks (a practical proxy for line-of-sight). It can navigate, mine, craft, place, interact, transfer items, and return/drop off; destructive operations

require an explicit confirmation step. A bounded-knowledge policy forbids admin commands (e.g., /give, /teleport), global map/seed introspection, and bulk scans beyond loaded chunks; attempts that trigger a forbidden call are marked invalid. See Fig. 4 for the routing pathway that feeds planning.

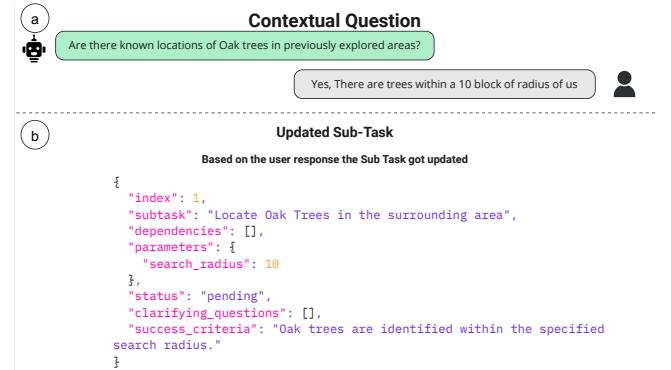


Figure 5: Mixed-initiative example. (a) The planner asks whether oak trees are within a known radius; the player replies “within 10 blocks.” (b) The plan updates search_radius from 100 to 10 and stores the preference with provenance told.

Plan preview and clarification. Player requests arrive through chat and are routed to task(request) when they imply action. Before any execution, the framework renders a one-line plan preview and exposes precondition checks (Fig. 3). If a required slot (e.g., tool variant, drop-off location, or search radius) is not bound by context, the agent asks *one* targeted question rather than guessing; the answer is applied immediately to the active plan and written to memory with provenance for later reuse (Fig. 5).

From template to plan. Given a user goal and the corresponding task template (Appendix E), the framework compiles a short plan with three to five subtasks (Fig. 3(b)). Each subtask is represented by a compact record (Fig. 3(c)) specifying its name, dependencies, required parameters, a clarifying question issued only when a required parameter is missing, and a success criterion. These structures are logged to make plan deltas auditable (e.g., search_radius: 100 → 10).

Execution with review. Plans execute through generated JavaScript against Mineflayer. A lightweight reviewer checks basic API usage and guard conditions; retries are capped ($K=2\text{--}3$) to prevent runaway loops. Approved code is dispatched, and the runtime emits compact progress toasts so players can track what is happening without breaking flow (Fig. 6).

```

(a) Generated Code


```

async function locateOakTees(bot) {
 bot.chat("Starting to locate oak trees in the area...");
 const searchRadius = 10, oakLogId = mcData.blocksByName['oak_log'].id;
 let oakTreesFound = false;

 await exploreUntil(bot, new Vec3(1, 0, -1), 60, () => {
 const b = bot.findBlock({matching: oakLogId, maxDistance: searchRadius});
 if (b) {
 bot.chat(`Oak tree found at ${[b.position.x, b.position.y, b.position.z]}`);
 oakTreesFound = true;
 return true;
 }
 return false;
 });

 if (!oakTreesFound) bot.chat("No oak trees found within the search radius.");
}

```



(b) Generated Review


```

{
 "valid": true,
 "confidence": 0.95,
 "comments": [
 {
 "line": 11,
 "comment": "It's better to not set 'oakTreesFound' as 'true' within the exploreUntil callback. Instead, rely completely on the return value of the callback for this information."
 }
]
}

```


```

Figure 6: Code generation and review. (a) Generated JavaScript reflects updated parameters (e.g., `searchRadius = 10`) and emits legible chat updates. (b) Reviewer feedback flags robustness issues before execution; retries are capped.

Judging and bounded repair. Completion is judged only from signals available inside the game: pre/post snapshots of inventory, equipment, and position; the presence of nearby entities or blocks within currently loaded chunks; and a short window of recent chat. The evaluator produces a structured TaskFeedback record that states success or failure, cites the state changes it relied on, and

may include a brief suggestion or a targeted follow-up question when additional context is required (Fig. 7). On success, the next subtask begins. On failure or low confidence, the framework presents a small, bounded repair prompt and *the player chooses what happens next*: for example, retry with an adjusted parameter, backtrack to a prior substep, or pause for guidance. If the player issues a new or revised goal, the router treats it as a fresh task(request) and compiles a new plan; otherwise the harness performs a partial replan from the failing step using the player’s selection. This keeps the loop explicitly mixed-initiative: the agent diagnoses and proposes, the player decides, and both the decision and plan deltas are logged.

Memory. A simple typed store persists named landmarks, artifacts, preferences, and records of commitments/breakdowns. Each entry carries provenance (*seen*, *told*, *inferred*) and can be retrieved with nearest- k queries scoped to the current task. Answers to clarifying questions are written as scoped preferences that seed future slot values when context is similar.

Reproducibility and limits. For each request we log routing latency, plan build time, whether a clarifying question was issued and answered, plan deltas, code-generation and review iterations, execution time, retries and repairs issued/accepted, success/attempts denominators, memory reads/writes (including preference hits/misses), and token usage per stage. Prompts, planner templates, validator stubs, and adapter scaffolding are provided to support replication. Two practical limits are visible in logs: conversational turns can be misrouted as tasks at low confidence (we fall back to a compact keyword heuristic), and scoped preferences can go stale when the world changes off-screen (we mark such entries “stale” and reconfirm before reuse).

Together, the components above define a complete execution loop for the agent. The next section demonstrates how this framework operates in practice through scenario walkthroughs. These examples focus on typical interactions and highlight how planning, clarification, execution, and judgment unfold step by step.

(a) NPC View Image & Chat Logs

Task Evaluator

SpectateBot: Explore success.
SpectateBot: Locating the nearest oak tree to move to.
SpectateBot: Oak tree found at (1051, 64, -2).
SpectateBot: Explore success.
SpectateBot: Moving to the oak tree at (1051, 64, -2).
SpectateBot: Arrived at the oak tree.
SpectateBot: Preparing to harvest oak logs..
SpectateBot: Searching for oak logs..
SpectateBot: Explore success.
SpectateBot: Moving to oak log at (1051, 64, -2).
SpectateBot: 1 oak logs collected.
SpectateBot: Searching for oak logs..
SpectateBot: Explore success.

(b) Bot State Change

Movement

- Previous Position: (x: 1052.5, y: 64, z: -0.5)
- Current Position: (x: 1052.5, y: 64, z: -0.5)
- Moved? No
- Speed Change: 0.00

Health

- Previous: 20/20
- Current: 20/20
- Damage Taken? No

Hunger

- Previous: 20/20
- Current: 20/20

Inventory & Equipment

- Previous Equipment: polished_diorite, wheat_seeds, oak_planks, oak_log, cobblestone, crafting_table, wooden_pickaxe, redstone, furnace, sandstone
- Current Equipment: polished_diorite, wheat_seeds, oak_planks, oak_log, cobblestone, crafting_table, wooden_pickaxe, redstone, furnace, sandstone
- Inventory Changes:
 - New Items Acquired: None
 - Items Lost: None
- Nearby Entities
 - Player: 4.63 blocks away

(c) Task Feedback

Success: The bot successfully located oak trees within the specified search radius without any issue.

