GATE: Graph-based Adaptive Tool Evolution Across Diverse Tasks

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown great promise in tool-making, yet existing frameworks often struggle to efficiently construct reliable toolsets and are limited to single-task settings. To address these challenges, we propose GATE (Graph-based Adaptive Tool Evolution), an adaptive framework that dynamically constructs and evolves a hierarchical graph of reusable tools across multiple scenarios. We evaluate GATE on open-ended tasks (Minecraft), agent-based tasks (TextCraft, DABench), and code generation tasks (MATH, Date, TabMWP). Our results show that GATE achieves up to 4.3× faster milestone completion in Minecraft compared to the previous state-of-the-art method, and provides an average improvement of 9.23% over existing tool-making methods in code generation tasks and 10.03% in agent tasks. Further analysis shows that GATE exhibits strong adaptive evolution capabilities, effectively balancing tool quantity, complexity, and functionality while maintaining high efficiency. Code and data are available at https://anonymous.4open.science/r/GATE-0.

1 Introduction

Large Language Models (LLMs) have demonstrated strong capabilities in code generation (Yang et al., 2025; Wang et al., 2025b), supporting a range of complex tasks such as mathematical reasoning (Guan et al., 2025; Li et al., 2025), and tabular computation (Wu et al., 2025). By generating executable code, LLMs extend their practical utility beyond static language modeling and enable more advanced agent-based decision making, as exemplified by frameworks like CodeActAgent (Wang et al., 2024b) and Openhands (Wang et al., 2025a). However, existing approaches typically treat each generated program as an independent instance, lacking robust mechanisms for reusing or evolving functional modules across tasks.

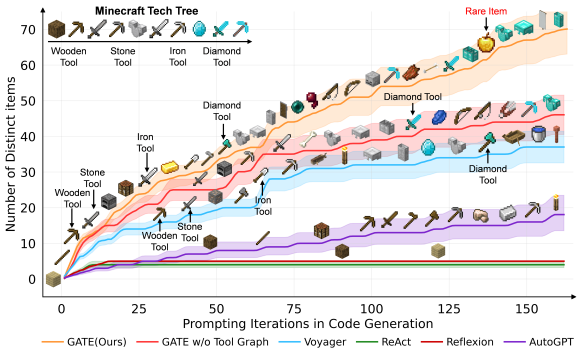

Figure 1: Performance of GATE in Minecraft. GATE continually discovers new items and skills during exploration, significantly outperforming other methods.

To overcome this, recent studies (Wang et al., 2023; Cai et al., 2023; Qian et al., 2023; Stengel-Eskin et al., 2024) have explored reusable tool libraries derived from task experiences. However, existing approaches still face key limitations hindering adaptability and scalability: (1) **Toolset Redundancy and Inefficiency**: Many frameworks generate tools independently per task, resulting in large, redundant libraries that are hard to manage and reuse. For example, Voyager (Wang et al., 2023) lacks tool deduplication, while CREATOR (Qian et al., 2023) and CRAFT (Yuan et al., 2023) create one tool per task, causing excessive repetition. Even Regal (Stengel-Eskin et al., 2024), which targets simplicity, is limited to basic arithmetic wrappers. (2) **Limited Flexibility and Generalization**: Most methods rely on fixed tool creation strategies or rigid interfaces tailored to narrow domains, making it difficult to adapt to diverse or evolving task requirements. For instance, Voyager (Wang et al., 2023) is restricted to the Minecraft environment, while others (Qian et al., 2023; Yuan et al., 2023; Stengel-Eskin et al., 2024) focus narrowly on code generation, lacking mechanisms to generalize beyond static task formats or domains.

In this paper, we present GATE (Graph-based Adaptive Tool Evolution), a framework centered

on a hierarchical **Tool Graph**, where nodes represent tools and edges encode invocation relationships. This layered graph acts as a structured substrate for maintaining, organizing, and adaptively evolving the tool library. Built upon it, we propose **GraphRank Retrieval**, which ranks and selects tools by combining semantic relevance with structural centrality, promoting efficient reuse and dynamic adaptation. To construct and refine the graph, two agents collaborate: the *Task Solver* identifies tool requirements, while the *Tool Manager* retrieves, composes, and updates tools through creation, merging, and pruning. This graph-centric and adaptive integration of structure, retrieval, and agent-driven maintenance distinguishes GATE from prior frameworks, enabling scalable and continual tool evolution across diverse tasks.

To comprehensively evaluate the effectiveness of our framework, we conduct experiments on both open-ended and closed-ended tasks. As shown in Figure 1, GATE achieves 3.5× better item discovery and 4.3× faster tech tree mastery in Minecraft compared to the previous SOTA. It also outperforms baselines by 5–32% in agent tasks and exceeds other tool-making methods by an average of 12.6% in code generation. Analysis reveals the adaptive evolution of the tool graph over tasks. Compared to prior methods, GATE offers a favorable trade-off in tool library size, complexity, and redundancy. Our contributions are summarized as follows:

1. We introduce the hierarchical **Tool Graph** as the core of GATE, encoding tools and their invocation across abstraction levels, and propose **GraphRank Retrieval** for efficient, controllable evolution.
2. Based on this design, we develop GATE, a framework in which two agents collaborate to evolve the Tool Graph through operations such as construction, merging, and pruning.
3. We comprehensively evaluate GATE, achieving SOTA performance, showing a favorable trade-off between performance and tool library size while demonstrating adaptability through task-specific Tool Graph.

2 Related Work

Reusable Tool Construction LLMs demonstrate strong code-generation abilities (Zhou et al., 2023; Zeng et al., 2025; Wang et al., 2025b). To improve reusability, CREATOR (Qian et al., 2023) separates planning and execution, while LATM (Cai et al., 2023), CRAFT (Yuan et al., 2023), and REGAL (Stengel-Eskin et al., 2024) pre-construct tools but often produce shallow or redundant wrappers. Trove (Wang et al., 2024c) composes tools at inference yet suffers from hallucination. Overall, existing methods either over-generate or lack structural depth, limiting scalability.

Tool Selection and Retrieval for Task Solving Another line of work focuses on retrieving tools for complex tasks (Wang et al., 2024a; Shi et al., 2025). ToolRerank (Zheng et al., 2024) improves selection via adaptive truncation and hierarchy-aware scoring, while Re-Invoke (Chen et al., 2024) enhances retrieval through synthetic queries and multi-view ranking. COLT (Qu et al., 2024b) integrates semantic and structural signals, and AvaTaR (Wu et al., 2024) and DRAFT (Qu et al., 2024a) optimize prompts and documentation. Most approaches, however, overlook cost-efficiency and adaptability. In contrast, our method combines retrieval relevance with structural importance, enabling scalable, efficient tool selection in dynamic tasks.

3 GATE Framework

As tool libraries expand, existing methods struggle with redundancy, limited scalability, and inefficient retrieval. To address these challenges, we propose the **GATE framework** (Figure 2), which unifies tool construction, evolution, and retrieval. A central component of GATE is the **Tool Graph**, a hierarchical representation where nodes encode tools and edges capture invocation relationships. In this section, we first introduce the structure of the Tool Graph (§3.1), then present GraphRank Retrieval for tool selection (§3.2), and finally describe how the Tool Graph is constructed and evolved (§3.3).

3.1 Tool Graph Architecture

GATE's tool graph is a hierarchical undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes tool nodes and $\mathcal{E}$ the edges representing dependencies.

Node($\mathcal{V}$) The node set $\mathcal{V}$ consists of two types of tools: basic tools, pre-defined by humans, and composed tools, which are created during training. Each node $v_i \in \mathcal{V}$ stores metadata, including the tool's name, docstring, implementation code, usage frequency, and layer position $L(v_i)$. The layer position of basic tools is set to 1, as they form the foundation of the graph. For composed tools, the layer position is determined by the dependencies between the tool and other nodes, where $\text{Call}(v_j, v_i)$ indicates whether tool v_j invokes tool

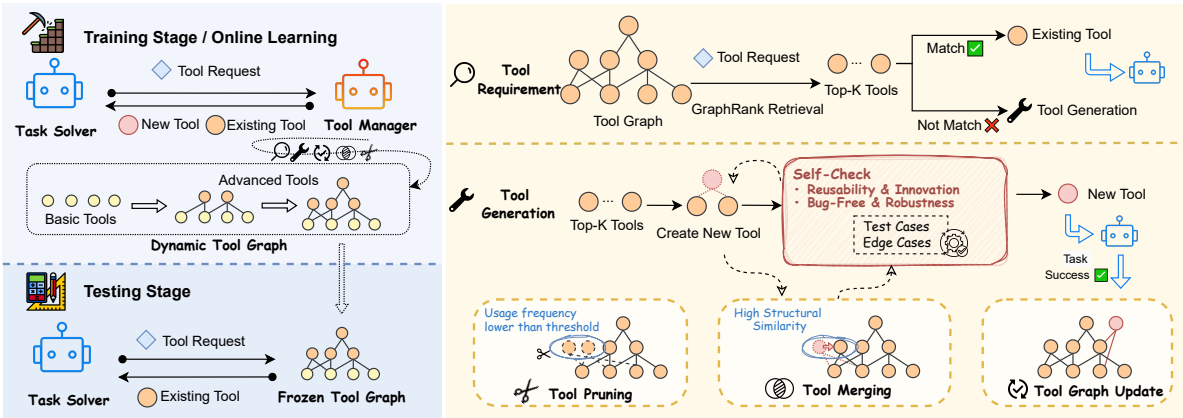

Figure 2: GATE consists of two agents: the Task Solver and the Tool Manager, which interact with an Adaptive Tool Graph. Key processes include Tool Requirement, Generation, Pruning, Merging, and Tool Graph Updates.

