Ghost in the Minecraft: Generally Capable Agents for Open-World Enviroments via Large Language Models with Text-based Knowledge and Memory

Xizhou Zhu[1,2*], Yuntao Chen[3*] , Hao Tian[2*] , Chenxin Tao[1,2*] , Weijie Su[2,4*] , Chenyu Yang[1*] , Gao Huang[1], Bin Li[4], Lewei Lu[2], Xiaogang Wang[2,5], Yu Qiao[6], Zhaoxiang Zhang[7], Jifeng Dai[1,6 ✉]

[1]Tsinghua University   [2]SenseTime Research
[3]Centre for Artificial Intelligence and Robotics, HKISI, CAS
[4]University of Science and Technology of China
[5]The Chinese University of Hong Kong   [6]Shanghai Artificial Intelligence Laboratory
[7]Institute of Automation, Chinese Academy of Science (CASIA)
{zhuxizhou,gaohuang,daijifeng}@tsinghua.edu.cn, chenyuntao08@gmail.com
tianhao2@senseauto.com, {tcx20,yangcy19}@mails.tsinghua.edu.cn,
jackroos@mail.ustc.edu.cn, binli@ustc.edu.cn, luotto@sensetime.com
xgwang@ee.cuhk.edu.hk, qiaoyu@pjlab.org.cn, zhaoxiang.zhang@ia.ac.cn

Abstract

The captivating realm of Minecraft has attracted substantial research interest in recent years, serving as a rich platform for developing intelligent agents capable of functioning in open-world environments. However, the current research landscape predominantly focuses on specific objectives, such as the popular "ObtainDiamond" task, and has not yet shown effective generalization to a broader spectrum of tasks. Furthermore, the current leading success rate for the "ObtainDiamond" task stands at around 20%, highlighting the limitations of Reinforcement Learning (RL) based controllers used in existing methods. To tackle these challenges, we introduce Ghost in the Minecraft (GITM), a novel framework integrates Large Language Models (LLMs) with text-based knowledge and memory, aiming to create Generally Capable Agents (GCAs) in Minecraft. These agents, equipped with the logic and common sense capabilities of LLMs, can skillfully navigate complex, sparse-reward environments with text-based interactions. We develop a set of structured actions and leverage LLMs to generate action plans for the agents to execute. The resulting LLM-based agent markedly surpasses previous methods, achieving a remarkable improvement of +47.5% in success rate on the "ObtainDiamond" task, demonstrating superior robustness compared to traditional RL-based controllers. Notably, our agent is the first to procure all items in the Minecraft Overworld technology tree, demonstrating its extensive capabilities. GITM does not need any GPU for training, but a single CPU node with 32 CPU cores is enough. This research shows the potential of LLMs in developing capable agents for handling long-horizon, complex tasks and adapting to uncertainties in open-world environments. See the project website at https://github.com/OpenGVLab/GITM.

1   Introduction

"What if a cyber brain could possibly generate its own ghost, create a soul all by itself? And if it did, just what would be the importance of being human then?"

— Ghost in the Shell (1995)

---

*Equal contribution. This work is done when Chenxin Tao and Weijie Su are interns at SenseTime Research.
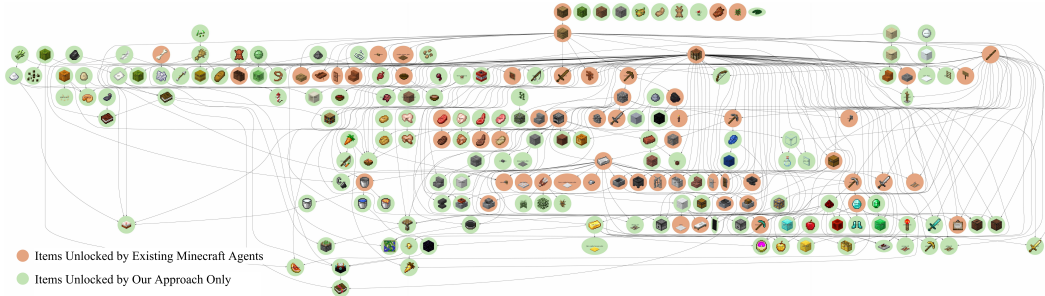✉ Corresponding to Jifeng Dai <daijifeng@tsinghua.edu.cn>.

Figure 1: **Our GITM unlocks the entire technology tree by obtaining all items in Minecraft Overworld.** Each node represents an individual item in Minecraft. The directed edges between nodes represent prerequisite relationships for obtaining items. For better readability, we manually merge some similar nodes, *e.g.*, "wooden_pickaxe", "wooden_axe", "wooden_hoe", and 'wooden_shovel' are merged into one node, and "wooden_pickaxe" is selected to represent the merged node. Existing Minecraft agents [2, 7, 25] only unlocked 78 / 262 = 30% items, while our GITM successfully unlocked all items.
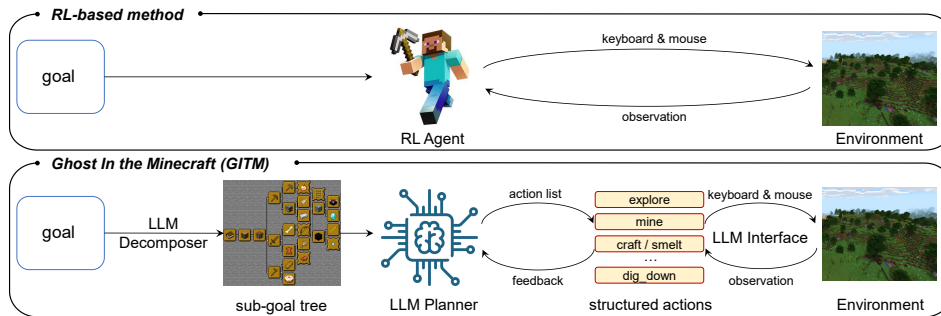

Figure 2: **Comparison between RL-based method and our GITM.** RL agents try to map an complex goal directly to a sequence of low-level control signals, while our GITM leverages LLM to break down the goals and map them to structured actions for final control signals.

Minecraft, as the world's best-selling game, boasts over 238 million copies sold and more than 140 million peak monthly active users [27]. Within the game, hundreds of millions of players have experienced a digital second life by surviving, exploring and creating, closely resembling the human world in many aspects. Given its massive scale, vast success, and unrestricted freedom, Minecraft has established itself as an unparalleled platform for researching autonomous and robust *Generally Capable Agents (GCAs)* [23] in open-world environments brimmed with long-horizon challenges, environmental disruptions, and uncertainties.

Minecraft acts as a microcosm of the real world. Developing an automated agent that can master all technical challenges in Minecraft is akin to creating an artificial intelligence capable of autonomously learning and mastering the entire real-world technology. However, existing researches [2, 7, 25] remain narrowly scoped. Prior studies have predominantly focused on the specific goal of `ObtainDiamond` [18]. Yet, in the process of obtaining diamonds, the number of types of items involved only accounts for <5% of the entire Minecraft world. `ObtainDiamond` only requires specialized skills in a specific domain, while obtaining all items in Minecraft demonstrates a wide range of knowledge and capabilities, similar to mastering multidisciplinary fields in the real world. As illustrated in Fig. 1, our work endeavors to obtain all items in Minecraft within a reasonable computation budget. This achievement stands as a significant milestone in the development of GCAs, illustrating the potential of intelligent agents to match human performance in terms of versatility and adaptability.

Although reinforcement learning (RL) [16] is the most popular paradigm for approaching GCAs, it has shown some staggering limitations in conquering Minecraft. RL-based agents typically require a vast number of learning steps (*e.g.*, nearly 30 million steps to obtain diamonds as reported in DreamerV3 [7]) and exhibit poor scalability when learning new tasks(*e.g.*, VPT [2] uses different agents for world exploration and diamond mining). As a consequence, adopting RL-based agents for

completing a wide range of tasks may require an prohibitively high number of training steps, making it impractical to obtain all items in Minecraft. This inefficiency and lack of adaptability have hindered the development of generally capable agents in open-world environments.

As shown in Fig. 2, the biggest dilemma of previous RL-based agents is how to map an extremely long-horizon and complex goal to a sequence of lowest-level keyboard/mouse operations. To address this challenge, we propose our framework Ghost In the Minecraft (GITM) [2], which uses Large Language Model (LLM)-based agents as a new paradigm. Instead of direct mapping like RL agents, our LLM-based agents employ a hierarchical approach. It first breaks down the decompose goal into sub-goals, then into structured actions, and finally into keyboard/mouse operations. Such decomposition is similar to how humans solve complex problems in the real world, enabling mastery of Minecraft with efficiency orders of magnitude higher than that of RL. LLM can also leverage text-based knowledge and memory to quickly acquire the ability to interact with the environment and accomplish goals, offering immense learning efficiency improvements, unlimited scalability and representing a disruptive innovation compared with RL. Our GITM framework has the potential to revolutionize the path to generally capable agents.