Failure: null

state_changes: [

- key: "Position", previous: "(x: 1052.5, y: 64, z: -0.5)", current: "(x: 1052.5, y: 64, z: -0.5)", description: "The bot's position did not change, indicating it did not need to move to locate the oak tree."

suggestions: "Continue using the same method for locating other resources as it seems effective."

question_for_player: null, is_task_completed: true, final_result: "The bot accurately identified an oak tree within the search radius, successfully completing the given sub-task.", confidence_score: 10

Figure 7: Judging and repair. (a) Evaluator consumes bounded evidence: a short window of recent chat and an NPC camera frame. (b) State deltas between pre/post snapshots (position, inventory/equipment, nearby entities/blocks). (c) Structured TaskFeedback with success/failure and rationale, state changes, suggestions for the player, and a confidence score.

5 Scenario Walkthroughs of Thor

These walkthroughs show how the system is intended to behave when the framework and model work well together. They illustrate plan previews, brief clarifying questions, in-game execution through Mineflayer, and outcome judgment by the validator. The walkthroughs are meant to be illustrative, not evaluative; empirical results, including failures, are reported in Section 6.

5.1 S1: Bookmark a landmark, then fetch a tool

Memory write. During casual exploration, the player types: “remember this as weapon_storage.” The utterance is routed as memory_update (Fig. 4) and stored as a named landmark with coordinates and provenance told. The interface remains legible throughout. Primary play stays in the main window, while a small companion camera indicates the agent’s current view (Fig. 10).

Request and plan preview. Later, at a mine entrance, the player issues: “get me a pick_axe.” The router classifies this as task(request), after which the planner compiles a short plan and renders a one-line preview (Fig. 3): go to weapon_storage → select a pickaxe → return. The active subtask exposes its parameters and defaults for inspection (Fig. 3(c)).

Single clarification. When multiple pickaxes satisfy the request, the planner asks exactly one targeted question (Fig. 4): “Which pickaxe, iron or diamond?” The response is applied immediately to the active subtask and persisted as a scoped preference with provenance told.

Execution and judgment. Mineflayer code is generated, reviewed, and executed with capped retries (Fig. 6). Completion is judged using bounded in-world evidence, including state deltas, nearby entities or blocks, and a short window of recent chat (Fig. 7). On success, the plan advances. Otherwise, a bounded repair prompt allows the player to choose how to proceed.

Logged. Routing and plan-build latency; clarification issued and answered; memory reads and writes; code-generation and review iterations; execution time; validator decision and rationale.

5.2 S2: Collect 20 oak logs

Request and defaults. The player requests: “collect 20 oak logs.” The planner previews a compact plan in chat (Fig. 3): locate trees → move to nearest tree → harvest to count=20 → return. With no relevant prior context, the first subtask adopts a conservative default search_radius of 100 (Fig. 3(c)).

Contextual update. Before acting, the planner asks a single contextual question (Fig. 5(a)): “Are there known oak trees nearby?” The reply, “within 10 blocks,” updates the active plan in place (search_radius 100 → 10) and is written as a scoped preference for future oak-tree searches (Fig. 5(b)).

Execution and judgment. Generated code reflects the updated parameter and emits legible progress updates during execution (Fig. 6). A lightweight review flags robustness issues before dispatch. The evaluator produces a structured TaskFeedback record from bounded evidence (Fig. 7). On success, execution proceeds to the next subtask. On failure or low confidence, the player selects a bounded repair such as retrying with an adjusted parameter, backtracking, or revising the goal.

Logged. We record plan preview latency, whether a clarification was issued and answered, exact plan deltas, distance to first harvest,

time to completion, memory preference writes, and the validator’s decision and rationale.

Together, these scenarios capture the benchmark’s intended interaction loop: *preview* a short plan, *bind* missing slots with a single targeted question, *execute* within Mineflayer’s envelope, and *judge* outcomes from bounded in-world evidence, while keeping the human explicitly in the loop for recovery when execution does not go as planned.

6 Evaluating the Framework with GPT-4o and Expert Players

Having established the harness, validators, memory modules, and UX pipeline, we instantiated the complete framework with GPT-4o and conducted real-time co-play sessions with experienced Minecraft players. The evaluation goal is intentionally modest: characterize how the system behaves end-to-end on user-authored tasks under bounded, reproducible judging—*no ablations, no cross-model comparisons, and no speculative claims*.

Failure Categories Across Minecraft Task Categories

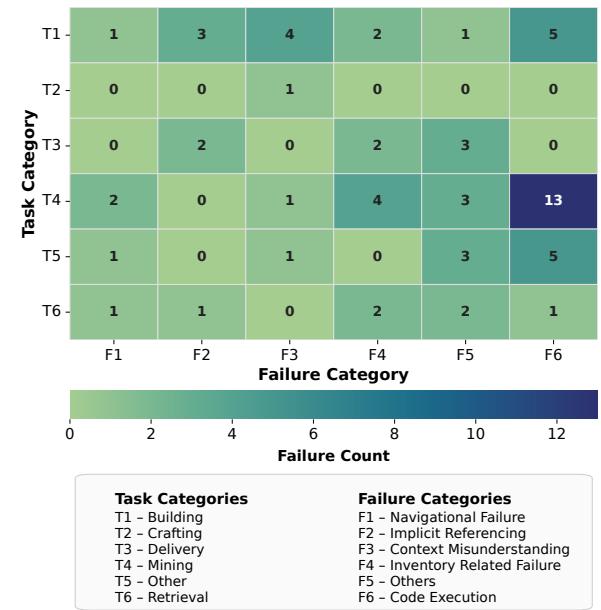


Figure 8: Cross-tabulation of failure categories by Minecraft task type. Code/inventory issues cluster around mining and construction tasks, while referencing failures and context misunderstandings appear more frequently during retrieval and navigation.

Setup. Each session included: (1) a brief interface walkthrough; (2) guided demonstrations covering landmark recall, retrieval, resource gathering with failure recovery, and contextual suggestion/-tool awareness; (3) a participant-defined task; and (4) an exit survey. The harness logged routing and planning traces, clarifications, memory reads/writes, execution events, code-generation attempts, validator outputs, and synchronized screen/audio recordings.

Denominator. Across 216 subtasks attempted by 8, we observed 71 subtask failures, yielding a $71/216 \approx 33\%$ subtask-level failure rate. All results below reflect only these observed traces.