v_i in its implementation:

$$\text{Call}(v_j, v_i) = \begin{cases} 1, & \text{if } v_j \text{ calls } v_i, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

The dependencies of node v_j, denoted as $D(v_j) \subset \mathcal{V}$, are given by $D(v_j) = \{v \in \mathcal{V} \mid \text{Call}(v_j, v) = 1\}$. The layer position of v_j is then computed as:

$$L(v_j) = \max_{v \in D(v_j)} L(v) + 1 \quad (2)$$

Edge ($\mathcal{E}$) The edge set $\mathcal{E}$ represents invocation relationships between tools. For any two nodes $v_i, v_j \in \mathcal{V}$, if an invocation relationship exists between them, an edge is established, represented through the adjacency matrix E. This construction of edges is crucial for capturing the functional dependencies between tools, reflecting how tools interact and depend on each other within the graph. The adjacency matrix $E = \{e_{ij}\}_{N \times N}$ represents these relationships, where e_{ij} is defined as:

$$e_{ij} = e_{ji} = \text{Call}(v_i, v_j) \vee \text{Call}(v_j, v_i) \quad (3)$$

3.2 GraphRank Retrieval

To capture semantic and structural similarity among tools, we propose *GraphRank Retrieval*, combining vector similarity with a modified PageRank (Xing and Ghorbani, 2004). Retrieval is cast as a random walk on the Tool Graph with prior p_0 and transition matrix M; nodes are ranked by stationary distribution GR, and the top-k are returned.

Given a query and k, we embed it using *text-embedding-ada-002* (OpenAI, 2022) and compute the cosine similarity s_i between it and each tool's docstring embedding. These similarity scores are then subsequently normalized to get p_0:

$$p_0 = \left[\frac{s_1}{\sum s_i}, \frac{s_2}{\sum s_i}, \ldots, \frac{s_N}{\sum s_i} \right] \quad (4)$$

To model the transition probability distribution from each node to others, we treat the distribution as uniform, with the transition probabilities determined by the weight matrix E. These probabilities are derived from the column-normalized weight matrix $M = \{m_{ij}\}_{N \times N}$ as follows:

$$m_{ij} = \begin{cases} e_{ij} / \sum_{l=1}^{N} e_{lj} & \text{if } \sum_{l=1}^{N} e_{lj} > 0, \\ 1/N & \text{otherwise.} \end{cases} \quad (5)$$

For isolated nodes, the transition probabilities to all other nodes are uniformly set to $1/N$, thereby ensuring full participation in the Markov chain. Given the probability distribution GR_{t-1} at time step $t-1$, the update rule becomes:

$$GR_t = (1-d) \cdot p_0 + d \cdot M^T GR_{t-1} \quad (6)$$

where $d \in [0, 1]$ controls the balance between the prior distribution p_0 and the structural transitions encoded in M. We set $d = 0.4$ to slightly favor the semantic prior over structural propagation (see Figure 12 for ablation results).

3.3 Tool Graph Construction

Building on the Tool Graph, GATE employs two agents that jointly construct and maintain it (Figure 2). The Task Solver analyzes tasks, executes code, and decides when to request or terminate, supporting the actions **ToolRequest**, **CallTools**, and **Terminate**. The Tool Manager creates and edits tools to form toolchains, with actions **RetrieveTool**, **CreateTool** and **EditTool**. Both agents retrieve candidates via GraphRank from the Tool Graph and a base set. As shown in Figure 2 and Figure 3, the construction loop has four stages: Tool Requirement, Tool Creation and Self-Check, Tool Merging, and Tool Graph Update (adding nodes,

linking edges, and pruning low-utility tools). Detailed prompts for both agents are in Appendix E.1.

Tool Requirement Given a task, the Task Solver identifies the required tool functionalities and sends them to the Tool Manager. GraphRank retrieves relevant tools for each requirement, which the Tool Manager then leverages to construct new tools.

Tool Creation and Self-Check Since retrieved tools alone cannot always satisfy new task requirements, the Tool Manager composes and extends them into candidate tools. Their design follows four principles: (1) **Reusability**: clear naming and general interfaces; (2) **Reuse of Existing Tools**: promote modularity via composition; (3) **Innovation**: extend or integrate functionality; and (4) **Completeness**: cover edge cases and exceptions. To ensure reliability, each candidate undergoes a Self-Check, an independent review against these principles. Passing tools are added to $\mathcal{V}_{\text{checked}}$; others are iteratively refined.

Tool Merging After creating new tools, we assess their overlap with existing tools to reduce redundancy and improve the tool graph structure. $\mathcal{V}_{\text{check}}$ are compared with existing tools $\mathcal{V}$ using the Smith-Waterman algorithm (Smith et al., 1981) to measure structural similarity. Redundant tools of v_i are represented as $\mathcal{R}(v_i)$. If $\mathcal{R}(v_i)$ is not empty, the Tool Manager combines the functionalities of v_i and $\mathcal{R}(v_i)$ to generalize a new tool, replacing v_i.

Tool Graph Update Only correctly solved tasks are considered for updating the tool graph. The tools finally used in these correct solutions are denoted as $\mathcal{V}_{\text{used}} \subset \mathcal{V}_{\text{checked}}$. We then analyze the invocation relationships among the utilized tools, where $v_i \in \mathcal{V}_{\text{used}}$ and $v_j \in \mathcal{V}$, and update the edge set $\mathcal{E}$ and node set $\mathcal{V}$ as follows:

$$\begin{aligned} \mathcal{E} &\leftarrow \mathcal{E} \cup \{(v_i, v_j) \mid \text{Call}(v_i, v_j) = 1\} \\ \mathcal{V} &\leftarrow \mathcal{V} \cup \mathcal{V}_{\text{used}} \end{aligned} \tag{7}$$

Finally, we remove redundant tools $\mathcal{R}(v_i)$ for $v_i \in \mathcal{V}_{\text{used}}$, unless they contributed to a higher-level tool.

Pruning To optimize the tool graph, pruning is periodically performed by removing low-usage nodes, with thresholds that adapt over time and by tool level. This prevents premature removal of high-level but critical tools. To preserve structural integrity, we enforce that if a node is retained, all its child nodes are also preserved.

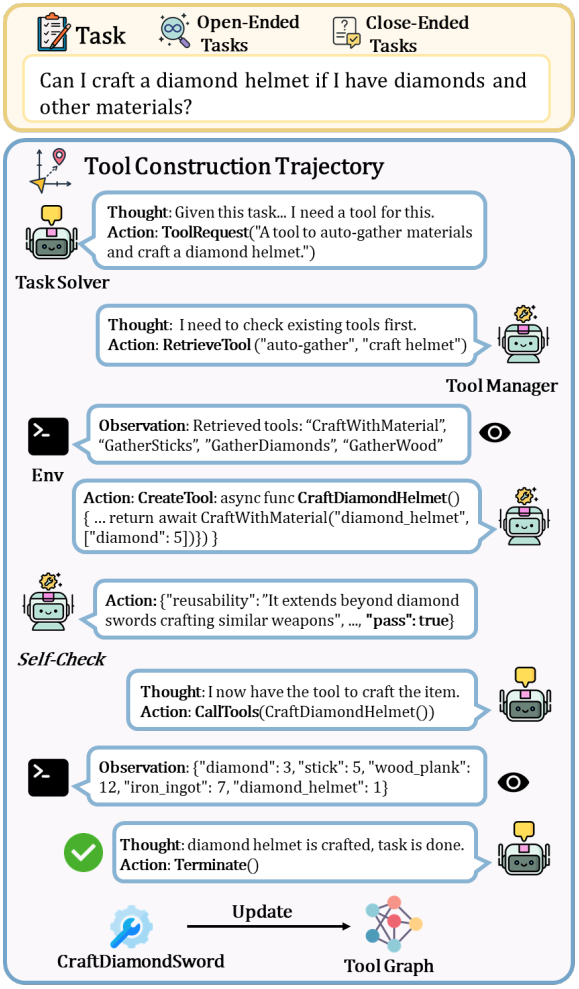

Figure 3: A simple example illustrating how the Task Solver and Tool Manager collaborate to construct tools and update the Tool Graph. More examples of created tools are provided in Appendix F.1.

4 Experiment Setup

We comprehensively evaluate GATE on open-ended and closed-ended tasks, including code generation and dynamic agent tasks across games, math, and data science.

4.1 Open-Ended Tasks

Open-ended tasks lack fixed solutions, requiring agents to explore, adapt, and evolve over time. They assess long-horizon reasoning, tool evolution, and the ability to pursue unbounded goals. Detailed task setups are provided in Appendix A.1.

Benchmark We use Minecraft as the experimental platform, where agents collect resources and craft tools to achieve objectives. The environment builds on Voyager (Wang et al., 2023) and uses Mineflayer (PrismarineJS, 2013) JavaScript APIs for motor control. Performance is measured by the number of iterations to complete tool upgrades,

with each code execution counted as one iteration.