Specifically, the proposed LLM-based agent consists of an LLM Decomposer, an LLM Planner, and an LLM Interface, which are responsible for the decomposition of sub-goals, structured actions, and keyboard/mouse operations, respectively. Given a goal in Minecraft, LLM Decomposer first decomposes it into a series of well-defined sub-goals according to the text-based knowledge collected from the Internet. Then, LLM Planner plans a sequence of structured actions for each sub-goal. The structured actions are defined with clear semantics and corresponding feedback, enabling LLMs to understand surrounding environments and make decisions at the cognitive level. LLM Planner also records and summarizes successful action lists into a text-based memory to enhance future planning. Finally, LLM Interface execute the structured actions to interact with the environment by processing raw keyboard/mouse input and receiving raw observations.

In this paper, we demonstrate the feasibility of employing Large Language Models (LLMs) to develop Generally Capable Agents (GCAs) within an open-world environment built from Minecraft. By exploiting the common sense and reasoning capabilities of LLMs for hierarchical goal decomposition, as well as utilizing text-based knowledge and memory, this paradigm shows the possibility of enabling agents to address a wide range of challenges within Minecraft and allowing them to effectively handle such open-world environment. Consequently, our agent has surpassed all previous methods in achieving the `ObtainDiamond` goal (+47.5% success rate). Our agent also demonstrates superior learning efficiency compared to previous methods, reducing the number of environment interaction steps by more than $10,000\times$. Specifically, VPT [2] needs to be trained for 6,480 GPU days, DreamerV3 [7] needs to be trained for 17 GPU days, while our GITM does not require any GPUs and can be trained in just 2 days using a single CPU node with 32 CPU cores. More importantly, by obtaining all items in Minecraft Overworld as a milestone, this work represents a crucial first step towards achieving GCAs that can handle any task humans can accomplish in Minecraft.

2 Related Work

Minecraft agents are intelligent programs that can perform various tasks within Minecraft world. Reinforcement learning has dominated this area for many years. Some initial attempts have tried to use hierarchical RL [14, 15, 22] or imitation learning [1] in MineRL competitions [6, 10, 17]. Recently, with large-scale web data, VPT [2] builds a foundation model for Minecraft by learning from videos. Based on its success, many works [18] have also explored to finetune foundation model with human feedback. On the other hand, as Minecraft agents become increasingly proficient in handling simple tasks, the importance of multi-task learning becomes more prominent. Some previous works have adopted knowledge distillation [24] and curriculum learning [11], while recent works [3, 5] tried to construct a language-conditioned multi-task agent via feeding the goal description embedding into the model.

Recently, researchers have come to aware the extraordinary general planning ability for LLMs [8]. Many works [9, 25, 28] have leveraged LLMs for enhancing the high-level planning ability of minecraft agents. Inner Monologue [9] leveraged environment feedback to improve the planning ability of LLM. DEPS [25] further extended this closed-loop interaction by introducing description,
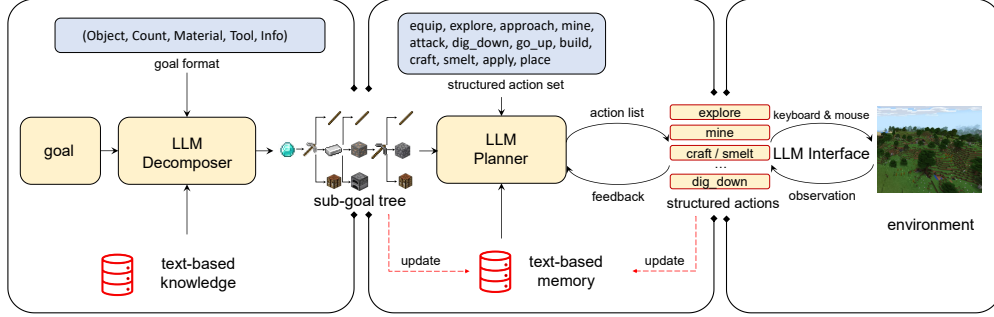

Figure 3: **Overview of our GITM.** Given a Minecraft goal, the LLM Decomposer divides the goal into a sub-goal tree. The LLM Planner then plans an action sequence for each sub-goal. Finally, the LLM Interface executes each action in the environment. Our LLM-based agents can be further enhanced by leveraging text-based knowledge and memory.

explainer and selector. Plan4MC [28] pre-defined basic skills and instructed LLM to extract the relationship between skills to construct a skill graph.

Unlike previous RL-based or RL with LLM methods, our LLM-native approach brings the minecraft agent to another level both in efficiency and robustness by leveraging high-level action abstraction and text-based knowledge and memory.

Large Language Models with Tools Extending the ability of LLMs by leveraging external tools have drawn a lot of attentions recently. Several works [4, 13, 21] have explored to augment LLMs with robot perception and control abilities. Code as Polices[13] tried to prompt LLM to generate codes that can drive robots. PaLM-E [4] unified robot perception, instruction following, task planning and low-level control into a unified framework. Another line of works tries to build external plugins around LLMs to enhance its ability. Toolformer [19] tries to teach LLMs to choose and use a wide range of tools like calculator and search engines and incorporate the results from tools into text generation. HuggingGPT [20] builds an agent for leveraging a combination of vision, language and audio models hosted on HuggingFace for completing user request. API Bank [12] proposes a syntheic benchmark suite for evaluating the how good LLMs are for using external tools.

Compared with these tool-augmented LLMs, our agents are tasks for much more complex goals in a high uncertain open-world.

3 Method

Traditional RL-based agents struggle to develop generally capable agents in Minecraft. The core issue is that they attempt to map extremely long-horizon and complex goals directly to the lowest-level keyboard and mouse operations. To overcome this, we propose LLM-based agents in Fig. 2 that utilize hierarchical goal decomposition. LLM Decomposer, LLM Planner, and LLM Interface are introduced to progressively decompose the task goal into sub-goals, structured actions, and keyboard/mouse operations. Moreover, LLM-based agents can leverage text-based knowledge and memory to quickly acquire the skills needed to master Minecraft.

3.1 LLM Decomposer

Rather than directly assigning the task goal to the agent and expecting a comprehensive and robust action plan, this work suggests the more practical strategy of decomposing the task goal into a series of more achievable sub-goals. By addressing each constituent sub-goal, the task goal can be progressively achieved. To this end, an LLM Decomposer is proposed. Goals are fed to the decomposer and recursively decomposed into a sub-goal tree. Text-base knowledge provides the necessary information for decomposition.

Goal Format. Since we aim to obtain all items in Minecraft, all goals can be defined in the format of

$$(\texttt{Object, Count, Material, Tool, Info}), \qquad (1)$$

where "Object" denotes the target item, "Count" specifies the target quantity. "Material" and "Tool" refer to prerequisites needed to obtain the target item. "Info" stores the text-based knowledge related to this goal. Given a specific goal, its sentence embedding extracted from a pre-trained LLM is used
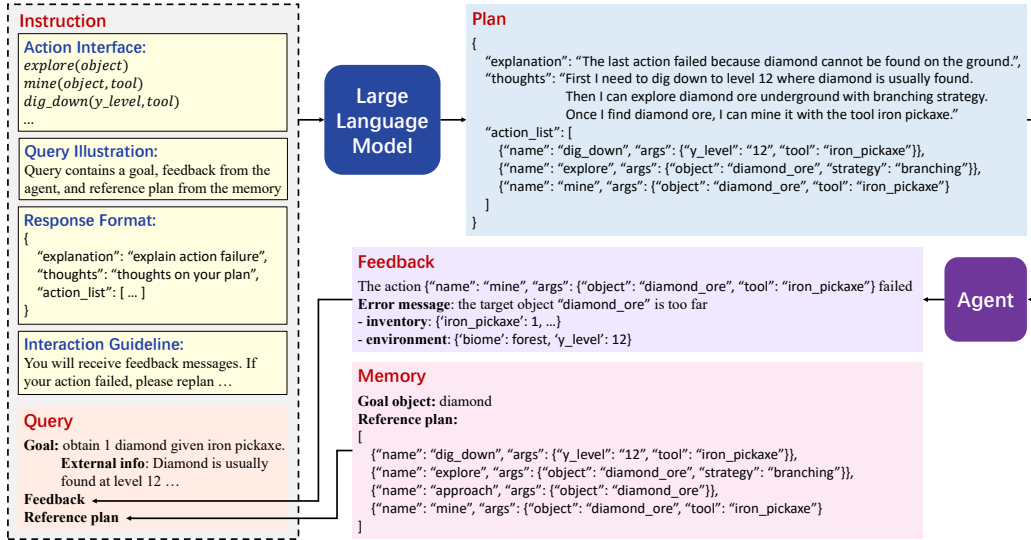
4

Figure 4: **Illustration of our planning process with the LLM Planner and the agent in the loop.** Given a specific goal, the planner generates plans with structured actions under the guidance of instruction, user query, previous feedback, and reference plan from memory. The agent executes the actions and provides feedback for the following planning.

to retrieve the most relevant text-based knowledge from an external knowledge base. Then, the LLM identifies the required material, tools, and related information from the gathered knowledge. The complete instructions for the LLM are described in Appendix.