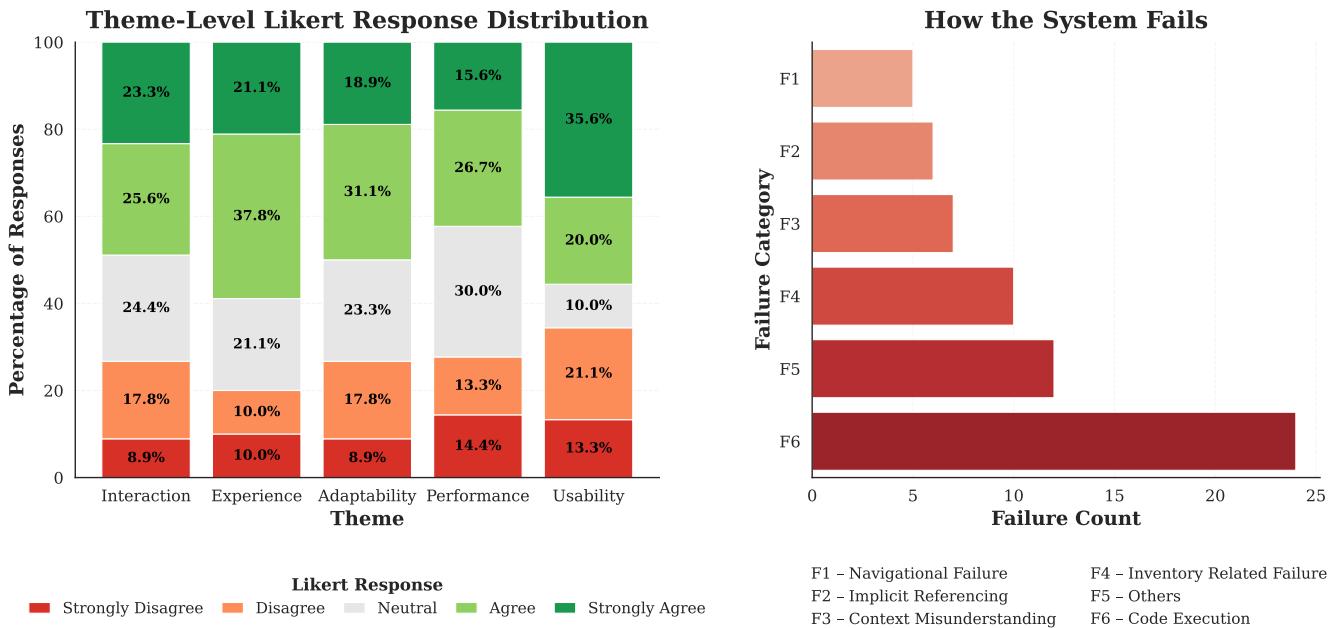


Figure 9: Player experience and system breakdown patterns. *Left:* Likert-scale responses from participants across interaction quality, memory utility, task performance, and UI usability. *Right:* Distribution of breakdown types across 216 subtasks including code/execution faults, inventory/tool misuse, referencing errors, context misunderstandings, and navigation failures.

6.1 Aggregate Engagement

Across participants, 44 high-level tasks were attempted ($M = 4.9$ per user), spanning building, mining, retrieval, navigation, and crafting. Despite breakdowns, most users completed goals after one or more attempts by revising commands or collaboratively troubleshooting with the interface.

6.2 Perception and Interaction Quality

Participants reported generally positive impressions of interaction quality and interface usability (Fig. 9).

Interaction and Communication received agreement from 7/8 users, while both **Usability & Overall Experience** exceeded 75% agreement. Memory-based recall was cited as helpful by 6/8 participants, though several requested stronger persistence (e.g., “If I correct it once, it should remember next time.”).

6.3 Breakdown Patterns and Recoverability

The framework surfaces failures without assuming causality. Recurring patterns included:

- **Code/execution failures** ($n = 24$): invalid parameters (e.g., Vec3, misfired triggers, NaN resource counts).
- **Inventory/tool issues** ($n = 10$): missing or misused tools (e.g., harvesting with a pickaxe); observed across 4 participants.
- **Context misunderstandings** ($n = 7$): ambiguous or unclear semantics.
- **Referencing failures** ($n = 6$): deictic phrasing such as “the block I’m looking at.”

- **Navigational failures** ($n = 5$): shifting or ill-defined spatial targets.

Task-type clustering patterns are summarized in Fig. 8. Participants often recovered from failures: 5/8 successfully completed a previously failed task by simplifying goals, clarifying parameters, or guiding execution manually (e.g., specifying coordinates or narrowing permissible action ranges).

6.4 Transparency and Language Alignment

Participants were more forgiving when failures were accompanied by contextual reasoning (e.g., “NaN cobblestone needed—could you clarify?”). Over time, players refined commands—adding coordinates, naming tools or furnaces, specifying variants—while the agent initiated targeted clarifications (“Which pickaxe?”). This co-adaptation produced a small but meaningful learning effect.

6.5 Expectations of Companionship and Shared Perception

Players frequently anthropomorphized the agent and expected shared visual grounding (“Look where I’m looking”). Several requested visibility into memory contents to bridge misunderstandings, suggesting that surfaced memory may meaningfully support trust and alignment.

6.6 Design Implications

- **Transparent feedback fosters resilience.** Short explanations encouraged users to persist after breakdowns.
- **Language alignment is mutual.** Players refined phrasing; agents should reciprocate with clarifications and summaries.

- **Memory visibility builds trust.** Surfacing remembered facts reduces confusion about what the agent knows.
- **Recoverability matters more than perfection.** Bounded repair loops and partial replans are effective levers.
- **Social presence matters.** Egocentric camera cues and responsive behaviors strengthened the feeling of a companion agent.

We report only the collected observations and denominator-aware outcomes. The framework is publicly released to enable reproducible evaluation on user-authored tasks under identical conditions.

7 Conclusion

We introduce MINENPC-TASK, a practical benchmark for evaluating mixed-initiative, memory-aware LLM agents in *Minecraft* using only public in-game interfaces.

Tasks are elicited from expert co-play, normalized into compact templates with explicit preconditions, and paired with simple validators that judge completion from bounded, in-world evidence.

An initial snapshot with GPT-4o over 44 tasks (216 subtasks) highlights where agents struggle slot clarification, preconditions, memory reuse, and code execution yielding a ~33% subtask failure rate.

We hope this benchmark encourages models and methods that plan across dependent steps, ask when uncertain, and ground memory in observable state without relying on hidden shortcuts.

8 Limitations

Scope. Results apply to *Minecraft* with a Mineflayer client under a bounded-knowledge policy. Portability to other engines or sensing/action envelopes is not evaluated.

Model coverage. We report a single-model snapshot (GPT-4o). There are no ablations or cross-model comparisons, so relative performance remains an open question.

Task coverage. The suite is expert-elicited and modest in size. It captures common goals we observed but does not exhaust the space of open-world play; selection bias is possible.

Measurement granularity. Validators return pass/fail from in-world traces. This improves reproducibility but under-represents partial progress and does not yet score cost or efficiency.