Baselines We compare GATE with several representative agent algorithms: ReAct (Yao et al., 2022), Reflexion (Shinn et al., 2023), AutoGPT (Richards), and Voyager (Wang et al., 2023). For ReAct, Reflexion, and AutoGPT, we report the results as presented in prior work (Wang et al., 2023). **Voyager is re-evaluated using GPT-4o under our settings for fair comparison.**

Implementation GATE operates via online learning: the Task Solver incrementally solves subtasks while the Tool Manager maintains and prunes the tool graph. We use GPT-4o as the base model for all tasks, with settings: top-5 for tool retrieval (Wang et al., 2023), and pruning every 40 steps.

4.2 Close-Ended Tasks

Close-ended tasks are characterized by predefined goals and explicit ground truth. We conduct experiments on two representative settings: single-turn code generation tasks and multi-turn agent tasks. Detailed task setups are provided in Appendix A.2.

Benchmark For single-turn tasks, we adopt the algebra subset (levels 4–5) from MATH (Hendrycks et al., 2021), levels 7–8 from TabMWP (Grand et al., 2023), and the Date dataset (Srivastava et al., 2022). For multi-turn tasks, we use TextCraft (Côté et al., 2019) (text-based game) and DABench (Hu et al., 2024) (data science benchmark). As DABench lacks prior evaluation protocols, we split it into train/test sets. For all other datasets, we follow the original splits (Yuan et al., 2023; Stengel-Eskin et al., 2024). Dataset sizes (train/test): MATH (200/405), Date (66/180), TabMWP (200/470), TextCraft (98/77), and DABench (98/158). We report average accuracy on the test sets.

Baselines For code tasks, we compare with PoT (Chen et al., 2022), LATM (Cai et al., 2023), CREATOR (Qian et al., 2023), CRAFT (Yuan et al., 2023), and REGAL (Stengel-Eskin et al., 2024). For multi-turn agents, we include ReAct (Yao et al., 2022), Reflexion (Shinn et al., 2023), and Plan-Execution (Shridhar et al., 2023; Yang et al., 2023). **For tool-building baselines, tools are retrained with GPT-4o on the same datasets.**

Implementation GATE constructs tools during training with GPT-4o, followed by a pruning phase. During testing, the tool graph is frozen and used for retrieval-augmented inference, where prompts include training data and tool examples. Following prior work, we set the retrieval size to 3. All tool-building methods are evaluated using the same training/test splits and model configuration.

Models To ensure fair comparison, all methods are evaluated using the same set of models and tool libraries. We use in-context learning on open-source models (*Qwen2.5-7B-Instruct*, *Qwen2.5-Coder-7B*, *Qwen2.5-14B*, *Deepseek-Coder-6.7B*, *Deepseek-Coder-33B*) and closed-source models (*GPT-3.5-turbo-1106*, *Claude-3-haiku*, *GPT-4o*). Temperature is fixed at 0.3, and **each experiment is run 3 times with average results reported**.

5 Main Results

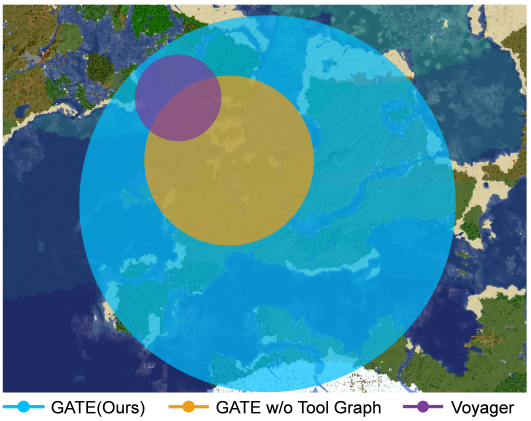

Figure 4: Bird's-eye views of explored regions in Minecraft maps. Full results are in Appendix B.2.

GATE Enhances Open-Ended Task Performance via Structured Tool Evolution. In open-ended environments, GATE outperforms Voyager, discovering unique items (Figure 1). It reaches wooden, stone, and iron milestones 23.0×, 13.4×, and 7.5× faster than baselines (Table 1), crafts Diamond Tool 4.34× faster, and travels 2.7× longer distances (Figure 4). These results show that structured tool evolution in GATE drives faster progress and broader exploration in open-ended tasks.

GATE Enables Self-Improvement on GPT-4o and Boosts Performance on Other Models in Close-Ended Tasks. Table 2 demonstrates GATE's effectiveness across open- and closed-source models. GATE enables self-improvement on GPT-4o and boosts other models. On average, GPT-4o improves by 5%, while other models gain 10.03% and 9.23% on agent and code sub-tasks, respectively. For instance, GPT-3.5-turbo-1106 improves 32.4% on TextCraft, and Qwen2.5-Coder-7B-Instruct achieves 16.85% on Date. These re-

Table 1: Mastery of the Tech Tree in the open-ended task. Numbers indicate required iterations, where fewer imply higher efficiency. "N/A" denotes unavailable results. Entries marked with "*" are from Voyager (Wang et al., 2023). More detailed results are in Appendix B.1.

Method	Wood Tool	Stone Tool	Iron Tool	Diamond Tool
ReAct*	N/A(0/3)	N/A (0/3)	N/A(0/3)	N/A(0/3)
Reflexion*	N/A(0/3)	N/A(0/3)	N/A(0/3)	N/A(0/3)
AutoGPT*	92±72(3/3)	94±72(3/3)	135±103(3/3)	N/A(0/3)
Voyager	7±4(3/3)	12±3(3/3)	48±19(3/3)	126±0(2/3)
GATE *w/o tool graph*	6±2(3/3)	11±5(3/3)	31±9(3/3)	125±19(3/3)
GATE *(ours)*	**4±0**(3/3)	**7±1**(3/3)	**18±3**(3/3)	**29±2**(3/3)

Table 2: Test results of different models on the close-ended task. The results are presented for both open-source and closed-source models. "w/o d." denotes the absence of the tool demo in our method. In the Agent task, "base." represents ReAct (Yao et al., 2022), "Refl." represents Reflexion (Shinn et al., 2023), and "Plan." represents Plan-Execution (Shridhar et al., 2023; Yang et al., 2023). In the Single-turn Code Tasks, "base." represents POT (Chen et al., 2022), and "Crea." represents CREATOR (Qian et al., 2023). Detailed results are provided in Appendix B.3.

Multi-turn Agent Tasks						Single-turn Code Tasks							
DS/Mthds	*Base.*	*Refl.*	*Plan.*	*Ours*	*w/o d.*	*DS/Mthds*	*Base.*	*Crea.*	*Craft*	*Latm*	*Regal*	*Ours*	*w/o d.*
Qwen2.5-7B-Instruct													
TextCraft	32.90	37.60	14.53	**44.02**	42.31	MATH_{alg}	59.42	59.75	50.86	33.33	58.02	**73.00**	69.63
DA-Bench	75.77	77.78	57.99	**83.54**	73.00	Date	57.59	58.33	62.45	61.57	74.81	**78.33**	78.15
						TabMWP	80.57	86.38	70.32	40.72	80.91	**89.78**	88.51
Qwen-Coder-7B-Instruct													
TextCraft	9.40	17.52	12.82	22.22	**23.08**	MATH_{alg}	52.02	50.94	49.47	55.17	54.07	**69.54**	63.86
DA-Bench	75.89	73.58	42.34	**81.22**	76.22	Date	61.48	57.41	61.74	52.22	74.63	78.33	**80.55**
						TabMWP	92.70	89.86	87.52	26.01	83.46	**95.11**	93.26
Deepseeker-Coder-6.7B-Instruct													
TextCraft	2.56	11.10	6.41	**15.38**	14.10	MATH_{alg}	23.95	14.34	18.52	20.38	12.10	**27.57**	24.86
DA-Bench	0.63	1.27	7.59	16.78	**18.76**	Date	58.89	43.88	46.85	29.61	53.89	**67.78**	63.89
						TabMWP	70.14	81.94	66.45	13.19	52.48	**87.80**	82.23
Qwen2.5-14B-Instruct													
TextCraft	71.79	68.37	44.87	73.93	**76.92**	MATH_{alg}	63.54	63.46	61.52	70.67	61.40	**77.16**	74.57
DA-Bench	85.44	86.58	61.51	**87.97**	86.97	Date	84.44	79.44	81.30	46.27	86.48	**88.70**	87.22
						TabMWP	93.19	90.49	73.29	45.58	91.13	94.68	**95.19**
Deepseeker-Coder-33B-Instruct													
TextCraft	8.90	12.06	2.63	**16.67**	15.28	MATH_{alg}	27.45	30.62	22.13	31.90	22.13	**35.06**	30.12
DA-Bench	38.46	53.79	8.22	**60.00**	57.05	Date	65.00	61.85	60.00	38.43	61.11	**74.16**	70.95
						TabMWP	83.69	89.72	80.92	22.10	80.92	**92.76**	87.45
GPT-3.5-turbo-1106													
TextCraft	26.92	43.59	10.27	52.85	**59.33**	MATH_{alg}	29.22	39.17	19.71	19.49	22.97	**42.39**	34.32
DA-Bench	67.30	55.06	16.24	**72.15**	71.52	Date	71.67	66.49	61.11	55.49	73.33	**76.85**	74.44
						TabMWP	75.32	80.00	69.51	49.35	76.17	**83.83**	82.34
Claude-3-haiku													
TextCraft	57.69	46.54	16.02	62.73	**66.87**	MATH_{alg}	26.34	34.16	32.59	19.05	28.48	**34.24**	32.02
DA-Bench	74.68	76.16	37.39	**82.28**	81.01	Date	81.67	45.56	74.63	53.33	70.56	**82.78**	80.37
						TabMWP	70.56	72.24	82.37	38.55	78.37	**90.78**	90.21
GPT-4o													
TextCraft	90.79	92.11	62.34	**96.15**	94.87	MATH_{alg}	60.98	69.13	62.22	61.94	61.73	69.80	**74.28**
DA-Bench	90.16	89.69	81.43	**91.60**	90.41	Date	94.44	77.78	88.33	77.06	93.89	**95.00**	95.00
						TabMWP	96.60	92.98	88.96	76.74	97.66	97.66	**97.86**

sults indicate that GATE yields consistent, transferable gains, benefiting both proprietary and open-source models via structured tool evolution.