Recursive Decomposition. This goal format enables recursive decomposition of each goal into a sub-goal tree. Specifically, given a goal, all prerequisite items are listed as sub-goals, including materials, tools, and their corresponding quantities. Then, the recursive decomposition continues for each sub-goal until it has no prerequisites. After the decomposition is completed, the execution sequence of the sub-goals is planned through post-order traversal. Such goal decomposition significantly enhances the success rate of LLM planning, especially for goals necessitating long-horizon planning.

Text-based Knowledge. External knowledge is essential for automatic goal decomposition. We build an external knowledge base documented in text from the Minecraft Wiki on the Internet [3] and the item crafting/smelting recipes, providing an exhaustive source of knowledge about the Minecraft world. For instance, if we need to craft a wooden pickaxe, the item crafting recipe will indicate that the required materials are three planks and two sticks, and the necessary tool is a crafting table. It also provides information about the distribution of raw materials. For example, diamonds are frequently found in levels 10~12 underground.

3.2 LLM Planner

LLMs excel at language understanding and reasoning but struggle with low-level control and multimodal perception. To leverage LLMs' strengths while addressing their limitations, we develop structured actions and feedback mechanisms as an abstract interface for them to manage agent-environment interaction. We propose an LLM-based Planner to achieve goals in Minecraft. Given a goal, it generates structured actions to control agents, receives feedback, and revises plans accordingly. It also has a text memory that aids planning by providing solutions for frequent goals.

Structured Actions. The structured actions are designed with well-defined functions and clear semantics, enabling LLMs to make decisions at the cognitive level. A structured action can be defined as follows:

$$(\text{Name}, \text{Arguments}, \text{Description}), \tag{2}$$

---

[3] https://minecraft-archive.fandom.com/wiki/Minecraft_Wiki

Table 1: **Examples of structured actions.** A structured action contains name and arguments for execution, as well as description to help LLMs understand and decide when to choose this action.

Name	Arguments	Description
equip	object	Equip the object from the inventory: used to equip equipment, including tools, weapons, and armor.
explore	object, strategy	Move around to find the object: used to find objects including block items and entities on the ground.
approach	object	Move close to a visible object: used to approach the object you want to attack or mine.
mine/attack	object, tool	Attack / Mine the object with the tool: used to attack / mine the object within reach.
dig_down/go_up	ylevel, tool	Dig down / Go up with the tool: used to go down / up underground.
build	blueprint	Build according to a blueprint: used to place corresponding objects on locations according to a preset blueprint.
craft/smelt	object, tool, material	Craft / Smelt the object with the materials and tool: used to craft new object that is not in the inventory or is not enough.
apply/place	object, tool	Apply / Place the tool on the object: used to apply tools or place blocks.

The name and arguments defines the action we want the agent to execute, while the action description provides enough information for letting LLMs know when to choose the corresponding actions, as shown in Tab. 1.

We extract the set of structured actions by leveraging the powerful reasoning capability of LLMs. Specifically, a pre-trained LLM is utilized to decompose the 3141 predefined tasks provided by MineDojo [5] into action sequences. Instructions for guiding LLMs on action decomposition are provided in Appendix. Then, we extract the structured actions by selecting frequent actions and merging actions with similar functionalities. See Appendix for the set of structured actions.

Feedback Mechanism. Open-loop planning cannot guarantee success, especially in open-world environments, where agents might encounter unexpected events. Feedback is crucial to form an effective closed loop. Without appropriate feedback, the LLM has no information about the consequences of actions and may repeat failed action plans. Feedback message is designed to present the agent's current state in the environment (*i.e.*, inventory and environment), as well as the success and failure information for each executed actions, as shown in Fig. 4. By incorporating this feedback message, the LLM can update its understanding of the environment, refine their strategies, and adapt their behavior accordingly.

Planning. Once the abstract interface is prepared, a pre-trained LLM is queried to generate goal-specific action sequence. This is achieved through carefully designed instructions and user queries, enabling the LLM to efficiently create and revise the plans. Fig. 4 illustrates the planning process. See Appendix for the full description.

`Instruction` specifies the guidelines that LLMs must follow when planning, including 1) *Action Interface* provides functional descriptions of the structured actions and their parameters; 2) *Query Illustration* clarifies the structure and meaning of user queries; 3) *Response Format* requires LLM to return responses in the format of {Explanation, Thought, Action List}, where "Explanation" requires LLMs to explain the reason for action failure, "Thought" requires LLM to use natural language to plan before outputting action sequences as a chain-of-thought (CoT) mechanism [26], and "Action List" outputs a list of structured actions to be executed; 4) *Interaction Guideline* guides LLMs to correct failed actions based on the feedback message, thus enabling the LLM to revise the plan.

`User Query` provides the specific query to LLMs for a given goal, including 1) *Goal* represents the objective by text as "Obtain `Count` `Item`, given `Material` and `Tool`. Extra info: `Info`" according to Eq. (1); 2) *Feedback* is the feedback information of the abstract interface; 3) *Reference Plan* provides a common reference plan for the current goal retrieved from the text-base memory.

Text-based Memory is designed for LLM to maintain common reference plans for each encountered objective as experiential knowledge. LLMs acquire the experience about controlling agents and resolving specific situations through game play and agent interaction. Instead of starting from scratch every time, using prior experience allows LLMs to handle tasks more efficiently, a process similar to human skill improvement through practice.

To this end, we design a text-based memory mechanism for LLM to store and retrieve gained knowledge. Unlike the RL-based model, which stores knowledge in parameters, this textual knowledge is explicit, logical, and closely aligned with human thought processes. This allows for direct application to a wide range of similar tasks, leading to more efficient learning and improved generalization.

Specifically, during each game episode, once the goal is achieved, the entirely executed action list would be stored in memory. The LLM may achieve the same goal under various circumstances, resulting in a range of different plans. To identify a common reference plan suitable for general situations, essential actions from multiple plans are summarized. This summarization process is
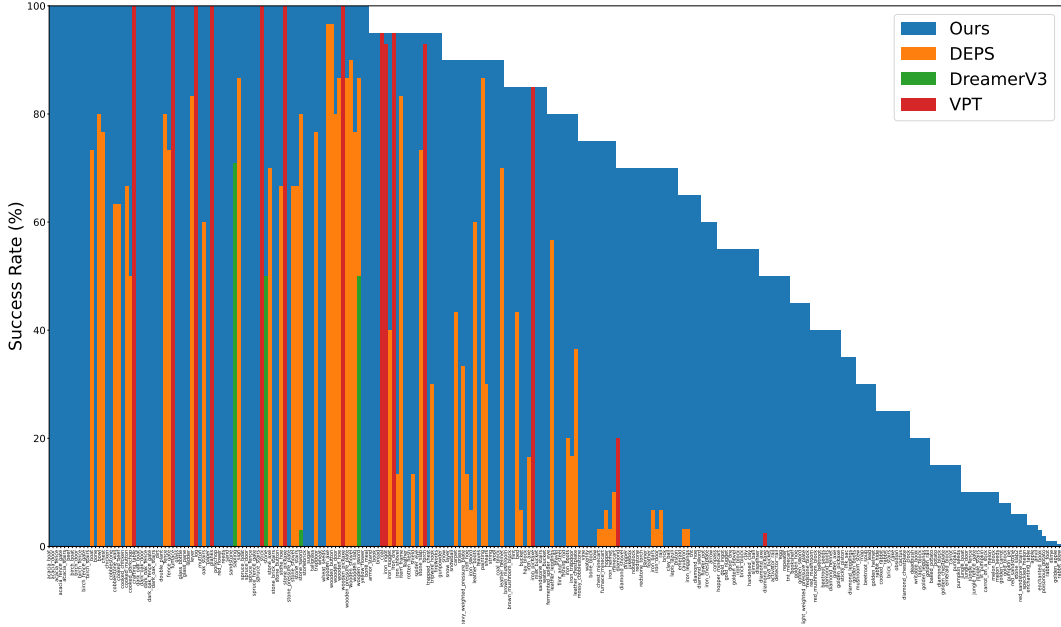
Figure 5: **Success rate for all items in the entire Minecraft Overworld Technology Tree**. The x axis lists all item names. We overlay the results from our GITM and the best results from baselines.

also implemented using LLMs (see Appendix for details). When encountering similar goals, the LLM creates new plans based on the summarized reference plans retrieved from memory. Successful action sequences from these new plans are also added to memory for future summarization. As the LLM-based Planner accumulates summaries, it becomes increasingly effective.