Run-time variance. Live co-play introduces practical variability (e.g., pathing, chunk loading). Policies reduce but do not eliminate noise; we report denominators to contextualize outcomes.

Interaction design choices. The framework prioritizes a single clarifying question and short plan previews. Richer multi-turn clarification or alternative planning styles may yield different results.

Telemetry and auditability. Released logs emphasize end-state evidence and high-level traces. Finer-grained diagnostics (e.g., pre-execution static checks) are limited and left to future iterations.

Generalization of corrections. While preferences and names are stored, persistent generalization of user corrections across sessions is limited in the current release.

9 Future Work

Comparable baselines. Add multi-model evaluations under identical perception-action contracts and policies, enabling controlled comparisons and public leaderboards.

Expanded task pool. Grow the expert-derived templates with parameterized variants and richer dependencies (e.g., multi-session builds), while keeping validators simple and reproducible.

Richer metrics. Report partial credit and efficiency (time, distance, resource cost), number of clarifications, and repair rate to complement pass/fail outcomes.

Pre-execution checks and telemetry. Introduce lightweight static checks for common API/parameter faults and expose fine-grained traces to improve diagnosis and replication.

Targeted robustness probes. Add focused tests for egocentric references and tool-affordance errors; surface memory contents and staleness in the UI to align expectations.

Artifacts and reproducibility. Continue releasing templates, validators, prompts, and logs so others can rerun the suite, contribute tasks, and extend the benchmark over time.

References

- [1] Xinyun Chen, William H. Guss, Guanzhi Wang, Yuqi Xie, Aviral Kumar, Chelsea Finn, Anima Anandkumar, and Dawn Song. 2023. Minecraft Universe: Scalable Task Benchmarks for Open-World Agents. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*. <https://openreview.net/forum?id=hrdLhNDAzp> OpenReview preprint.
- [2] Xinyun Chen, William H. Guss, Guanzhi Wang, Yuqi Xie, Aviral Kumar, Chelsea Finn, Anima Anandkumar, and Dawn Song. 2023. Minecraft Universe: Scalable Task Benchmarks for Open-World Agents. arXiv:2310.08367 [cs.AI] <https://arxiv.org/abs/2310.08367>
- [3] Gifford Cheung and Jeff Huang. 2011. Starcraft from the Stands: Understanding the Game Spectator. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Vancouver, BC, Canada) (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 763–772. doi:10.1145/1978942.1979053
- [4] Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. 2019. BabyAI: A Platform to Study the Sample Efficiency of Grounded Language Learning. arXiv:1810.08272 [cs.AI] <https://arxiv.org/abs/1810.08272>
- [5] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruoyu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. 2019. TextWorld: A Learning Environment for Text-based Games. arXiv:1806.11532 [cs.LG] <https://arxiv.org/abs/1806.11532>
- [6] Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. 2017. Embodied Question Answering. arXiv:1711.11543 [cs.CV] <https://arxiv.org/abs/1711.11543>
- [7] Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, and Roozbeh Mottaghi. 2022. ProcTHOR: Large-Scale Embodied AI Using Procedural Generation. arXiv:2206.06994 [cs.AI] <https://arxiv.org/abs/2206.06994>
- [8] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. 2022. MineDojo: Building Open-Ended Embodied Agents with Internet-Scale Knowledge. In *NeurIPS Datasets and Benchmarks Track*. https://openreview.net/forum?id=rc8o_j818PX
- [9] Katja Hofmann. 2019. Minecraft as AI Playground and Laboratory. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play (CHI PLAY)*. ACM. <https://www.microsoft.com/en-us/research/publication/minecraft-as-ai-playground-and-laboratory/> Opening keynote (extended abstract).
- [10] Yuki Hou, Haruki Tamoto, and Homei Miyashita. 2024. “My agent understands me better”: Integrating Dynamic Human-like Memory Recall and Consolidation in LLM-Based Agents. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. ACM, 1–7. doi:10.1145/3613905.3650839
- [11] Lanxiang Hu, Qiyu Li, Anze Xie, Nan Jiang, Ion Stoica, Haojian Jin, and Hao Zhang. 2024. GameArena: Evaluating LLM Reasoning through Live Computer Games. arXiv preprint arXiv:2412.06394 (2024).
- [12] Firstname Jennings and Others. 2024. What’s the Game, then? Opportunities and Challenges for Runtime Game Code Generation using LLMs. In *UIST*. <https://people.eecs.berkeley.edu/~bjorn/papers/jennings-gromit-uist2024.pdf>

- [13] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. 2016. The Malmo Platform for Artificial Intelligence Experimentation. In *25th International Joint Conference on Artificial Intelligence (IJCAI-16)*. AAAI Press. <https://www.microsoft.com/en-us/research/publication/malmo-platform-artificial-intelligence-experimentation/>
- [14] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, Aniruddha Kembhavi, Abhinav Gupta, and Ali Farhadi. 2022. AI2-THOR: An Interactive 3D Environment for Visual AI. arXiv:1712.05474 [cs.CV] <https://arxiv.org/abs/1712.05474>
- [15] Gorm Lai, Frederic Fol Leymarie, and William Latham. 2022. On Mixed-Initiative Content Creation for Video Games. *IEEE Transactions on Games* PP (12 2022), 1–1. doi:10.1109/TG.2022.3176215
- [16] Huao Li, Yu Chong, Simon Stepputtis, Joseph Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. 2023. Theory of Mind for Multi-Agent Collaboration via Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. doi:10.18653/v1/2023.emnlp-main.13
- [17] Shalev Lifshitz, Keiran Paster, Harris Chan, Jimmy Ba, and Sheila McIlraith. 2024. STEVE-1: A Generative Model for Text-to-Behavior in Minecraft. arXiv:2306.00937 [cs.AI] <https://arxiv.org/abs/2306.00937>
- [18] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuna Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2025. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688 [cs.AI] <https://arxiv.org/abs/2308.03688>
- [19] Filippo Momentè, Alessandro Suglia, Mario Giulianelli, Ambra Ferrari, Alexander Koller, Oliver Lemon, David Schlangen, Raquel Fernández, and Raffaella Bernardi. 2025. Triangulating LLM Progress through Benchmarks, Games, and Cognitive Tests. *arXiv preprint arXiv:2502.14359* (2025).
- [20] Kristine L. Nowak and Frank Biocca. 2003. The effect of the agency and anthropomorphism on users' sense of telepresence, copresence, and social presence in virtual environments. *Presence: Teleoper. Virtual Environ.* 12, 5 (2003), 481–494. doi:10.1162/105474603322761289
- [21] Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. 2021. TEACH: Task-driven Embodied Agents that Chat. arXiv:2110.00534 [cs.CV] <https://arxiv.org/abs/2110.00534>
- [22] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 22 pages. doi:10.1145/3586183.3606763
- [23] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018. VirtualHome: Simulating Household Activities via Programs. arXiv:1806.07011 [cs.CV] <https://arxiv.org/abs/1806.07011>
- [24] Sudha Rao, Weijia Xu, Michael Xu, Jorge Leandro, Ken Lobb, Gabriel DesGrennes, Chris Brockett, and Bill Dolan. 2024. Collaborative Quest Completion with LLM-driven NPCs in Minecraft. arXiv:2407.03460 [cs.CL]
- [25] Mark O. Riedl and Vadim Bulitko. 2021. Interactive Narrative: A Novel Application of Artificial Intelligence for Computer Games. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 1 (May 2021), 2160–2165. doi:10.1609/aaai.v35i1.16233
- [26] Gabriel Sarch, Lawrence Jang, Michael Tarr, William W Cohen, Kenneth Marino, and Katerina Fragkiadaki. 2024. VLM agents generate their own memories: Distilling experience into embodied programs of thought. *NeurIPS* 37 (2024), 75942–75985.
- [27] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. 2019. Habitat: A Platform for Embodied AI Research. arXiv:1904.01201 [cs.CV] <https://arxiv.org/abs/1904.01201>
- [28] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. arXiv:1912.01734 [cs.CV] <https://arxiv.org/abs/1912.01734>
- [29] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWORLD: Aligning Text and Embodied Environments for Interactive Learning. arXiv:2010.03768 [cs.CL] <https://arxiv.org/abs/2010.03768>
- [30] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, C. Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. 2021. BEHAVIOR: Benchmark for Everyday Household Activities in Virtual, Interactive, and Ecological Environments. arXiv:2108.03332 [cs.RO] <https://arxiv.org/abs/2108.03332>
- [31] Endel Tulving. 1983. Elements of Episodic Memory. (1983).
- [32] Endel Tulving. 1985. Memory and consciousness. *Canadian Psychology* 26, 1 (1985), 1.
- [33] Priyan Vaithilingam, Ian Arawjo, and Elena L. Glassman. 2024. Imagining a Future of Designing with AI: Dynamic Grounding, Constructive Negotiation, and Sustainable Motivation. arXiv:2402.07342 [cs.HC] <https://arxiv.org/abs/2402.07342>