GATE Achieves Significant Improvements Over Other Tool-Making Methods in Close-Ended Tasks. As shown in Table 2, GATE consistently outperforms other tool-making methods by an average of 10.03%. LATM and CRAFT perform worse than the baseline without additional tools, suggesting their libraries are less effective. Contrary to CREATOR and CRAFT, which separate tool making from calling (Qian et al., 2023; Yuan et al., 2023) , our results show that directly generating code yields better performance. These findings further confirm the reliability and broad applicability of GATE in tool construction.

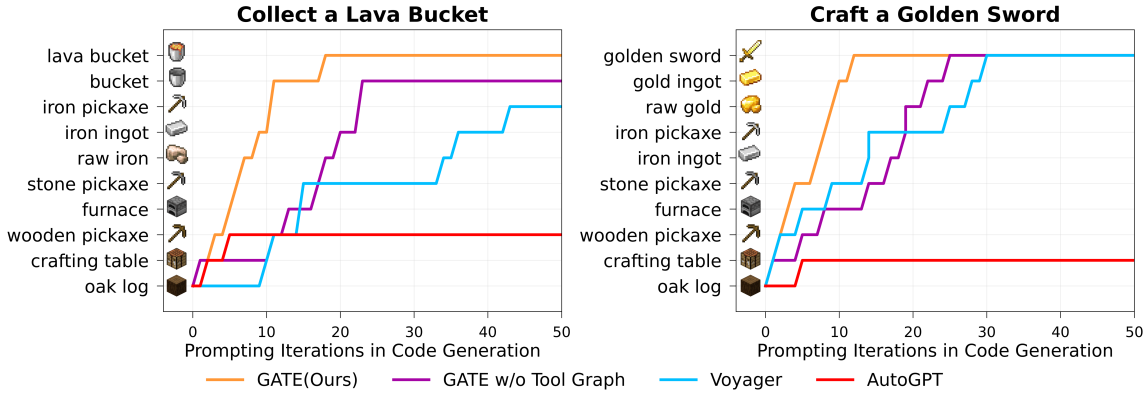

Figure 5: Zero-shot generalization on unseen tasks. The figure shows intermediate progress for each method on two tasks. ReAct and Reflexion are omitted due to limited progress. Full results are in Appendix B.2.1.

6 Analysis

6.1 Adaptation of GATE to Unseen Tasks

Table 3: Zero-shot generalization to unseen tasks. The maximal iterations are set to 50. Results with "*" are from Voyager. "w/o t. g." denotes *without tool graph*.

Method	Gold Sword	Compass	Diamond Pickaxe	Lava Bucket
ReAct*	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)
Reflexion*	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)
AutoGPT*	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)	N/A ($^0/_3$)
Voyager	46±15 ($^3/_3$)	18±2 ($^3/_3$)	22±4 ($^3/_3$)	39 ($^1/_3$)
GATE w/o t. g.	33±20 ($^3/_3$)	21±6 ($^3/_3$)	34±6 ($^3/_3$)	N/A ($^0/_3$)
GATE (ours)	**14±2 ($^3/_3$)**	**17±10 ($^3/_3$)**	**14±2 ($^3/_3$)**	**21±5 ($^3/_3$)**

GATE generalizes effectively to unseen environments through its structured tool graph. To evaluate generalization, we reset the environment, cleared the agent's inventory, and assigned unseen tasks in Minecraft. As shown in Table 3 and Figure 5, GATE completes tasks 2.2× faster than Voyager and achieves a 1.8× improvement over the ablation without the tool graph. These results show that the tool graph enhances transferability and compositional reasoning, enabling the agent to adapt and reuse knowledge in new tasks. By supporting inter-tool invocation and preserving structured dependencies, the graph provides a more coherent and extensible mechanism for tool use than Voyager's unstructured toolset, yielding greater robustness in unfamiliar environments.

6.2 Adaptive Evolution of the Tool Graph

The tool graph self-organizes and refines its hierarchy during training. As shown in Figure 6, it begins with basic tools and simple relationships, forming sparse, low-level abstractions. As task complexity grows, tool reuse intensifies, and high-frequency tools emerge as key intermediaries.

GATE adapts graph depth and structure to task demands. Figure 7 shows that tasks like MATH and TabMWP (which rely on Python libraries) yield shallow graphs with tools concentrated in lower layers (e.g., 51.5% at level two in MATH). In contrast, domain-specific tasks such as Minecraft and TextCraft produce deeper, multi-layered graphs (TextCraft up to 7 layers). As depth increases, higher-level tools realize equivalent functions with fewer operations. These patterns reflect adaptability to task complexity, supporting deeper feature extraction and flexible, hierarchical tool libraries.

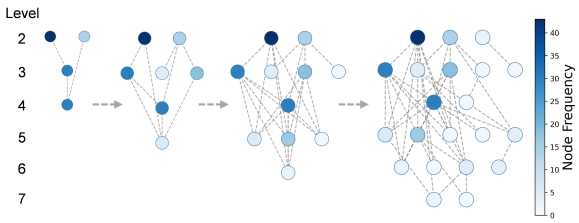

Figure 6: Tool graph evolution in Minecraft. Snapshots every 40 steps; basic tools omitted for clarity as they link across levels. Full visualizations in Appendix D.3.

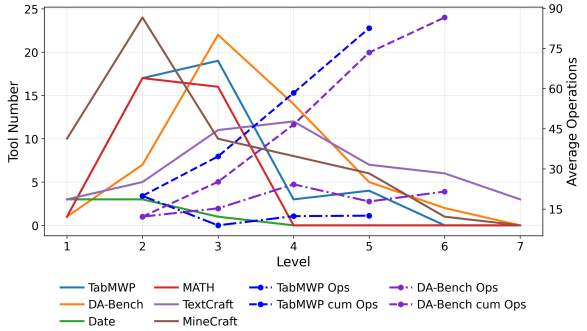

Figure 7: Layered Node Distribution of the Tool Graph. "Tool Number" is tools per level; "cum ops" is the cumulative operations and function calls.

6.3 Comparison with Other Tool Libraries

GATE achieves a superior balance between abstraction, generalizability, and efficiency across tool libraries. As shown in Table 4, GATE at-

Table 4: Comparison of Tool Libraries Constructed by Different Methods for Single-turn Code Generation. "In." represents average performance improvement.

Method	$\text{MATH}_{algebra}$			TabMWP			Date		
	cpl	lib	In.(%).	cpl	lib	In.(%)	cpl	lib	In.(%)
LATM	-	-	−3.9	-	-	−43.8	-	-	−20.2
CREATOR	34.2	405	+2.3	16.2	470	+2.6	12.6	180	−10.6
CRAFT	9.5	138	−3.2	12.2	180	−5.1	11.8	25	−4.9
REGAL	4.4	8	−2.8	5.7	7	−2.7	4.57	9	+1.7
GATE	13.6	145	**+10.7**	13.8	43	**+8.7**	11.6	11	**+8.3**

tains the highest average improvement (Avg In.) among all methods while using 86.2% fewer tools than CREATOR and CRAFT. Despite this reduction, it delivers a 9.23% performance gain, indicating that its tools are concise and semantically rich. Tool complexity (cpl), measured by AST node counts, remains moderate, reflecting efficient abstraction without unnecessary overhead.

GATE constructs compact yet expressive tools that generalize more effectively than prior libraries. Compared to CREATOR and CRAFT, GATE achieves similar or higher task success rates with a smaller library, improving efficiency by 9.23% while maintaining balanced tool complexity. Detailed tool lists of other methods are provided in Appendix D.2. In contrast, REGAL reduces tool count through pruning but sacrifices abstraction, as many of its tools collapse into simple wrappers over base functions. Leveraging a structured tool graph, GATE supports compositional reasoning and inter-tool reuse, yielding a compact, powerful library that sustains performance and generalizability.

7 Ablation Studies

Table 5: Ablation of optional GATE components. "*W/o Merg.*", "*W/o Sf.-Chk.*", and "*W/o Prun.*" denote removal of Tool Merging, Self-Check, and Pruning. "*Top-k Ret.*" indicates vector-based Top-k Retrieval. Pruning is omitted for open-ended tasks completing milestones early. More results are in Appendix C.