3.3 LLM Interface

Unlike the existing RL-based agents that directly control keyboard and mouse, LLM-based agents interact with the environment through structured actions and feedback messages. The LLM interface serves to implement structured actions as keyboard/mouse operations, and extract observations provided by the environment into feedback messages.

Structured actions can be implemented in various ways such as hand-written scripts or RL-learned models. While RL-learned models have been employed in Minecraft previously, they were either broad in functionality but inefficient in practice, or too specific in functionality, limiting their applicability to general tasks and actions. Clarifying the capability boundary of RL-learned models is challenging. Instead, in this work, we choose to implement structured actions using hand-written scripts. Since structured actions are well-defined and easy to implement, we can manually implement them based on observations (*e.g.*, location, LiDAR, and voxel) and basic operations (*e.g.*, move, jump, adjust camera angle, click left mouse button, and click right mouse button) provided by the MineDojo [5] environment. See Appendix for details.

Feedback messages can be obtained directly from the environment. These include whether the structured action execution succeeded or failed. If the execution fails, the reason for the failure is additionally notified. It also includes the current state of the agent in the environment, including the items in the inventory, the current biome and depth, etc. See Appendix for details.

4 Experiments

Task Definition and Metrics. We measure the ability of GITM through item collection tasks. We only collect items could be found in the Overworld. We exclude items could only be obtained by trading with villagers, opening treasure chest or find a special structure on the map, using a tool enchanted with Silk Touch. This give us a total of 262 tasks. For the assessment of our agent, we employ "Coverage of the Overworld Technology Tree" and "Success Rate" as evaluation metrics.

4.1 Main Result

Unlocking the Entire Technology Tree by Obtaining All Items. Compared with existing Minecraft agents [2, 7, 25] which mostly focuses on solving the `ObtainDiamond` task and could only unlock a limited part of the full technology tree (13/262 for Dreamerv3, 15/262 VPT, 69/262 for DEPS), our approach could collect all 262 items as shown in Fig. 1. There are two major blockers for existing methods. For RL-based methods like VPT [2] and DreamerV3 [7], the goal item(diamond) is hard-coded into the model weights, which means there are no easy way to re-task the trained RL agents for collecting other items in the inference stage. Moreover, the low training efficiency hinders them from solving extremely long-horizon tasks (e.g., obtaining a "enchanted_book"). For methods like DEPS [25] that use an RL controller [3] and LLM planner still rely on pre-trained RL agents to execute specific subtasks (e.g. mining 1 "cobblestone") in the generated plan. So these approaches still suffer from the inability of RL-based methods alone to generalize to unseen tasks (e.g. obtaining "lapis_lazuli"). In contrast, we extract a well-defined set of structured actions by using LLMs to decompose over 3000 predefined MineDojo tasks. This provides broad, open-world Minecraft capability. Combined with LLM planning, it enables solving more complex tasks than `ObtainDiamond` - which RL cannot achieve. Our knowledge bases also improve efficiency. To our knowledge, we present the first agent to unlock the entire Overworld technology tree - a level of open-world skill RL-based methods have not demonstrated.

Success Rate for the Entire Technology Tree. We show the success rate of our method for collecting all Overworld items in Fig. 5. Our methods could achieve 100% success rate for simple tasks like collecting wooden tools. It achieves non-zero success rates for all items which indicates a strong collecting capability. The successful rate for collecting different items change smoothly for our agent, which showcase the robustness of our method against the highly uncertain open world environment.

4.2 Comparison with Other Minecraft Agents

Table 2: **Comparison of our GITM with previous methods on `ObtainDiamond` challenge.**

Method	Success Rate (%)				
DreamerV3	-	50.0	3.0	0.01	0.01
DEPS	90.0	80.0	73.3	10.0	0.6
VPT	100.0	100.0	100.0	85.0	20.0
Our GITM	**100.0**	**100.0**	**100.0**	**95.0**	**67.5**
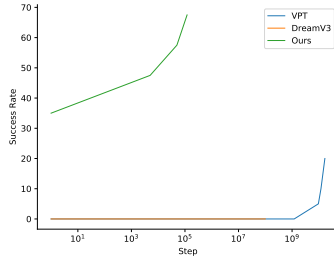
Figure 6: **Comparison of learning efficiency.**

We compared our LLM-based method with three existing agents: VPT [2], DreamerV3 [7], and DEPS [25] on the well known `ObtainDiamond` challenge, i.e, obtaining a diamond from scratch in Minecraft. Previous methods set different time limits of a single episode of game play (20 minutes for VPT, 30 minutes for Dreamerv3, and 10 minutes for DEPS). For fair comparison, we use the strictest limit of previous methods: 10 minutes (12,000 steps at 20Hz control).

Success Rate for Obtaining Diamond and Other Items. Since VPT and Dreamerv3 are not targeted for collecting items other than diamond, we mainly compare our method with DEPS for items not related to obtain diamonds. Overall, our GITM and VPT rank task difficulty similarly, but DEPS rankings severely fluctuate for tasks more complex than mining coal. Dreamerv3 also behaves oddly by having an abnormally low success rate on tasks like obtaining a stone sword. As shown in Fig. 2, most agents performs generally well for easy tasks relating to make wooden tools. VPT could even rival with our GITM for the success rate of obtaining iron axes. But for obtaining diamonds, our method wins over any other methods by 3.5 times on the succeess rate.

This giant improvement comes from the following two aspects: First, we employ the strong long-term planning capability of LLMs to decompose the complex tasks into feasible sub-goals and tackle them step by step. Second, our model can directly leverage external knowledge such as the suitable locations for mining ores, while RL models need to explore themselves and may not acquire reliable knowledge.

Table 3: **Ablation study.** The milestone items from left to right are crafting table 🪵, wooden pickaxe 🔨, stone pickaxe 🔨, iron pickaxe 🔨, and diamond 💎. The success rate is calculated under time limit of 12000 steps (total) and query limit of 30 (each sub-goal). "Goal Decomp." and "External Info." indicates goal decomposition and external knowledge respectively.

Goal Decomp.	Feedback	External Info.	Memory	Success Rate (%)				
				🪵	🔨	🔨	🔨	💎
				57.5	32.5	5.0	0.0	0.0
✓				90.0	90.0	67.5	2.5	0.0
✓	✓			97.5	95.0	77.5	20.0	5.0
✓	✓	✓		100.0	100.0	100.0	57.5	35.0
✓	✓	✓	✓	100.0	100.0	100.0	95.0	67.5

Learning Efficiency. Besides measuring the success rate of each agents, we also compare the learning efficiency of our model with other learnable models. Since DEPS uses a LLM-based planner without learning mechanism and a pre-trained RL-based controller, its performance could not improve with more episodes played and is excluded from the comparison here.

It usually takes tens of millions of steps to train an RL agent by updating parameters before its success rate starts to converges to meaningful non-zero numbers. However, the success rate for RL-based agents increases rather slowly even after them starts to converge. On the contrary, the learning process of our LLM-based agent is considerably faster. As shown in Fig. 6, our method requires several orders less episodes than any other methods before doubling its initial success rate. Moreover, our method is extremely sample efficient as our success rate raises from 35% to 47.5% by learning from the first five thousand steps. By just playing each task several times and summarize successful experience into the memory, the LLM-based agent can acquire explicit experiential knowledge and achieve significantly higher success rate.

4.3 Ablation Study

We conduct ablation experiments on the `ObtainDiamond` task. We set a time limit of 10 minutes of game play (12000 steps at the control frequency of 20Hz). When leveraging goal decomposition, for each sub-goal, we set the maximum number of queries to LLM as 30, and exceeding the query limit will be judged as a failure. For each setting, we run 40 games and calculate the success rate. Tab. 3 records the success rates of achieving the final goal diamond as well as the milestones in this goal, including crafting table, wooden pickaxe, stone pickaxe, and iron pickaxe.

Goal Decomposition. Without goal decomposition, the planner can only accomplish several short-term tasks such as obtaining stone axes with rather low success rate of 5%, which indicates the necessity of goal decomposition. Leveraging the powerful long-term planning capabilities of LLMs, the goals are decomposed into sub-goals feasible and practical for the planner, so the success rate for obtaining stone axes advances from 5% to 67.5% by leveraging goal decomposition alone.

Feedback Message. Feedback contains the agent's state and the execution result of the actions, which helps the planner to understand and make another attempt to correct the mistakes in the previous and deal with special cases. This enables the planner to accomplish a broader range of goals with higher success rate. As shown in the 3rd row of Tab. 3, our agent gain the ability to collect diamond by combining feedback with goal decomposition.

External Knowledge Base. External knowledge contains general rules, crafting recipes, and common tricks in Minecraft, such as the recipes for crafting iron ingot and iron pickaxe, the suitable location to find diamond ore, and the efficient way to get cobblestone. Providing the planner with this information greatly boosts the success rate of obtaining iron pickaxe and diamond, and the success rate of mining diamond increase by 7 times by learning from the knowledge base that diamonds are more likely to appear in specific levels.