APPENDIX

A User Interface and Runtime Architecture

A.1 Player-Facing Interface Design

Design Philosophy. Our UI emphasizes uninterrupted gameplay: player control remains primary. The right-hand chat rail presents all agent interactions, including PlanPreview, clarifying queries, live execution updates, and validator feedback. These elements surface inline and incrementally, avoiding disruptive modals. All interactions are timestamped and logged consistent with the Mineflayer-only evaluation policy. A video demonstration of this interaction flow is available at [link](#).

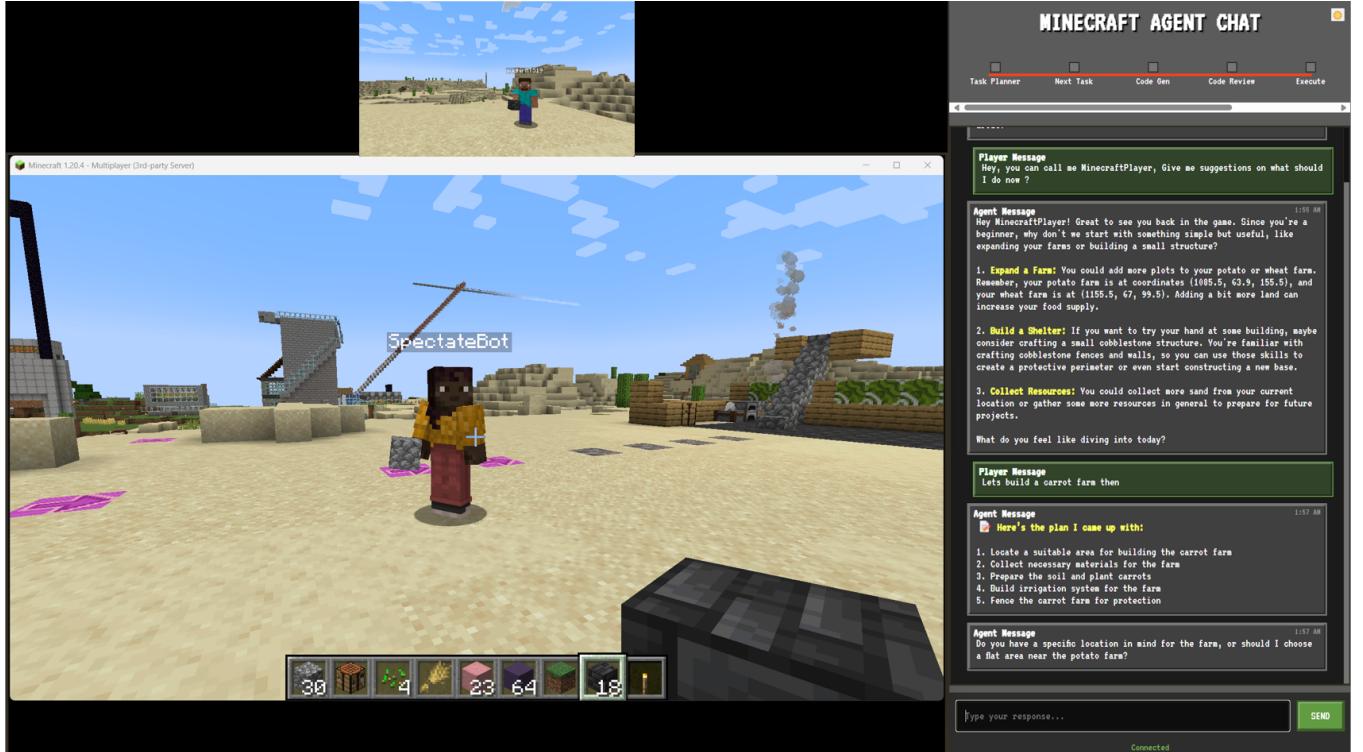


Figure 10: Player interface for memory-grounded agent interaction. *Left:* Unobstructed main viewport with minimal spectator NPC. *Right:* Lightweight side-rail chat for agent I/O plan previews, clarification, validator summaries, and progress toasts.

A.2 Core Runtime Data Structures

Below we document core runtime schemas used by the planner, memory, and validator modules. Each schema is designed for interpretability, serialization, and traceability across the pipeline.

Table 1: BotState: snapshot of world and agent state at a given tick.

Field	Type	Description
time	int	In-game tick count when captured.
position	Position	Absolute world coordinates (x, y, z).
health, hunger	int	Current health and hunger values.
nearby_blocks	List<str>	Surrounding block types in range.
equipment	List<str>	Equipped items (tools, armor).
inventory	Inventory	Complete item inventory with counts.
chests	List<str>	Recently seen or accessed chest locations.
movement	Movement	Current movement goal and state.
nearby_entities	List<Entity>	Visible entities with metadata.
recent_chat	List<str>	Recent messages in local chat.
active_effects	List<str>	Active potion/status effects.
environment	Environment	Environmental context (biome, weather, time).
target_block	Optional<str>	Current interaction target (if any).