DA/Mthds	*W/o Merg.*	*W/o Sf.-Chk.*	*W/o Prun.*	*Top-k Ret.*	*GATE*
Wood	6±2 (3/3)	8±4(3/3)	-	**4±1**(3/3)	**4±0**(3/3)
Stone	11±3 (3/3)	11±3(3/3)	-	**7±2**(3/3)	**7±1**(3/3)
Iron	20±4(3/3)	24±8(3/3)	-	25±7(3/3)	**17±3**(3/3)
Diamond	56±9(3/3)	52±0(1/3)	-	74±40(3/3)	**29±2**(3/3)
Textcraft	68.09%	71.36%	69.23%	70.56%	**73.93%**
Date	72.40%	71.11%	78.89%	85.57%	**88.70%**

Self-Check and Tool Merging drive most gains. We ablate four components: GraphRank Retrieval, Tool Merging, Self-Check, and Pruning to evaluate their impact on tool-graph performance. Open-ended tasks use GPT-4o, while close-ended tasks

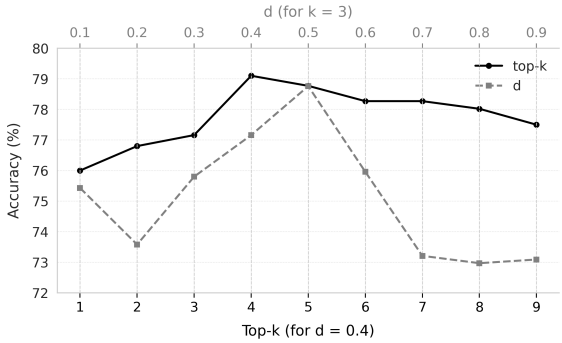

Figure 8: Ablation study of top-k and d on MATH: The accuracy changes with varying k (bottom axis, for fixed $d = 0.4$) and d (top axis, for fixed $k = 3$).

are trained with GPT-4o and tested on *Qwen2.5-14B-Instruct*. We also compare with vector-based Top-k retrieval to isolate graph connectivity effects (Table 2). Among these components, Self-Check and Tool Merging contribute most. Removing Self-Check causes a 16.3% accuracy drop on Date and slower Minecraft progression (Table 5), confirming its role in validating tool construction. Tool Merging improves efficiency by reducing redundancy; without it, task accuracy and graph quality decline, with milestone times up to sixfold longer.

GraphRank improves tool evolution through structured retrieval. GraphRank Retrieval accelerates tool evolution by modeling dependencies for more accurate selection. As shown in Figure 8, accuracy on MATH remains steady between 77% and 79% as Top-k increases, indicating robustness to retrieval width. In contrast, varying the threshold d has a stronger effect: performance peaks around d=0.4 and declines beyond d>0.7, suggesting that moderate connectivity yields the best balance between structural exploration and stability.

8 Conclusion

In this paper, we introduce GATE, a framework that dynamically constructs a hierarchical tool graph through two-agent collaboration. By modeling tool dependencies and integrating GraphRank for retrieval, GATE enables efficient tool discovery, composition, and reuse, addressing challenges in tool construction. Experiments show that GATE outperforms existing methods on open- and closed-ended tasks, achieving higher accuracy and adaptability. Further analysis shows that GATE balances performance and library size while preserving strong generalizability and adaptability. These results establish GATE as a robust solution for autonomous tool-building and advanced agent systems.

9 Limitations

While our framework shows strong adaptability across diverse tasks, future work could examine its applicability to multimodal settings such as GUI agents and large-scale structured code generation. These extensions would deepen the evaluation of generalizability and expose broader capabilities and limitations of LLM-based tool construction. Although our tools are generated and executed in isolation, considerations of tool safety remain a worthwhile topic for future exploration.

References

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammadhossein Bateni, Chen-Yu Lee, and Tomas Pfister. 2024. Re-invoke: Tool invocation rewriting for zero-shot tool retrieval.

Marc-Alexandre Côté, Akos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, and 1 others. 2019. Textworld: A learning environment for text-based games. In *Computer Games: 7th Workshop, CGW 2018, Held in Conjunction with the 27th International Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sweden, July 13, 2018, Revised Selected Papers 7*, pages 41–75. Springer.

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. *Advances in Neural Information Processing Systems*, 35:18343–18362.

Douglas J Futuyma and Gabriel Moreno. 1988. The evolution of ecological specialization. *Annual review of Ecology and Systematics*, pages 207–233.

Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. 2023. Learning interpretable libraries by compressing and documenting code. In *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, and 1 others. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

OpenAI. 2022. Text-embedding-ada-002: New and improved embedding model. Accessed: 2025-02-11.

PrismarineJS. 2013. Prismarinejs/mineflayer: Create minecraft bots with a powerful, stable, and high-level javascript api. `https://github.com/PrismarineJS/mineflayer`. Accessed: 2025-02-15.

PrismarineJS. 2013. Prismarinejs/mineflayer: Create minecraft bots with a powerful, stable, and high-level javascript api. `https://github.com/PrismarineJS/mineflayer`. Accessed: 2025-01-08.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *arXiv preprint arXiv:2305.14318*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024a. From exploration to mastery: Enabling llms to master tools via self-driven interactions.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024b. Towards completeness-oriented tool retrieval for large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 1930–1940. ACM.

Toran Bruce Richards. Significant-gravitas/autogpt: An experimental open-source attempt to make gpt-4 fully autonomous., 2023. *URL https://github.com/Significant-Gravitas/AutoGPT*.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. 2025. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. *arXiv preprint arXiv:2503.01763*.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2(5):9.

Kumar Shridhar, Koustuv Sinha, Andrew Cohen, Tianlu Wang, Ping Yu, Ram Pasunuru, Mrinmaya Sachan, Jason Weston, and Asli Celikyilmaz. 2023. The art of llm refinement: Ask, refine, and trust. *arXiv preprint arXiv:2311.07961*.

Temple F Smith, Michael S Waterman, and 1 others. 1981. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, and 1 others. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. 2024. Regal: Refactoring programs to discover generalizable abstractions. *arXiv preprint arXiv:2401.16467*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. 2024a. Toolgen: Unified tool retrieval and calling via generation. *arXiv preprint arXiv:2410.03439*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025a. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, and 1 others. 2025b. Epicoder: Encompassing diversity and complexity in code generation. *arXiv preprint arXiv:2501.04694*.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024c. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis N. Ioannidis, Karthik Subbian, Jure Leskovec, and James Zou. 2024. Avatar: Optimizing llm agents for tool usage via contrastive reasoning.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25497–25506.

Wenpu Xing and Ali Ghorbani. 2004. Weighted pagerank algorithm. In *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.*, pages 305–314. IEEE.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. 2023. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*.

Yuanhang Zheng, Peng Li, Wei Liu, Yang Liu, Jian Luan, and Bin Wang. 2024. Toolrerank: Adaptive and hierarchy-aware reranking for tool retrieval.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and 1 others. 2023. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*.

A Experimental Details

A.1 Open-ended Task

Benchmark We employed the benchmark proposed by Voyager (Wang et al., 2023), using Minecraft as the experimental platform. Minecraft provides a sandbox environment where players gather resources and craft tools to achieve various goals. The simulation is built on MineDojo (Fan et al., 2022) and uses Mineflayer (PrismarineJS, 2013) JavaScript APIs for motor control.

Baselines We conducted a comprehensive comparison with four baselines.

- **ReAct:** ReAct (Yao et al., 2022) uses chain-of-thought prompting [46] by generating both reasoning traces and action plans with LLMs. We provide it with our environment feedback and the agent states as observations.

- **Reflexion:** Reflexion (Shinn et al., 2023) is built on top of ReAct (Yao et al., 2022)with self-reflection to infer more intuitive future actions.

- **AutoGPT:** AutoGPT (Richards) is a popular software tool that automates NLP tasks by decomposing a high-level goal into multiple subgoals and executing them in a ReAct-style loop. We re-implement AutoGPT by using GPT-4O to do task decomposition and provide it with the agent states, environment feedback, and execution errors as observations for subgoal execution We provide it with execution errors and our self-verification module.

- **Voyager:** Voyager (Wang et al., 2023) is a system that integrates an automated curriculum, a scalable skill library, and an iterative prompting framework based on environmental feedback to explore, store, and accumulate skill library within the Minecraft environment.

Metric The evaluation metric is based on the number of iterations required to progress through tool upgrades, from wooden to stone, iron, and finally diamond tools. Each execution of code is considered one iteration.

Model We leverage GPT-4o for text completion, along with the text-embedding-ada-002 API for text embedding. We set all temperatures to 0 except for the automatic curriculum, which uses temperature = 0.1 to encourage task diversity.

Setting We set the maximum number of iterations to 160. For both GATE and Voyager, all agents are controlled by GPT-4o, with the number of tools retrieved per iteration set to 5. To ensure a fairer comparison, we removed the Tool Requirement Stage and bug-free checks in GATE , and allowed a maximum of 3 self-checks per iteration.

Item Types and Levels In the Minecraft task, there are different types and levels of items. Diamond tools are the highest level, and rare items such as golden apples also exist. High-level tools require some lower-level items to craft. Table 6 lists the key items in the Minecraft task.