Text-based Memory. Leveraging the reference plan recorded in the memory, the planner can handle the task it has encountered more efficiently. The success rates of obtaining iron pickaxe and diamond are 95.0% and 67.5%, surpassing the model without memory by 37.5% and 32.5%, respectively.

5   Conclusion

We introduce the GITM framework, which utilizes Large Language Models (LLMs) for hierarchical decomposition of goals. GITM introduces LLM Decomposer, LLM Planner and LLM Interface to gradually decompose goals into sub-goals, structured actions and keyboard/mouse operations. This work makes significant progress towards the `ObtainDiamond` goal, outperforming all previous methods by a significant margin (+47.5% success rate). This proves the potential inefficiency and poorly scalability of Reinforcement Learning (RL) in Minecraft, breaking the traditional reliance on RL. Moreover, by obtaining all items in Minecraft Overworld, this research marks a critical step toward Generally Capable Agents (GCAs) that match human performance in Minecraft.

References

[1] A. Amiranashvili, N. Dorka, W. Burgard, V. Koltun, and T. Brox. Scaling imitation learning in minecraft. *arXiv preprint arXiv:2007.02701*, 2020.

[2] B. Baker, I. Akkaya, P. Zhokov, J. Huizinga, J. Tang, A. Ecoffet, B. Houghton, R. Sampedro, and J. Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

[3] S. Cai, Z. Wang, X. Ma, A. Liu, and Y. Liang. Open-world multi-task control through goal-aware representation learning and adaptive horizon prediction. *arXiv preprint arXiv:2301.10034*, 2023.

[4] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[5] L. Fan, G. Wang, Y. Jiang, A. Mandlekar, Y. Yang, H. Zhu, A. Tang, D.-A. Huang, Y. Zhu, and A. Anand-kumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *arXiv preprint arXiv:2206.08853*, 2022.

[6] W. H. Guss, S. Milani, N. Topin, B. Houghton, S. Mohanty, A. Melnik, A. Harter, B. Buschmaas, B. Jaster, C. Berganski, et al. Towards robust and domain agnostic reinforcement learning competitions: Minerl 2020. In *NeurIPS 2020 Competition and Demonstration Track*, pages 233–252. PMLR, 2021.

[7] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[8] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[9] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[10] A. Kanervisto, S. Milani, K. Ramanauskas, N. Topin, Z. Lin, J. Li, J. Shi, D. Ye, Q. Fu, W. Yang, et al. Minerl diamond 2021 competition: Overview, results, and lessons learned. *NeurIPS 2021 Competitions and Demonstrations Track*, pages 13–28, 2022.

[11] I. Kanitscheider, J. Huizinga, D. Farhi, W. H. Guss, B. Houghton, R. Sampedro, P. Zhokhov, B. Baker, A. Ecoffet, J. Tang, et al. Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft. *arXiv preprint arXiv:2106.14876*, 2021.

[12] M. Li, F. Song, B. Yu, H. Yu, Z. Li, F. Huang, and Y. Li. Api-bank: A benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.

[13] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[14] Z. Lin, J. Li, J. Shi, D. Ye, Q. Fu, and W. Yang. Juewu-mc: Playing minecraft with sample-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:2112.04907*, 2021.

[15] H. Mao, C. Wang, X. Hao, Y. Mao, Y. Lu, C. Wu, J. Hao, D. Li, and P. Tang. Seihai: A sample-efficient hierarchical ai for the minerl competition. In *Distributed Artificial Intelligence: Third International Conference, DAI 2021, Shanghai, China, December 17–18, 2021, Proceedings 3*, pages 38–51. Springer, 2022.

[16] Y. Matsuo, Y. LeCun, M. Sahani, D. Precup, D. Silver, M. Sugiyama, E. Uchibe, and J. Morimoto. Deep learning, reinforcement learning, and world models. *Neural Networks*, 2022.

[17] S. Milani, N. Topin, B. Houghton, W. H. Guss, S. P. Mohanty, K. Nakata, O. Vinyals, and N. S. Kuno. Retrospective analysis of the 2019 minerl competition on sample efficient reinforcement learning. In *NeurIPS 2019 Competition and Demonstration Track*, pages 203–214. PMLR, 2020.

[18] S. Milani, A. Kanervisto, K. Ramanauskas, S. Schulhoff, B. Houghton, S. Mohanty, B. Galbraith, K. Chen, Y. Song, T. Zhou, et al. Towards solving fuzzy tasks with human feedback: A retrospective of the minerl basalt 2022 competition. *arXiv preprint arXiv:2303.13512*, 2023.

[19] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[20] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[21] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[22] A. Skrynnik, A. Staroverov, E. Aitygulov, K. Aksenov, V. Davydov, and A. I. Panov. Hierarchical deep q-network from imperfect demonstrations in minecraft. *Cognitive Systems Research*, 65:74–78, 2021.

[23] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

[24] C. Tessler, S. Givony, T. Zahavy, D. Mankowitz, and S. Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI conference on artificial intelligence*, 2017.

[25] Z. Wang, S. Cai, A. Liu, X. Ma, and Y. Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023.

[26] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[27] Wikipedia contributors. Minecraft — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Minecraft&oldid=1155148900, 2023.

[28] H. Yuan, C. Zhang, H. Wang, F. Xie, P. Cai, H. Dong, and Z. Lu. Plan4mc: Skill reinforcement learning and planning for open-world minecraft tasks. *arXiv preprint arXiv:2303.16563*, 2023.

A Implementation Details

A.1 LLM Decomposer

We use `gpt-3.5-turbo` from OpenAI API [4] for goal decomposition. The prompt is shown as follows, which consists of two parts: instruction with the role of "SYSTEM" and query with the role of "USER". The {object quantity}, {object name} and {related knowledge} are injectable slots that will be replace with corresponding texts before fed into the LLM.

---

[4] https://platform.openai.com/docs/api-reference

The recursive decomposition generates a sub-goal tree starting from the final goal object as the root node: if a goal has some prerequisites (materials or tools), for each required material or tool, we add a child node representing the goal of obtaining that material or tool, and then recursively decompose the child node, until there is no more prerequisites. The related knowledge is from: 1) Crafting/smelting recipes in MineDojo [5], written in the form "Crafting {quantity} {object} requires {material} as the material and {tool} as the tool"; 2) Wiki on the Internet [5]. We extract the paragraphs with keywords "obtaining", "mining", "sources", etc.

---

[5] https://minecraft-archive.fandom.com/wiki/Minecraft_Wiki

A.2   LLM Interface

Instruction for Extracting Structured Actions. To extract structured actions, we first ask LLM to generate a tree-structured action planning for each of the 3141 predefined tasks provided by MineDojo, and then converts each action step into a (`verb`, `object`, `tool`, `material`) tuple. During decomposition, it is essential to ensure actions are neither too broad nor too specific. We adjusted the depth of the action decomposition tree to achieve balance, and empirically set the depth as 2 to meet our requirements.

Specifically, we use `gpt-3.5-turbo` from OpenAI API to generate the structured actions. We add the following instruction to the content of "SYSTEM" role to generate the tree-structured plan. We add the goal description, *e.g.*, "find material and craft a iron pickaxe", to the content of "USER" role and then asks LLM to response according to the requirements.

SYSTEM:
You serve as an assistant that helps me play Minecraft.

I will give you my goal in the game, please break it down as a tree-structure plan to achieve this goal.

The requirements of the tree-structure plan are:

1. The plan tree should be exactly of depth 2.
2. Describe each step in one line.
3. You should index the two levels like '1.', '1.1.', '1.2.', '2.', '2.1.', etc.
4. The sub-goals at the bottom level should be basic actions so that I can easily execute them in the game.

USER:
The goal is to {`goal description`}. Generate the plan according to the requirements.

After that, we extract the action tuple from each sentence of the leaf nodes. We use the following instruction as the content of "SYSTEM" role to extract the tuple, and add the sentence to the content of "USER" role.

SYSTEM:
You serve as an assistant that helps me play Minecraft.

I will give you a sentence. Please convert this sentence into one or several actions according to the following instructions.

Each action should be a tuple of four items, written in the form ('verb', 'object', 'tools', 'materials')

'verb' is the verb of this action.
'object' refers to the target object of the action.
'tools' specifies the tools required for the action.
'material' specifies the materials required for the action.
If some of the items are not required, set them to be 'None'.

USER:
The sentence is {`sentence`}. Generate the action tuple according to the requirements.

Then, we extract the structured actions by selecting frequent actions and merging actions with similar functionalities. The set of structured actions is {`equip, explore, approach, mine/attack, dig_down, go_up, build, craft/smelt, apply`}. Note that we disregard more detailed action decomposition for `attack` and `build` to remove overly detailed short-term actions and focus on long-term task completion.