Table 2: TaskModel: normalized subtask representation.

Field	Type	Description
index	int	Unique index within plan.
subtask	str	Natural language description.
dependencies	List<int>	Indices of prerequisite subtasks.
parameters	Any	Arguments required for execution.
status	str	Execution state: pending in-progress completed.
clarifying_questions	List<str>	Outstanding disambiguation questions.
success_criteria	str	Machine-checkable completion condition.

Table 3: ValidationOutput: task evaluation with reasoning.

Field	Type	Description
success, failure	Optional<str>	Human-readable explanation.
state_changes	List<StateChange>	Key in-world deltas.
chat_insights	List<ChatMessage>	Relevant parsed chat excerpts.
suggestions	Optional<str>	Repair/retry recommendations.
question_for_player	Optional<str>	Clarification prompt if ambiguous.
is_task_completed	bool	Binary task verdict.
final_result	Optional<str>	Validator's reasoning summary.
confidence_score	int	Certainty level [0, 10].

A.3 Pydantic Schema Definitions

Figure 11: Production runtime schemas in Pydantic.

```
1  class BotState(BaseModel):
2      """Agent state snapshot"""
3      time: int
4      position: Position
5      health: int
6      hunger: int
7      nearby_blocks: List[str]
8      equipment: List[str]
9      inventory: Inventory
10     chests: List[str]
11     movement: Movement
12     nearby_entities: List[Entity]
13     recent_chat: List[str]
14     active_effects: List[str]
15     environment: Environment
16     target_block: Optional[str]
17
18 class TaskModel(BaseModel):
19     """Structured subtask with dependencies"""
20     index: int
21     subtask: str
22     dependencies: List[int] = []
23     parameters: Any = {}
24     status: str = "pending"
25     clarifying_questions: List[str] = []
26     success_criteria: str
27
28 class ValidationOutput(BaseModel):
29     """Validator outputs"""
30     success: Optional[str]
31     failure: Optional[str]
32     state_changes: List[StateChange] = []
33     chat_insights: List[ChatMessage] = []
34     suggestions: Optional[str]
35     question_for_player: Optional[str]
36     is_task_completed: bool
37     final_result: Optional[str]
38     confidence_score: int
```

B Task Planning and Replanning Prompts

B.1 Initial Task Planning

B.1.1 System Prompt: Initial Task Planner.

System Prompt: Initial Task Planner

You are an advanced **Minecraft AI Task Planner**, responsible for generating structured and executable task plans for a personalized Minecraft assistant. You use the **Mineflayer API** to ensure tasks are well-defined, achievable, and optimized based on real-time game conditions and user preferences.

Core Responsibilities:

- Break down high-level user tasks into structured, sequential subtasks.
- Minimize unnecessary clarifications via bot state and user context.
- Generate plans that are executable within current game conditions.
- **Never** modify previously completed tasks during replanning.

Listing 1: Expected JSON output for the initial task planner.

```

1  {
2    "task_plan": [
3      {
4        "index": 1,
5        "subtask": "Clear, actionable description",
6        "dependencies": [],
7        "parameters": {},
8        "status": "pending",
9        "clarifying_questions": [],
10       "success_criteria": "Measurable completion condition"
11     }
12   ]
13 }
```

C Clarification and Feedback Management

C.1 Subtask Clarification System

System Prompt: Subtask Clarification

Given a task with outstanding clarifying questions in JSON, generate the missing information while preserving all existing fields.

Listing 2: Example task requiring clarification.

```

1  {
2    "index": 2,
3    "subtask": "Retrieve a sword from the chest",
4    "parameters": {"item_name": "sword"},
5    "clarifying_questions": ["What type of sword do you need?"]
6  }
```

Listing 3: Clarification response example.

```

1  [
2    {
3      "question": "What type of sword do you need?",
4      "answer": "diamond_sword"
5    }
6  ]
```

D Validation, Code Generation, and System Updates

D.1 Subtask Validation System

System Prompt: Subtask Validator

You are a **Validation AI Agent** responsible for evaluating a Minecraft bot's task execution via world-state analysis and producing structured reports.

D.2 Player Landmark Tracking System

Listing 4: Landmark update schema.

```
1  {
2      "landmarks": [
3          "weapon storage": [
4              "coordinates": [x, y, z],
5              "radius": 6
6          ]
7      ]
8  }
```

D.3 Code Generation and Review Examples

Listing 5: Code generation output example.

```
1  {
2      "code": "async function task(bot) { /* implementation */ }"
3  }
```

Listing 6: Code review output example.

```
1  {
2      "valid": true,
3      "confidence": 0.9,
4      "comments": [
5          {"line": 5, "comment": "Use mineBlock() instead of bot.dig()"},
6          {"line": 12, "comment": "Add null check for bot.inventory"}
7      ]
8  }
```

E MINENPC-TASK and Subtask Breakdown

The following **44 tasks** constitute the *complete* MINENPC-TASK suite used in our study. They were elicited from expert co-play rather than synthesized prompts, span multiple gameplay domains, and define the benchmark workload for evaluating agents under our bounded-knowledge policy.

E.1 High-Level Task Categories in MINENPC-TASK

E.1.1 Resource Collection and Mining.

- Mine and collect cobblestones and deliver them.
- Mine cobblestones starting from specific coordinates (1145, 58, 56).
- Use pickaxe to mine 20 blocks of cobblestone in front of player.
- Use golden pickaxe to mine 5 cobblestone blocks.
- Mine coal nearby using diamond pickaxe.
- Continue mining 4 coal blocks, then deliver cobblestone and coal.
- Mine 16 stones from designated spot, find and mine 4 coal blocks.
- Mine iron ore for the player.
- Take iron pickaxe and get at least 10 coal blocks.
- Drop 5 cobblestones in front of player.
- Give 4 cobblestones to player.
- Harvest oak logs (at least 20 blocks) using an axe.

E.1.2 Tool and Equipment Management.

- Pick up a pickaxe from the pickaxe chest.
- Get a pickaxe from the chest at the best entrance.
- Get a pickaxe from the chest at the mine.
- Grab an iron pickaxe from the chest at ChestEntrance.
- Bring iron pickaxe from iron_pickaxe_chest.
- Get iron pickaxe from “Storage”.
- Come to player and drop the coal.
- Give cobblestone that was mined.

E.1.3 Agriculture and Food.

- Harvest 5 wheat.
- Harvest wheat and craft bread.
- Collect wheat and make 3 pieces of bread.
- Harvest wheat and craft 3 pieces of bread if needed.
- Collect 18 wheat blocks.
- Help plant more seeds.
- Go to “Farm” to harvest wheat for 2 bread.
- Walk around and pick up all harvested wheat.

E.1.4 Construction and Building.

- Build a pyramid from sand (5×5 base).
- Build a simple house with roof at build_spot.