A.2 Close-Ended Tasks

A.2.1 Agent Task

Benchmark We conducted experiments on two types of agent tasks, demonstrating GATE's capabilities in both game-related and data science tasks.

- **TextCraft:** We evaluate GATE on the TextCraft dataset (Futuyma and Moreno, 1988), which challenges agents to craft Minecraft items in a text-only environment (Côté et al., 2019). Each task instance provides a goal and a sequence of crafting commands, which include distractors. We use depth-2 splits for testing and reserve a subset of depth-1 recipes for development, resulting in a 99/77 train/test split.

- **InfiAgent-DABench:** We also test GATE on the InfiAgent-DABench benchmark (Hu et al., 2024), which evaluates LLM-based agents on end-to-end data analysis tasks. This benchmark consists of 257 questions across 52 CSV files, with each question corresponding to a unique CSV file. Agents are required to generate code to analyze data and produce the specified output format. We randomly selected 20 CSV files and their associated question-answer pairs as training data, resulting in a train/test split of 98/159 instances.

Baselines We compare GATE with three methods described below.

- **ReAct:** In this setting, we employ the executor to interact iteratively with the environment, adopting the think-act-observe prompting style from ReAct (Yao et al., 2022).

Category	level	Items
Tools	Wooden Tools	Wooden_Shovel,Wooden_Pickaxe,Wooden_Axe,Wooden_Hoe,Wooden_Sword
	Stone Tools	stone_pickaxe, stone_shovel,Stone_Axe,Stone_Hoe,Stone_Sword
	Iron Tools	iron_pickaxe, iron_axe, iron_sword, iron_shovel, iron_hoe
	Diamond Tools	diamond_pickaxe, diamond_sword, diamond_axe, diamond_shovel
Armor	Iron Armor	iron_chestplate, iron_helmet, iron_leggings
	Diamond Armor	diamond_chestplate, diamond_helmet, diamond_leggings, diamond_boots
Food	Raw Food	chicken, mutton, porkchop, rabbit, raw_rabbit, spider_eye, bone
	Cooked Food	cooked_beef, cooked_chicken, cooked_mutton, cooked_porkchop, cooked rabbit
	Advanced Food	golden apple

Table 6: List of item types and levels in the Minecraft task.

- **Plan-Execution:** In contrast, the Plan-and-Execute approach (Shridhar et al., 2023; Yang et al., 2023) generates a plan upfront and assigns each sub-task to the executor. To ensure each step is executable without further decomposition, we provide new prompts with more detailed planning instructions.

- **Reflexion:** In the Reflection setting (Shinn et al., 2023), the agent engages in self-reflection after each step, drawing on environmental feedback and exploration history.

Metric The most practically important aspect of the solutions is correctness. For Textcraft, we verify whether the agent's inventory contains the goal item. For DABench, we check if the agent's final answer matches the ground truth.

Model During training, we use GPT-4o to construct the tool library with a temperature setting of 0. In the testing phase, we conduct a comprehensive comparison of various open-source and closed-source models. The open-source models include *Qwen2.5-7B-Instruct, Qwen-Coder-7B-Instruct, Qwen2.5-14B-Instruct, Deepseeker-Coder-6.7B-Instruct, and Deepseeker-Coder-33B-Instruct*, while the closed-source models primarily include *gpt-3.5-turbo-1106* and *Claude-3-haiku*. During testing, the temperature is set to 0.3, and each experiment is repeated 3 times, with the average result reported.

Setting For ReAct, Reflexion, and GATE , the maximum number of steps is set to 20. For Plan-Execution, the maximum number of steps for each sub-task is set to 8. In GATE , the number of tools retrieved during testing is limited to 3.

A.2.2 Single-turn Code Task

Benchmark To further explore GATE's potential, we evaluated it on single-turn code generation tasks spanning mathematical reasoning, date comprehension, and tabular reasoning:

- **MATH:** We used a subset of the MATH dataset (Hendrycks et al., 2021), focusing on 405 level-4 and level-5 algebra problems (MATH contains 5 levels of difficulty) that require textual understanding and advanced reasoning. We randomly selected 200 examples from the test set of the MATH dataset to construct the tool network, resulting in a train/test split of 200/405.

- **Date:** We use the date understanding task from BigBenchHard (Srivastava et al., 2022), which consists of short word problems requiring date understanding. We follow the data splits provided by REGAL(Stengel-Eskin et al., 2024), resulting in a train/test split of 66/180.

- **TabMWP:** We further extend our general experiments on MATH by testing on TabMWP (Grand et al., 2023), a tabular reasoning dataset consisting of math word problems about tabular data. Based on the CRAFT (Yuan et al., 2023) splits, we selected 470 problems from levels 7 and 8 (TabMWP contains 8 levels) from the 1,000 test examples. Additionally, we randomly selected 200 examples from the TabMWP training set, resulting in a train/test split of 200/470.

Baselines For these tasks, we use Programs of Thoughts (PoT) (Chen et al., 2022) and other tool-making methods as baselines for comparison.

- **PoT:** LLM utilizes a program to reason through the problem step by step (Chen et al., 2022).

- **LATM:** LATM (Cai et al., 2023) samples 3 examples from the training set and create a tool for the task, which is further verified by 3 samples from the validation set. The created tool is then applied to all test cases.

- **CREATOR:** CREATOR (Qian et al., 2023) disentangle planning (tool making) from execution, enabling Large Language Models (LLMs) to autonomously create a specific tool for each test case during inference.

- **CRAFT:** CRAFT (Yuan et al., 2023) constructs task-specific toolsets by generating a tool for each training example. During testing, it utilizes a tool retrieval module and a reasoning process akin to CREATOR, generating a function first and then producing the corresponding invocation code.

- **REGAL:** During training, REGAL (Stengel-Eskin et al., 2024) refines primitive programs by extracting functions. In the testing phase, it retrieves both tools and refactored programs—comprising original and refactored versions—to generate a program that effectively solves the task.

Metric We use correctness as the evaluation metric, measuring whether the execution outcome of the solution program exactly matches the ground-truth answer(s).

Model The models for the single-turn code generation task are the same as those used for the Agent Task, as presented in Section A.2.1.

Setting To ensure a fair comparison, we make slight adjustments to each method. For all methods, we allow up to 3 times for format checking and correction, as small models may not always follow the required output format. For PoT, we use 6 fixed examples of basic tool usage as few-shot. CREATOR employs the rectifying process, while for CRAFT, we use the same training set as our method and construct the tool library with GPT-4o, retrieving 3 tools during testing. For Regal, we use PoT along with GPT-4o to obtain ground-truth code, select the correct program, and have GPT-4o reconstruct it. To maintain fairness in tool generation quality, we standardize the few-shot examples of basic tools and retrieve 3 tools, along with 3 usage examples from the current tool library, avoiding errors from pruned tools. For our method, we train with GPT-4o, retrieving 3 tools and their corresponding usage examples during testing, while fixing the basic tool few-shot examples to 3, ensuring consistency with PoT's total few-shot count.

B More Results

B.1 Open-ended Task

B.1.1 More complex tools

Our hierarchical graph architecture offers significant advantages in handling complex tasks and large-scale systems. As shown in Figure 13, Trial 1 starts with five nodes occupying three layers, and evolves into a five-layer network, with an increasing number of inter-tool calls. As shown in Figure 14, Trial 2 starts with four nodes occupying four layers, and evolves into a five-layer network with more inter-tool calls. As shown in Figure 15, Trial 3 starts with four nodes occupying three layers, and evolves into a six-layer network structure, with a growing number of inter-tool calls. Our tool graph becomes progressively more complex, flexibly expanding and optimizing its components. These results demonstrate that our method can generate tools that call each other, and combine them into more complex tools. This not only enhances scalability but also facilitates the creation of more sophisticated tools, enabling the solution of increasingly complex problems.

B.1.2 More types of inventory

Our method is able to generate more inventory types than Voyager. As shown in Table 7, we can see that GATE produces more inventory types in all three trials compared to Voyager.