Action Implementation. The observation of the action contains LiDAR rays with an interval of 5 degrees in the horizon and vertical direction for locating objects, and voxels with 10 units radius only for navigation, inventory, life status, and agent location status (X-ray cheating is carefully avoided). RGB is not used in our implementation, although it provides more information than LiDAR rays.

For example, the biome, and category of the dropping item can not be identified by LiDAR rays. Some objects may also be missed by LiDAR due to sparseness of LiDAR rays. We also set the breaking speed to 100 and strength to 100, mainly following [7]. The detailed implementation of each structured action is as follows:

- `equip`: The equip action calls the environment API to equip the required object. The action succeeds when the API returns success. The action fails when the object is not in inventory or the equip API returns failure.

- `explore`: The explore action traverses the world until object is visible. This action regards the world as a chessboard, and each node on the chessboard is the center point of a 20×20 units area. Two strategies are implemented depending on whether the agent is on the ground or not. When the agent is on the ground, the BFS explore will be adopted. When the agent is under the ground, mainly for exploring ore, the DFS explore will be adopted. In the DFS exploration, the agent will break the blocks to form a mine road with width of 1 and height of 2. The action succeeds when the object is visible. The action fails when the explore exceeds a preset steps of 10,000 but the required object is not found.

- `approach`: The approach action finds the nearest visible required object and walks towards the object. We adopt A^* algorithm for finding path. The A^* algorithm can jump, translate and fall in four directions of north, south, east and west. We also allow the agent to jump while placing a block under the agent for ascent. If the object is out of the voxel observation range, A^* algorithm is iteratively applied to find the location nearest to the object. The action succeeds when the ℓ^∞ norm distance between the object and agent is less than 2. The action fails when there is no required object visible or no path can be found to walk close to the object.

- `mine/attack`: The mine/attack action uses the keyboard attack API with the tools to attack the object. Only visible object could be mined or attacked. The object of mine should be blocks, and the agent will continue mining the block until it is broken. The object of attack should be entities, and the agent will iteratively approach and attack the entity until it is killed. After the block is broken or the entity is killed, if there are items dropped by them, the agent will approach the items to collect them. The action succeeds when the block is broken or the entity is killed. The action fails when there is no visible object, no required tools is in inventory, or the visible object is out of attack range.

- `dig_down`: The dig_down action iteratively breaks the block underfoot with the tool until the required ylevel is reached. If the agent is on the ground, before digging down, current location is stored for going up action. After the action succeeds, the state of the agent is set to under ground. The action succeeds when the required ylevel is reached. The action fails when it exceeds the reset max steps 10,000 or no required tool is in inventory.

- `go_up`: The agent will first go back to the location stored by dig_down. Then, the go_up action puts dirt blocks underfoot to raise the agent. After the action is finished, the state of agent is set to on the ground. The action succeeds when the pre-stored location is reached. The action fails when the walk fails, exceeds the reset max steps 10,000 or there is no required tool in inventory.

- `build`: The build action places the required blocks according to a given blueprint from bottom to up. The action succeeds when all blocks have been placed. The action fails when there are no enough materials in inventory or it is invalid to place some blocks.

- `craft/smelt`: The action calls the environment API to craft/smelt the required object. The action succeeds when the required object is obtained. The actions fails when there are no enough materials in inventory or the agent is unable to place the crafting table/furnace or the API fails.

- `apply`: The apply action calls the keyboard use API, and applies the specific tool to the object, *e.g.*, applying the bucket on water to obtain water bucket. The action succeeds when the API returns success. The action fails when there is no visible object, no tool in inventory or the API fails.

Feedback Message. After the execution of each action, we will get feedback from the structured actions. The feedback will refresh the agent's state in Sec. A.3.2, including current inventory, biome, ylevel and on/under the ground status. The feedback will also contain the success/fail message from these action, as well as the inventory change during the action.

A.3 LLM Planner

Here we present the prompt for planning with LLM. We also use `gpt-3.5-turbo` from OpenAI API as the LLM planner. The model accepts inputs in form of a chat, i.e., the prompt is a dialogue consisting of several messages, each of which contains a role and the content. We set the `Instruction` with the role "SYSTEM" at the beginning, and use the `User Query` with the role "USER" to query the LLM for response. The content of the `Instruction` and `User Query` are as follows.

A.3.1 Instruction

SYSTEM:

You serve as an assistant that helps me play the game Minecraft.

I will give you a goal in the game. Please think of a plan to achieve the goal, and then write a sequence of actions to realize the plan. The requirements and instructions are as follows:

1. You can only use the following functions. Don't make plans purely based on your experience, think about how to use these functions.

`explore(object, strategy)`
Move around to find the object with the strategy: used to find objects including block items and entities. This action is finished once the object is visible (maybe at the distance).
Augments:
- object: a string, the object to explore.
- strategy: a string, the strategy for exploration.

`approach(object)`
Move close to a visible object: used to approach the object you want to attack or mine. It may fail if the target object is not accessible.
Augments:
- object: a string, the object to approach.

`craft(object, materials, tool)`
Craft the object with the materials and tool: used for crafting new object that is not in the inventory or is not enough. The required materials must be in the inventory and will be consumed, and the newly crafted objects will be added to the inventory. The tools like the crafting table and furnace should be in the inventory and this action will directly use them. Don't try to place or approach the crafting table or furnace, you will get failed since this action does not support using tools placed on the ground. You don't need to collect the items after crafting. If the quantity you require is more than a unit, this action will craft the objects one unit by one unit. If the materials run out halfway through, this action will stop, and you will only get part of the objects you want that have been crafted.
Augments:
- object: a dict, whose key is the name of the object and value is the object quantity.
- materials: a dict, whose keys are the names of the materials and values are the quantities.
- tool: a string, the tool used for crafting. Set to null if no tool is required.

`mine(object, tool)`
Mine the object with the tool: can only mine the object within reach, cannot mine object from a distance. If there are enough objects within reach, this action will mine as many as you specify. The obtained objects will be added to the inventory.
Augments:
- object: a string, the object to mine.
- tool: a string, the tool used for mining. Set to null if no tool is required.

`attack(object, tool)`
Attack the object with the tool: used to attack the object within reach. This action will keep track of and attack the object until it is killed.
Augments:
- object: a string, the object to attack.
- tool: a string, the tool used for mining. Set to null if no tool is required.

```
equip(object)
```
Equip the object from the inventory: used to equip equipment, including tools, weapons, and armor. The object must be in the inventory and belong to the items for equipping.
Augments:
- object: a string, the object to equip.

```
digdown(object, tool)
```
Dig down to the y-level with the tool: the only action you can take if you want to go underground for mining some ore.
Augments:
- object: an int, the y-level (absolute y coordinate) to dig to.
- tool: a string, the tool used for digging. Set to null if no tool is required.

```
go_back_to_ground(tool)
```
Go back to the ground from underground: the only action you can take for going back to the ground if you are underground.
Augments:
- tool: a string, the tool used for digging. Set to null if no tool is required.

```
apply(object, tool)
```
Apply the tool on the object: used for fetching water, milk, lava with the tool bucket, pooling water or lava to the object with the tool water bucket or lava bucket, shearing sheep with the tool shears, blocking attacks with the tool shield.
Augments:
- object: a string, the object to apply to.
- tool: a string, the tool used to apply.

2. You cannot define any new function. Note that the "Generated structures" world creation option is turned off.

3. There is an inventory that stores all the objects I have. It is not an entity, but objects can be added to it or retrieved from it anytime at anywhere without specific actions. The mined or crafted objects will be added to this inventory, and the materials and tools to use are also from this inventory. Objects in the inventory can be directly used. Don't write the code to obtain them. If you plan to use some object not in the inventory, you should first plan to obtain it. You can view the inventory as one of my states, and it is written in form of a dictionary whose keys are the name of the objects I have and the values are their quantities.

4. You will get the following information about my current state:
- inventory: a dict representing the inventory mentioned above, whose keys are the name of the objects and the values are their quantities
- environment: a string including my surrounding biome, the y-level of my current location, and whether I am on the ground or underground
Pay attention to this information. Choose the easiest way to achieve the goal conditioned on my current state. Do not provide options, always make the final decision.

5. You must describe your thoughts on the plan in natural language at the beginning. After that, you should write all the actions together. The response should follow the format:
{
    "explanation": "explain why the last action failed, set to null for the first planning",
    "thoughts": "Your thoughts on the plan in natural languag",
    "action_list": [
        {"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
        {"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
        {"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"}
    ]
}
The action_list can contain arbitrary number of actions. The args of each action should

> correspond to the type mentioned in the Arguments part. Remember to add "'dict'" at the beginning and the end of the dict. Ensure that you response can be parsed by Python json.loads
>
> 6. I will execute your code step by step and give you feedback. If some action fails, I will stop at that action and will not execute its following actions. The feedback will include error messages about the failed action. At that time, you should replan and write the new code just starting from that failed action.