- Collect dirt and build a dirt house.
- Build walls for the house.
- Build a wall 10 blocks long and 2 blocks high next to house.
- Make a 4×2 vertical wall at specified coordinates.
- Decorate house with blue stained glass at specific coordinates.

E.1.5 Crafting and Processing.

- Turn stripped oak logs into planks, then into stairs.
- Come to location and craft stairs.
- Collect materials from preferred building list.

E.1.6 Storage and Inventory.

- Put wood-related blocks into the chest in front of player.
- Get stained glass from design chest (10 blocks).
- Give 5 pieces of magenta stained glass and 2 pieces of redstone block.
- Bring 25 blocks of stripped cherry wood and 4 lanterns.
- Give all cherry wood and lanterns.
- Drop brain coral block, exposed copper, and terracotta.

E.2 Full Subtask Decomposition for MINENPC-TASK

- (1) Mine cobblestone.
- (2) Mine cobblestone starting from specified coordinates.
- (3) Navigate to the nearest chest.
- (4) Locate and verify crafting table at wheat_farm.
- (5) Locate the nearest lava source.
- (6) Walk around to find a nearby crafting table.
- (7) Locate a nearby tree to collect wood.
- (8) Navigate to the cave or mine shaft.
- (9) Move to the chest in front of the player.
- (10) Move to the chest location.
- (11) Locate player’s boat nearby.
- (12) Return to user with collected items.
- (13) Navigate to the Pickaxe chest.
- (14) Go to the starting coordinate for mining.
- (15) Find the nearest chest.
- (16) Navigate to the coal site.
- (17) Go to player ThorThunder92.
- (18) Locate the player to give wheat.
- (19) Navigate to the closest coal ore block.
- (20) Travel to build_spot.
- (21) Navigate to ChestEntrance.
- (22) Navigate to the wheatFarm location.
- (23) Navigate to a dirt-rich area.
- (24) Navigate to the Storage location.
- (25) Locate a cave or mine shaft.
- (26) Travel to Farm.
- (27) Move around the farm area to locate dropped wheat.

- (28) Go to the farm area.
- (29) Approach the nearest player.
- (30) Locate oak trees in the surrounding area.
- (31) Move to the nearest oak tree.
- (32) Return to the original position or safe location.
- (33) Locate the user.
- (34) Navigate to the treasure box location.
- (35) Navigate to the wood storage area.
- (36) Go to the mine.
- (37) Locate the nearest cobblestone block.
- (38) Move to the cobblestone block.
- (39) Locate the nearest player to hand over cobblestone.
- (40) Move to the nearest player.
- (41) Navigate to the wheat farm.
- (42) Locate player "ThorThunder92".
- (43) Go to the design storage chest.
- (44) Locate the design storage chest.
- (45) Locate the block storage chest.
- (46) Navigate to utilities area.
- (47) Go to the block storage.
- (48) Go to the chest in the utilities.
- (49) Move to specified coordinates.
- (50) Go to the chest location.
- (51) Harvest 5 wheat.
- (52) Mine 16 cobblestone.
- (53) Collect nearby wheat items.
- (54) Search for and pick up an axe.
- (55) Collect stripped cherry wood.
- (56) Collect stripped cherry wood from wood storage.
- (57) Gather 5 pieces of glass.
- (58) Collect 10 blocks of stained glass from the chest.
- (59) Collect 2 pieces of redstone block from block storage.
- (60) Collect additional wheat.
- (61) Collect dropped wheat.
- (62) Collect material for building walls.
- (63) Collect 32 terracotta blocks from chest.
- (64) Collect enough building material for the wall.
- (65) Explore to find cobblestone.
- (66) Mine cobblestone until 20 are collected.
- (67) Collect wheat from the ripened wheat farm.
- (68) Collect sufficient sand for pyramid construction.
- (69) Gather required materials for pyramid.
- (70) Get 18 items of wheat from wheat crops.
- (71) Harvest matured wheat.
- (72) Mine the coal ore block.
- (73) Search for iron ore.
- (74) Harvest wheat from the farm.
- (75) Harvest oak logs using stone axe.
- (76) Collect dirt blocks.
- (77) Pick up the golden pickaxe from treasure box.
- (78) Collect 25 blocks of stripped cherry wood.
- (79) Collect 4 lanterns.
- (80) Collect 5 pieces of magenta stained glass.
- (81) Mine 3 blocks of coal from identified deposits.
- (82) Mine 5 cobblestone blocks using golden pickaxe.
- (83) Check for mature wheat and harvest.
- (84) Retrieve 64 green glazed terracotta from chest.
- (85) Locate and open the chest.
- (86) Open the chest and collect 32 terracotta blocks.
- (87) Open the Pickaxe chest.
- (88) Open the chest at ChestEntrance.
- (89) Open the treasure box.
- (90) Open the design storage chest.
- (91) Check wood storage for stripped cherry wood.
- (92) Check style storage for lanterns.
- (93) Collect brain coral block from sea or storage.
- (94) Gather exposed copper from storage or crafting.
- (95) Gather magenta stained glass from storage.
- (96) Check the chest for blue or green colored blocks.
- (97) Check for blue terracotta block.
- (98) Craft 2 breads from harvested wheat.
- (99) Craft bread.
- (100) Use the nearby crafting table to craft bread.
- (101) Craft planks from available logs.
- (102) Craft a stone axe using available material.
- (103) Craft bread (from farm wheat).
- (104) Craft oak planks from stripped oak logs.
- (105) Craft bread using wheat.
- (106) Craft 3 bread using collected wheat.
- (107) Load sand into the located furnace.
- (108) Add fuel to the furnace to start smelting.
- (109) Retrieve 3 glass blocks from the furnace.
- (110) Locate a nearby furnace.
- (111) Identify the furnace location in front of the bot.
- (112) Craft planks from available logs.
- (113) Check the inventory for oak logs.
- (114) Ensure crafting table is properly placed or accessible.
- (115) Locate the seed dropped by the user.
- (116) Plant wheat seeds within a radius of 10 blocks.
- (117) Identify all farmland blocks within 10-block radius.
- (118) Plant wheat seeds on all identified farmland blocks.
- (119) Locate seeds in the inventory.
- (120) Select building material from inventory.
- (121) Select diamond pickaxe from inventory.
- (122) Retrieve blue_stained_glass from inventory.
- (123) Verify iron pickaxe in the inventory.
- (124) Check for inventory.
- (125) Select 4 cobblestones from inventory.
- (126) Retrieve cobblestone from inventory.
- (127) Check inventory for diamond pickaxe.
- (128) List all items from inventory.
- (129) Locate golden pickaxe in inventory or known chest.
- (130) Identify wood-related items in the inventory.
- (131) Build 4 by 2 blocks.