The inventory collected by GATE in each trial is

- **Trial 1:** *oak_log, birch_log, oak_planks, birch_planks, crafting_table, stick, wooden_pickaxe, dirt, cobblestone, coal, stone_pickaxe, raw_copper, furnace, copper_ingot, andesite, raw_iron, granite, iron_ingot, iron_pickaxe, shield, diorite, raw_gold, lapis_lazuli, redstone, diamond, diamond_pickaxe, bucket, gold_ingot, iron_chestplate, arrow, iron_sword, iron_helmet, diamond_sword, diamond_helmet, lightning_rod, chest,*

iron_axe, iron_leggings, sandstone, dandelion, spider_eye, string, iron_shovel, copper_block, iron_door, iron_hoe, kelp, bow, dried_kelp, torch, cooked_beef, gray_wool, cobbled_deepslate, tuff, diamond_leggings, bone, diamond_chestplate, chicken, white_banner, cooked_chicken, egg, feather, oak_sapling, apple, acacia_log, golden_apple, diamond_axe

- **Trial 2:** *oak_sapling, oak_log, stick, oak_planks, crafting_table, wooden_pickaxe, dirt, cobblestone, stone_pickaxe, diorite, raw_iron, coal, lapis_lazuli, gravel, furnace, iron_ingot, raw_copper, sandstone, granite, iron_pickaxe, andesite, raw_gold, gold_ingot, diamond, diamond_pickaxe, redstone, cobbled_deepslate, bucket, iron_sword, arrow, bow, bone, birch_log, chest, amethyst_block, calcite, smooth_basalt, iron_chestplate, diamond_sword, diamond_helmet, iron_leggings, diamond_boots, water_bucket, string, orange_tulip, mutton, white_wool, porkchop, dandelion, cooked_porkchop, cooked_mutton*

- **Trial 3:** *jungle_log, stick, oak_sapling, jungle_planks, crafting_table, dirt, wooden_pickaxe, cobblestone, stone_pickaxe, raw_iron, raw_copper, furnace, iron_ingot, iron_pickaxe, coal, diorite, lapis_lazuli, andesite, moss_block, clay_ball, redstone, raw_gold, cobbled_deepslate, granite, diamond, diamond_pickaxe, copper_ingot, gunpowder, bucket, gravel, gold_ingot, oak_log, iron_sword, iron_chestplate, chest, diamond_sword, spruce_sapling, rotten_flesh, bone, rose_bush, water_bucket, string, oak_planks, grass_block, diamond_helmet, iron_leggings, emerald, snowball, rabbit_hide, rabbit, spruce_log, cooked_rabbit, diamond_boots*

The inventory collected by Voyager in each trial is

- **Trial 1:** *oak_log, birch_log, oak_sapling, birch_sapling, oak_planks, stick, crafting_table, wooden_pickaxe, dirt, cobblestone, stone_pickaxe, raw_copper, white_tulip, coal, furnace, copper_ingot, granite, raw_iron, iron_ingot, lightning_rod, iron_pickaxe, pink_tulip, orange_tulip, sandstone, shears, shield, diorite, cobbled_deepslate, iron_block, chest, tuff, lapis_lazuli, redstone, diamond, raw_gold, gold_ingot, diamond_pickaxe, diamond_helmet, diamond_sword, sand, andesite, arrow, bone, iron_chestplate, beef, leather, oak_leaves, porkchop, cooked_beef, leather_leggings*

- **Trial 2:** *dirt, oak_log, oak_planks, crafting_table, stick, oak_sapling, wooden_pickaxe, cobblestone, coal, stone_pickaxe, raw_iron, granite, lapis_lazuli, raw_copper, furnace, iron_ingot, copper_ingot, iron_helmet, iron_pickaxe, diorite, andesite, salmon, ink_sac, iron_chestplate, lightning_rod, cooked_salmon, stone, stonecutter, rotten_flesh, gravel, flint, chest, iron_leggings, copper_block, cobbled_deepslate, tuff, diamond, diamond_pickaxe, raw_gold, gold_ingot, redstone, diamond_sword, egg, diamond_boots, diamond_axe*

- **Trial 3:** *jungle_log, jungle_planks, oak_sapling, oak_log, crafting_table, stick, wooden_pickaxe, dirt, cobblestone, coal, stone_pickaxe, raw_copper, furnace, copper_ingot, magma_block, lightning_rod, stone_axe, jungle_boat, kelp, sand, sandstone, glass, raw_iron, granite, lapis_lazuli, diorite, iron_ingot, bucket, iron_pickaxe, chest, andesite, redstone, dried_kelp, iron_chestplate, wooden_sword, shield, iron_sword*

Method	Trial 1	Trial 2	Trial 3
Voyager	50	45	37
GATE(Ours)	67	51	53

Table 7: Number of different inventory types produced by each trial

B.2 Longer Exploration Path

To better demonstrate the exploration capabilities of the agent, we compared the exploration trajectories and their lengths. As shown in Figure 10, our agent exhibits longer and more persistent exploration capabilities than Voyager. In Table 8, the trajectory lengths of our agent are consistently much greater than those of Voyager. GATEis able to traverse across multiple terrains, with an average distance 2.66 times longer than Voyager. Additionally, GATE can explore across different continental plates, while Voyager remains confined to a sin-

gle plate, highlighting the exceptional exploration capability of GATE.

Method	Trial 1	Trial 2	Trial 3	*Avg*
Voyager	1925.74	4102.99	902.13	2310.29
GATE(Ours)	5665.75	8908.57	3895.06	6156.46
Performance Gain	2.94	2.17	4.32	2.66

Table 8: Exploration trajectory length in each trial, where *Performance Gain* = *ours*/*voyager*.

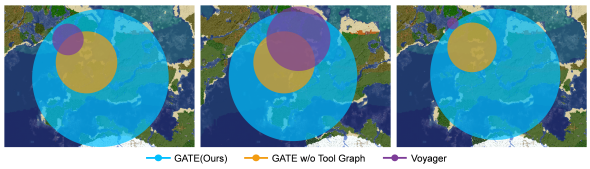

Figure 9: Map coverage: Three bird's eye views of Minecraft maps. The trajectories are plotted based on the position coordinates where each agent interacts.

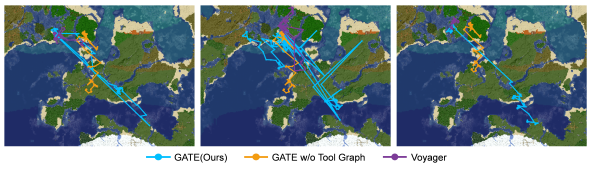

Figure 10: Movement trajectory Map: Three bird's eye views of Minecraft maps. The trajectories are plotted based on the position coordinates where each agent interacts.

B.2.1 Efficient Zero-Shot Generalization Enabled by the Tool Graph

Efficient Zero-Shot Generalization to Unseen Tasks Based on the results presented in Table 9 and Figure 11, we can clearly observe the significant advantages of GATE in the open-ended task. Table 9 shows the number of iterations required for different methods to complete various tasks (Gold Sword, Compass, Diamond Hoe, Lava Bucket), where fewer iterations indicate higher efficiency. Compared to Voyager and GATE (w/o toolnet), GATE consistently requires significantly fewer iterations across all tasks, demonstrating substantial improvements in efficiency. Notably, in the Gold Sword task, GATE (ours) completes the task in just 14.00±1.73 iterations, whereas Voyager requires 46.33±14.57 iterations, showcasing its superior performance.

Figure 11 further visualizes the intermediate progress of different methods on the "Craft a Compass" and "Craft a Diamond Hoe" tasks. It is evident that GATE learns and masters the necessary skills for crafting items more quickly. As the number of prompting iterations increases, GATE reaches the task objectives significantly earlier than the other methods. Additionally, while GATE(w/o Tool Graph) performs better than Voyager, it still lags behind GATE, indicating that the ToolNet component plays a crucial role in enhancing the model's capability.

Overall, these experimental results demonstrate that GATE not only learns new skills and crafting techniques more efficiently but also that its key module, Tool Graph, is essential for overall performance improvement. This further validates the effectiveness of our approach in self-driven exploration and task generalization.

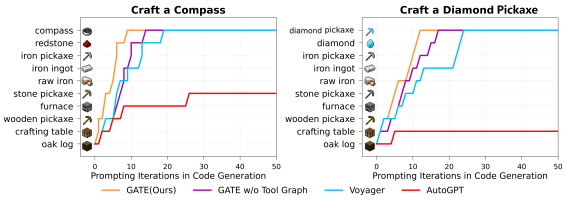

Figure 11: Zero-shot generalization to unseen tasks. Here, we visualize the intermediate progress of each method on the tasks "Craft a Compass" and "Craft a Diamond Hoe."

B.3 Close-Ended Tasks

B.3.1 Agent Task

Figures 16 and 17 present the tool network evolution diagrams of DA-Bench and TextCraft, which visually reflect the call relationships between different tool functions. In these diagrams, each node represents a specific tool function, edges indicate the call dependencies between tools, and the shading of the nodes reflects the frequency of tool calls—darker colors indicate higher call frequency. From Figure 16, it can be observed that in DA-Bench, the tool network expands progressively as the task advances, forming multiple core nodes with higher call frequencies. This suggests that certain key tools are frequently called during the task execution, playing a central role. Additionally, the tool call relationships exhibit a hierarchical and well-organized structure, reflecting DA-Bench's efficiency in tool dependency management.

In contrast, Figure 17 illustrates the tool network evolution of TextCraft, which also shows a similar expansion trend overall. However, compared to DA-Bench, the tool call frequency in TextCraft is more evenly distributed across multiple nodes,

Method	Trial	Gold Sword	Compass	Diamond Pickaxe	Lava Bucket
Voyager	Trial 1	48	16	24	N/A
	Trial 2	31	17	25	39
	Trial 3	60	20	18	N/A
	Average	46.33±14.57	17.67±2.08	22.33±3.79	39.00±0.00
GATE*(w/o toolnet)*	Trial 1	26	27	23	N/A
	Trial 2	18	22	18	N/A
	Trial 3	56	15	30	N/A
	Average	33.33±20.03	21.33±6.03	23.67±6.03	N/A±N/A
GATE*(ours)*	Trial 1	13	28	16	19
	Trial 2	13	10	14	27
	Trial 3	16	13	13	18
	Average	**14.00±1.73**	**17.00±9.64**	**14.33±1.53**	**21.33±4.93**

Table 9: The mastery of the tech tree in the Open-ended Task. The number indicates the number of iterations. The fewer the iterations, the more efficient the method. "N/A" indicates that the number of iterations for obtaining the current type of tool is not available.

meaning that the system calls a wider variety of tools during task execution, rather than relying on a few core tools. This distribution pattern may suggest that TextCraft adopts a more diverse tool usage strategy in task execution.