A.3.2 User Query

> **USER:**
>
> My current state:
> - inventory: {inventory}
> - environment: {environment}
>
> The goal is to {goal}.
>
> Here is one plan to achieve similar goal for reference: {reference plan}.
>
> Begin your plan. Remember to follow the response format.
> *or* Action {successful action} succeeded, and {feedback message}. Continue your plan. Do not repeat successful action. Remember to follow the response format.
> *or* Action {failed action} failed, because {feedback message}. Revise your plan from the failed action. Remember to follow the response format.

A.4 Memory

A.4.1 Learning Process

We maintain the text-based memory with a dictionary, whose keys are sub-goals and values are lists of successful action sequences for the corresponding sub-goals. The construction and update of the memory are through the following learning process:

- When encountering a new sub-goal that is not in the memory, the LLM planner creates plans without reference. Once the sub-goal is achieved, the entirely executed action sequence would be stored into the memory.

- When encountering a sub-goal with memory, the first action sequence in the recording list for this goal is retrieved as the reference plan, with which the LLM planner tries to achieve the goal. If it succeeds, the new executed action sequence will be added to the last of the recording list.

- For each sub-goal, once the number of action sequences recorded in its list reaches N, we pop all the N sequences and use LLM to summarize them into a common plan solution suitable for various scenarios, which is then put first in the list. N is set to 5 in all our experiments.

To learn the memory for obtaining all items, starting from scratch each time would take a long time. In addition, it is necessary to avoid spending the most of time on learning simple tasks and not investing enough in learning difficult tasks. To improve the learning efficiency, we suggest to study the sub-goals individually one by one. We first use our LLM Decomposer to generate sub-goal trees for all items, acquiring the set of all sub-goals involved. Then for each sub-goal, the LLM planner plays multiple times given its prerequisites including the required materials and tools. The learning process of the sub-goal is finished once we obtain $N = 5$ successful action sequences and summarize them into one common plan solution for reference.

A.4.2 Implementation of Memory Summarization

We also use `gpt-3.5-turbo` from OpenAI API for memory summarization but in a different dialogue. We use the following prompt to instruct the summarization with the role "SYSTEM". The slot {action description} is replaced with the same descriptions of interfaces of the structured actions as Sec. A.3.1. We list all the action sequences to be summarized in the query with the role "USER", which is fed into the LLM for response.

B Results of All Items

We provide the success rate of all items in the entire Minecraft Overworld Technology Tree in Tab. 4. We have attached a video of obtaining a *diamond* in the supplementary materials.

Experiment Setting. Considering the large number of items, including those difficult to be obtained, we implemented an incremental testing strategy. This strategy is designed to keep the testing costs within a reasonable range, while also accounting for the rarity of certain items. We avoided a uniform increase in the number of tests across all items to accommodate the hardest-to-obtain ones, which would have resulted in prohibitive testing costs. Instead, we employed a incremental testing process.

For each item, we begin with 20 games. If the success count is less than or equal to 1, we increase to 50 games. If the success count remains less than or equal to 1, we further increase to 100, and eventually 200 games. This testing continues until the success count finally exceeds 1, or we complete 200 games. By following this efficient strategy, we ensure a cost-effective and reliable evaluation of each item, regardless of its availability. Moreover, because some items need long-term planning and crafting chain, we do not set restrictions on the time limit or query limit.

Exploring Biome. Biomes can be a key factor that strongly influences the success rate. Some items, like cactus, pumpkin, or melon, can only be found in specific biomes. The distribution of biomes highly limits the success rate of some items.

Table 4: **Success rate for all 262 items in the entire Minecraft Overworld Technology Tree.**

Item Name	Success Rate	Item Name	Success Rate	Item Name	Success Rate	Item Name	Success Rate
acacia boat	100.0	stone sword	100.0	gravel	80.0	beetroot seeds	40.0
acacia door	100.0	stonebrick	100.0	iron boots	80.0	diamond boots	40.0
acacia fence	100.0	sugar	100.0	iron trapdoor	80.0	diamond helmet	40.0
acacia fence gate	100.0	tallgrass	100.0	leather chestplate	80.0	golden axe	40.0
acacia stairs	100.0	trapdoor	100.0	leather leggings	80.0	golden pickaxe	40.0
beef	100.0	wheat	100.0	bone block	75.0	red mushroom	40.0
birch boat	100.0	wheat seeds	100.0	bow	75.0	red mushroom block	40.0
birch door	100.0	wooden axe	100.0	chest minecart	75.0	diamond leggings	35.0
birch fence	100.0	wooden button	100.0	furnace minecart	75.0	golden boots	35.0
birch fence gate	100.0	wooden door	100.0	hopper	75.0	ink sac	35.0
birch stairs	100.0	wooden hoe	100.0	iron helmet	75.0	sticky piston	35.0
boat	100.0	wooden pickaxe	100.0	minecart	75.0	beetroot soup	30.0
bone	100.0	wooden pressure plate	100.0	mossy cobblestone	75.0	golden helmet	30.0
bone meal	100.0	wooden shovel	100.0	vine	75.0	lead	30.0
bowl	100.0	wooden slab	100.0	waterlily	75.0	map	30.0
chest	100.0	wooden sword	100.0	banner	70.0	mushroom stew	30.0
chicken	100.0	yellow flower	100.0	brick	70.0	brick stairs	25.0
cobblestone	100.0	armor stand	95.0	clay ball	70.0	cactus	25.0
cobblestone wall	100.0	book	95.0	diamond	70.0	cake	25.0
cooked beef	100.0	bread	95.0	diamond shovel	70.0	clock	25.0
cooked chicken	100.0	coal	95.0	dropper	70.0	cooked rabbit	25.0
cooked mutton	100.0	fireworks	95.0	feather	70.0	diamond chestplate	25.0
cooked porkchop	100.0	gunpowder	95.0	iron bars	70.0	obsidian	25.0
crafting table	100.0	iron ingot	95.0	iron door	70.0	rabbit	25.0
dark oak boat	100.0	iron nugget	95.0	jukebox	70.0	rabbit hide	25.0
dark oak door	100.0	iron ore	95.0	lapis lazuli	70.0	deadbush	20.0
dark oak fence	100.0	iron shovel	95.0	noteblock	70.0	golden leggings	20.0
dark oak fence gate	100.0	item frame	95.0	piston	70.0	golden rail	20.0
dark oak stairs	100.0	leather	95.0	rail	70.0	lapis block	20.0
dirt	100.0	rotten flesh	95.0	redstone	70.0	writable book	20.0
double plant	100.0	shield	95.0	redstone torch	70.0	baked potato	15.0
fence	100.0	spider eye	95.0	cauldron	65.0	carrot	15.0
fence gate	100.0	stone slab	95.0	diamond hoe	65.0	diamond block	15.0
furnace	100.0	torch	95.0	diamond sword	65.0	emerald block	15.0
glass	100.0	trapped chest	95.0	emerald	65.0	golden chestplate	15.0
glass bottle	100.0	tripwire hook	95.0	iron leggings	65.0	potato	15.0
glass pane	100.0	carpet	90.0	tnt	65.0	pumpkin	15.0
ladder	100.0	coal block	90.0	arrow	60.0	pumpkin seeds	15.0
lever	100.0	grass	90.0	compass	60.0	carrot on a stick	10.0
log	100.0	heavy weighted pressure plate	90.0	flower pot	60.0	jungle boat	10.0
mutton	100.0	iron hoe	90.0	iron chestplate	60.0	jungle door	10.0
oak stairs	100.0	iron sword	90.0	brick block	55.0	jungle fence	10.0
paper	100.0	leather boots	90.0	clay	55.0	jungle fence gate	10.0
planks	100.0	leather helmet	90.0	dispenser	55.0	jungle stairs	10.0
porkchop	100.0	leaves	90.0	gold ingot	55.0	lit pumpkin	10.0
red flower	100.0	painting	90.0	gold nugget	55.0	melon	10.0
reeds	100.0	shears	90.0	gold ore	55.0	melon block	10.0
sand	100.0	snow	90.0	golden shovel	55.0	melon seeds	10.0
sandstone	100.0	snow layer	90.0	hardened clay	55.0	gold block	8.0
sapling	100.0	snowball	90.0	hopper minecart	55.0	golden carrot	8.0
sign	100.0	string	90.0	iron block	55.0	pumpkin pie	8.0
spruce boat	100.0	wool	90.0	slime ball	55.0	red sandstone	6.0
spruce door	100.0	bed	85.0	activator rail	50.0	red sandstone stairs	6.0
spruce fence	100.0	brown mushroom	85.0	detector rail	50.0	speckled melon	6.0
spruce fence gate	100.0	brown mushroom block	85.0	diamond axe	50.0	stone slab2	6.0
spruce stairs	100.0	bucket	85.0	diamond pickaxe	50.0	anvil	4.0
stick	100.0	flint	85.0	egg	50.0	apple	4.0
stone	100.0	hay block	85.0	lava bucket	50.0	enchanting table	4.0
stone axe	100.0	iron axe	85.0	repeater	50.0	enchanted book	3.0
stone brick stairs	100.0	iron pickaxe	85.0	tnt minecart	50.0	poisonous potato	2.0
stone button	100.0	milk bucket	85.0	bookshelf	45.0	golden apple	1.0
stone hoe	100.0	sandstone stairs	85.0	golden hoe	45.0	rabbit foot	1.0
stone pickaxe	100.0	water bucket	85.0	golden sword	45.0	slime	1.0
stone pressure plate	100.0	fermented spider eye	80.0	light weighted pressure plate	45.0	rabbit stew	0.5
stone shovel	100.0	fishing rod	80.0	redstone block	45.0		
stone stairs	100.0	flint and steel	80.0	beetroot	40.0		

C   Supplementary Ablations

We make a more detailed comparison between our GITM with RL-based methods in Tab. 5. The most straightforward pipeline is to directly map the goal into keyboard/mouse operations. We gradually add goal decomposition and structured action stages into the pipeline, and ablate the use of RL-based models or LLM.