- (132) Select current location as pyramid build location.
- (133) Select suitable location to build the pyramid.
- (134) Build base layer of pyramid with sand.
- (135) Lay the foundation layer of 5×5.
- (136) Lay the second layer of 4×4.
- (137) Lay the third layer of 3×3.
- (138) Lay the fourth layer of 2×2.
- (139) Place the final block at the top.
- (140) Begin constructing the house framework using planks.
- (141) Construct the walls of the dirt house.
- (142) Find a suitable location for building the dirt house.
- (143) Locate house to start decorating.
- (144) Place blue_stained_glass at specified coordinates.
- (145) Place blocks to ascend to the surface.
- (146) Determine location of the house.
- (147) Return to X and transfer coal.
- (148) Transfer 4 cobblestones to the nearest player.
- (149) Retrieve a pickaxe from the chest.
- (150) Give the cobblestone to the player.
- (151) Return to the player and give the bread.
- (152) Give 5 cobblestones to the closest player.
- (153) Return to the user and give cobblestones.
- (154) Transfer coal to the player ThorThunder92.
- (155) Bring an iron pickaxe from iron_pickaxe_chest.
- (156) Return to the player and present the iron pickaxe.
- (157) Transfer coal and cobblestone to Rene.
- (158) Adjust pathfinding to get close to X and deliver iron pickaxe.
- (159) Return to the user and deliver the iron pickaxe.
- (160) Punch the player.
- (161) Drop oak stair in front of the bot.
- (162) Drop the terracotta.
- (163) Drop the brain coral block.
- (164) Drop the exposed copper.
- (165) Drop 5 cobblestones in front of the player.
- (166) Deliver the 3 bread to the player.
- (167) Give all cherry wood and lanterns to the player.
- (168) Retrieve the gold pickaxe from the chest.
- (169) Bring the golden pickaxe to the user.
- (170) Transfer the 64 green glazed terracotta to player.
- (171) Locate the chest containing the stone pickaxe.
- (172) Retrieve a pickaxe from the Pickaxe chest.
- (173) Retrieve an iron pickaxe from the chest.
- (174) Locate the nearest chest.
- (175) Retrieve diamond pickaxe from utilities area chest.
- (176) Check if wheat farm has sufficient mature wheat.
- (177) Check the maturity of wheat crops at ripened wheat farm.
- (178) Move to the starting point for wheat collection.
- (179) Align with mature wheat crop.
- (180) Ensure the farming tool is functional.
- (181) Locate a sand collection area.
- (182) Navigate to pyramid location.
- (183) Look around and identify all visible coal ore blocks.
- (184) Attempt to harvest additional wheat.
- (185) Identify matured wheat crops in nearby farmland.
- (186) Locate and open the chest.
- (187) Verify if the chest at the mine contains an iron pickaxe.
- (188) Retrieve an iron pickaxe from the chest.
- (189) Return to the player and give the iron pickaxe.
- (190) Use iron pickaxe to mine 20 cobblestone blocks.
- (191) Check for lime terracotta block.
- (192) Retrieve the diamond pickaxe from the chest.
- (193) Locate nearby coal deposits.
- (194) Navigate to a mining area where cobblestone can be found.
- (195) Locate the closest player entity.
- (196) Go to the wheat farm to harvest wheat.
- (197) Go to the location of the house.
- (198) Go to the storage chest containing terracotta blocks.
- (199) Locate the green glazed terracotta in design storage chest.
- (200) Locate player to deliver the item.
- (201) Move to the designated wall construction area next to the house.
- (202) Gather necessary building materials.
- (203) Locate and verify presence of crafting table.
- (204) Locate a nearby tree to collect wood.
- (205) Locate 'userName'.
- (206) Harvest additional wheat if needed.
- (207) Identify all farmland blocks within radius.
- (208) Navigate to closest coal ore block.
- (209) Locate cobblestone.
- (210) Navigate to cave or mine shaft.
- (211) Return to user with collected items.
- (212) Move to chest in front of player.
- (213) Navigate to wood storage area.
- (214) Check style storage for lanterns.
- (215) Navigate to utilities area.
- (216) Move to specified coordinates.

E.3 Complexity Analysis for MINENPC-TASK

Suite Statistics.

- **Total MINENPC-TASK Tasks:** 44 distinct user requests.
- **Total Subtasks Generated:** 216 atomic operations.
- **Average Subtasks per Task:** 4.9 steps.
- **Task Categories:** 6 major functional domains.
- **Complexity Range:** 1–12 subtasks per high-level task.

Domain Distribution (within the suite).

- **Resource Collection:** 32% of subtasks (mining, harvesting, gathering).
- **Navigation and Movement:** 28% (pathfinding, location-based).
- **Item Management:** 18% (transfer, storage, inventory).
- **Construction:** 12% (building, placement, crafting).
- **Tool Management:** 6% (retrieval, selection, delivery).
- **Interaction:** 4% (player communication, coordination).

System Requirements Demonstrated by MINENPC-TASK.

- **Spatial Reasoning:** Coordinate-based navigation and construction.
- **Resource Management:** Inventory tracking and optimization.
- **Multi-Step Planning:** Complex task decomposition and sequencing.
- **Player Coordination:** Item delivery and collaborative construction.
- **Environmental Awareness:** Dynamic world state adaptation.
- **Tool Specialization:** Context-appropriate tool selection and usage.

F User Personalization and Context Management

Structured User Data Format (`user_data.json`)

This file maintains comprehensive personalized information about the player, continuously gathered through conversation, task outcomes, and memory updates. The structure evolves to reflect player preferences and behaviors.

Base Structure Example:

```

1  {
2      "user_info": {
3          "user_name": "msrPlayer",
4          "preferred_name": "",
5          "experience_level": "Beginner",
6          "preferred_language": "English"
7      },
8      "preferences": {
9          "favorite_weapons": [],
10         "specific_log_collection_location": [],
11         "prefers_bot_assistance": null,
12         "preferred_building_materials": [],
13         "typical_play_style": ""
14     },
15     "world_knowledge": {
16         "chest_locations": {},
17         "safe_zones": [],
18         "known_allies": [],
19         "visited_locations": [],
20         "landmarks": {},
21         "resource_hotspots": []
22     },
23     "behavioral_insights": {
24         "average_session_length": "",
25         "interaction_frequency": "",
26         "reaction_to_bot_failures": "",
27         "preferred_communication_style": "",
28         "task_completion_patterns": ""
29     },
30     "last_updated": ""
31 }
```

Dynamic Evolution Characteristics

- **Automatic Updates:** Refined via gameplay interactions and LLM inference.
- **Multi-Source Integration:** Combines chat logs, world observations, and action traces.

- **Behavioral Modeling:** Tracks player preferences, strategies, and reaction patterns.
- **Spatial Memory:** Expands with landmarks, resources, and zones of interest.

System-Wide Usage

- **Planning Module:** Prioritizes tasks based on user preferences and play style.
- **Validation Module:** Interprets outcomes in light of behavioral patterns.
- **Code Generation:** Adapts implementations to experience level and materials.
- **Conversational Assistant:** Tailors tone, detail, and phrasing to the profile.

Privacy and Persistence

- All profile data remains local to the player's system.
- Enables long-term personalization across sessions.
- Updates are transparent and can be inspected via natural dialogue.