Table 5: **Ablation study.** The milestone items from left to right are crafting table ⬛, wooden pickaxe ⛏, stone pickaxe ⛏, iron pickaxe ⛏, and diamond 💎. The success rate is calculated under time limit of 12000 steps (total) and query limit of 30 (each sub-goal). "Goal Decomp." indicates whether to use LLM Decomposer to decompose the goal into sub-goals. "Goal / Sub-Goal to Structured Actions / Keyboard & Mouse Mapping" indicates which method is used for the mapping from goal / sub-goals to structured actions / keyboard & mouse operations.

	Goal Decomp.	Structured Action	Goal / Sub-Goal to Structured Actions / Keyboard & Mouse Mapping	Success Rate (%)				
				⬛	⛏	⛏	⛏	💎
(a)			Specialist RL Model (VPT)	100.0	100.0	100.0	85.0	20.0
(b)			Goal-conditioned RL Model (DEPS)	0.0	0.0	0.0	0.0	0.0
(c)			Our LLM Planner	0.0	0.0	0.0	0.0	0.0
(d)	✓		Goal-conditioned RL Model (DEPS)	90.0	80.0	30.0	0.0	0.0
(e)	✓		Our LLM Planner	0.0	0.0	0.0	0.0	0.0
(f)		✓	Our LLM Planner	57.5	32.5	5.0	0.0	0.0
(g)	✓	✓	Our LLM Planner	100.0	100.0	100.0	95.0	67.5

Implementation Details. We can only find open-sourced RL models from VPT [2] and DEPS [25], so they are adopted for the ablation. VPT model is specifically trained for the `ObtainDiamond` challenge, while DEPS model can use goal description as input to guide the model's output. We refer to them as specialist RL model and goal-conditioned RL model, respectively. As for the use of LLM Planner, we note that if structured action is not used, LLM Planner will be inevitably asked to output reasonable keyboard/mouse operations. However, LLM Planner does not have access to environment observations, so it cannot directly output reasonable keyboard/mouse operations.

Direct Mapping. See Tab. 5(a)(b)(c). It is hard to directly mapping the long-horizon goal into reasonable keyboard/mouse operations. While a specialist RL model (*i.e.*, VPT) can deliver promising results, it requires large amount of data and computational resources to train such a model [2] (720 V100 GPUs for 9 days). Moreover, a different goal will require further training of the specialist RL model, limiting the versatility of this paradigm. The goal conditional RL model (*i.e.*, DEPS) cannot achieve the goal, because the model [25] we have access to is not generalizable to all scenarios. If only the final goal is given, it will ignore preconditions, such as not crafting the necessary iron pickaxe when mining diamonds. LLM also fails to accomplish the goal. The primary reason is that it can not handle environment observation and keyboard/mouse operations well.

Structured Action. We design structured actions to interact with the environment, and provide an abstract interface. Tab. 5(f) shows that adding structured action significantly improves LLM's performance. This is because structured actions can deal with environment observations and keyboard/mouse operations more precisely, unleashing the reasoning potential of LLM. We are not aware of a RL model using structured actions currently. It is possible for structure actions to enhance the RL model as well, and we will explore it in the future work.

Goal Decomposition. Decomposing the goal into sub-goals can simplify the whole task. Tab. 5(b)(d) and Tab. 5(f)(g) show its effectiveness for both goal-conditioned RL model and our method. By exploiting goal decomposition, it is possible for our method to accomplish long-term tasks with high success rate.

Comparison between RL-based methods. We also note the paradigm shift from traditional RL-based methods to our GITM leads to a great performance boost. Comparing Tab. 5(d)(g), where we only change the goal-conditioned RL model to LLM with strutured actions, our method significantly outperforms the RL model.

D `ObtainDiamond`

We demonstrate a case of the popular `ObtainDiamond` challenge in Fig. 7. During the process, the agent have to collect materials, *i.e.*, log, stone and iron ore, as shown in Fig 7(a)(c)(e). Necessary tools, *i.e.*, wooden pickaxe, stone pickaxe, furnace and iron pickaxe are also crafted in Fig 7(b)(d)(f)(h).
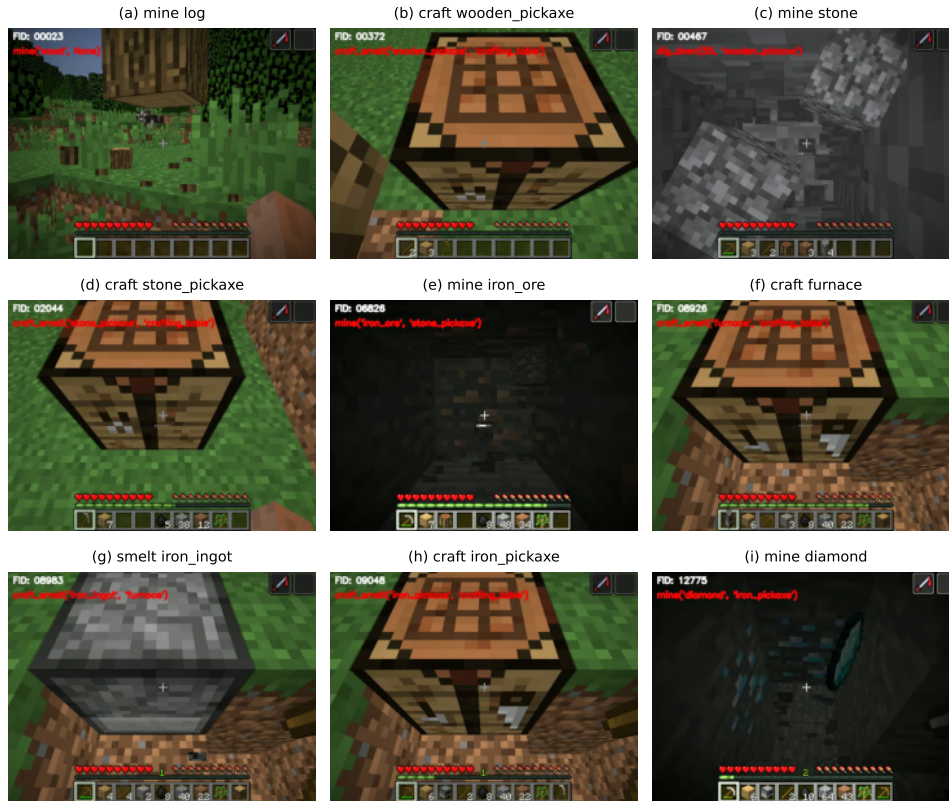
Figure 7: **A case of the popular** `ObtainDiamond` **challenge.** Figure(e)(i) are enhanced in brightness for better display.

Finally the diamond is obtained in Fig 7(i). We have attached a video of obtaining a *diamond* in the supplementary materials.

E   Applications

(a) Shelter with Farmland   (b) Iron Golem   (c) Redstone Circuit   (d) Nether Portal

Figure 8: Demonstration of the applications. GITM can construct `Shelter with Farmland` and `Iron Golem` for survival, `Redstone Circuit` for automation equipment, and `Nether Portal` for the Nether world exploration.

Our proposed GITM makes survival and the nether exploration possible in Minecraft which has never been accomplished by existing agents. To achieve this, our agent builds four necessary items, including `Shelter with Farmland`, `Iron Golem`, `Redstone Circuit`, and `Nether Portal`, shown in Fig. 8. `Shelter with Farmland` is firstly built to keep the agent from being attacked by monsters at night and provide enough food. `Iron Golem` can automatically attack monsters to protect the agent and the shelter. `Redstone Circuit` is the foundation of all automation equipment. `Nether Portal` is the entrance to the Nether world.