A comparative analysis of the two figures reveals that, although both DA-Bench and TextCraft exhibit certain hierarchical and expansive characteristics in their tool call patterns, DA-Bench relies more heavily on a few core tools, whereas TextCraft displays a more dispersed tool call pattern. This contrast not only highlights the differences in tool usage between the two, but also emphasizes the importance and effectiveness of ToolNet.

B.3.2 Single-turn Code Task

As shown in the Figure18 19, this illustrates the evolution of the tool graph for the Math and TabMWP tasks. It is evident that the tool graph gradually becomes more complex, creating multiple layers of tools, making the tool graph more intricate. Since the Date task can be solved with fewer tools, there is no evolution of the tool graph. However, the generated tools can still effectively solve the task, while there exists a multi-level calling relationship.

C More Ablations

C.1 Open-ended Task

As shown in Figure 12, GATEsignificantly outperforms methods that lack certain functional modules in discovering new Minecraft items and skills. It can be observed that the performance is worst when "w/o retrieval" is used, indicating that the absence of retrieval has the greatest impact on overall functionality and plays a crucial role, thereby validating the effectiveness of our retrieval method. The performance with "w/o duplication" is slightly better, indicating its importance is weaker than that of "w/o retrieval." The performance of "w/o check" and "w/o pruning" is better, but still far behind GATE, which further demonstrates the importance and effectiveness of each functional component.

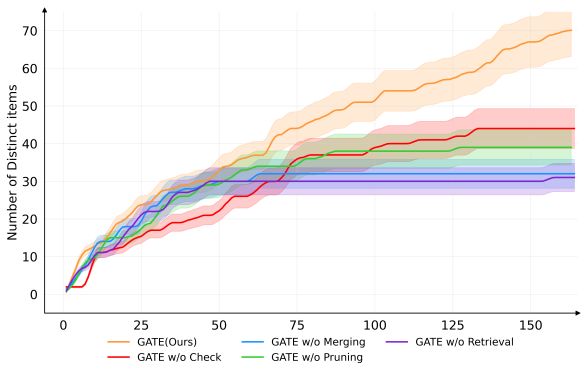

Figure 12: Ablation study of the iterative prompting mechanism. AETN surpasses all other options, highlighting the essential significance of each functional module in the iterative prompting mechanism.

C.2 Closed-Ended Task

For the Closed-Ended Task, we select Textcraft from the Agent Task and Date from the Single-turn Code Task to evaluate the effectiveness of several components in our method. The results are shown in the Table 10.

Method	TextCraft	Date
W/o Self-Check	42	9
W/o Merging	49	11
W/o pruning	46	9
GATE	44	4

Table 10: The number of tools in Close-Ended Task.

D Tool Making

D.1 Basic Tools

As shown in the Table 11 , the basic tools generated by each method are displayed.

D.2 Tool construction Lists

CREATOR:

- **MATH:** *sum of areas, find largest won matches, find K, total distance after bounces, find common ratio sum, count lattice points with distance squared, find c for radius, find circle equation and constants, polynomial degree product, calculate cells, find fiftieth term, find non domain values, inverse function product, find m and n, sum of fractions from roots, find roots of quadratic, main, find coefficients, compute expression, prime factors, find x y, find second largest angle, find y coordinate, find constants, evaluate expression, find b for one solution, find c, find minimum value, find possible s, solve expression, find cone height, solve abc, find minimum expression, . . . , time to hit ground, sum of reciprocals of roots, solve x floor x product, sum of possible x, find constant a, sum of squares of solutions, find cost per extra hour, is triangular number, find smallest b greater than 2011, solve exponential equation, solve club suit equation, find degree of h, f, find vertical asymptotes, domain width, maximize revenue, future value, total savings, find min interest rate, equation, find integers, sum of x coordinates squared, find integer values of a, smallest c for real domain, smallest integer c, find m, required investment, simplify expression, g, distance between midpoints, compute x and power, greatest possible a, find continued fraction value, find a b, solve mnp, compute sum, sum of integers in range,*
- **Date:** *get us thanksgiving date, get date one week from first monday of 2019, calculate anniversary date, calculate yesterday from last day of january, calculate one week ago from first monday, get first monday of 2019, calculate yesterday, calculate yesterday from rescheduled meeting, calculate date a month ago from rescheduled meeting, calculate yesterday from first monday of 2019, get date 10 days before us thanksgiving, calculate one week ago from egg runout, calculate one week ago from end of first quarter, calculate date 24 hours later, calculate date a month ago, calculate date 24 hours after anniversary, calculate one week from today from rescheduled meeting, . . . , get tomorrow from us thanksgiving, calculate yesterday from day before yesterday, calculate yesterday from anniversary, calculate date 10 days ago, calculate one year ago from egg run out date, calculate tomorrow from yesterday, calculate one week from last day of january, calculate one week from anniversary, calculate yesterday from eggs run out, calculate tomorrow from today, calculate tomorrow from day before yesterday, calculate one week ago from today, calculate one week ago, calculate date one month ago from anniversary, calculate one year ago from given date, calculate one week from given date*
- **TabMWP:** *calculate total cost, smallest points, price difference, cost of river rafts, calculate median, calculate range, calculate total spent, rate of change, cost difference, cost for rides, rate of change vacation days, total participants, calculate mean glasses, find mode of states visited, rate of change straight A students, calculate median basketball hoops, count bins with toys in range, people with at least 3 trips, count teams with fewer than 80 swimmers, calculate median clubs, count exact pushups, children with less than 2 necklaces, people played exactly 3 times, count people with fewer than 80 pullups, range of states visited, find spent amount, . . . , calculate median miles, people with fewer than 3 seashells, calculate median glasses, cost to buy cockatiels, largest broken lights, calculate spent, calculate ice cream cost, range of soccer fields, patrons with at least 2 books, count bushes with 20 roses, total people played golf, range of articles, count shipments with exactly 60 broken plates, total cost for lip balms, rate of change scholarships, count teams with fewer than 50 members, count tests with 34 problems, find mode of soccer fields, rate of change hockey games, find lowest score, count pizzas with exactly 48 pepperoni, count peo-*

Tasks	Basic Tools
Other Tasks	ToolRequest, NotebookBlock, Terminate, CreateTool, EditTool, Python, Feedback, SendAPI, Feedback, Retrieval
Minecraft	smeltItem, killMob, waitForMobRemoved, givePlacedItemBack, useChest, exploreUntil, craftItem, mineBlock, shoot, placeItem, craftHelper, smeltItem, mineflayer, killMob, useChest, exploreUntil, craftItem, mineBlock, placeItem

Table 11: Basic tools in various methods.

ple with at least 30 points, cost of wooden benches, rate of change students, patients with fewer than 2 trips, find mode, total cost for hazelnuts, calculate mean fan letters, readers with at least 4 hats, count classrooms with 41 desks

CRAFT:

- **MATH:** *find pack size, count distinct solutions, calculate points, find tank capacity, solve exponential log equation, total energy equilateral triangle, inverse square law force, find max value, total logs in stack, sum of multiples of 13, calculate exponential growth, gravitational force, find x for piecewise composition, positive difference, specific piecewise func, day exceeds 200 cents, find lattice points, count integer parameters for integer solutions, count zeros in square of power of ten minus one, energy stored, sum of squares of roots, sum odd integers, find d minus e squared, compute complex series sum, total energy configuration, sum of areas, . . . , max item price, solve two variable system, inverse variation power, total distance hopped, is prime, total distance, find constant term of polynomial, total distance moved, find perpendicular slope, calculate inverse proportionality, find value of A, count integer a, find min items for higher score, apply r n times, find min x, day exceeds threshold, calculate area in square yards, solve log equation, total items produced, find variable for distance condition, solve time at speeds, find largest solution, find weight of object, calculate proportional value, calculate material cost, solve for variable, total elements in arithmetic sequence, transformed domain, find day for algae coverage, calculate energy stored, least value of y, solve bowling ball weight, find min froods*
- **Date:** *get today date, calculate one week ago, calculate n days from future date, calculate n days from date in format, calculate date days ago, calculate n months from date, calculate one week from today, calculate date after event, find palindrome day, calculate date a month ago, calculate date after days and months, calculate relative date, calculate n days from reference, calculate one year ago from today, calculate n hours from date, calculate date n days from, get date today, calculate date 10 days ago from deadline, calculate n weeks from date, . . . , calculate n units from date, calculate n years from date, calculate n weeks from first weekday of year, calculate today from tomorrow, find special day, calculate date 10 days ago from future, calculate n days after event, calculate date from days passed, calculate one week from christmas eve, calculate one year ago, calculate date 24 hours later, calculate n weeks from anniversary, calculate tomorrow from uk format date, calculate n days from date, is palindrome, calculate one week from first monday of year, calculate one week ago from anniversary*
- **TabMWP:** *get frequency, calculate volleyballs in lockers, calculate total cost from package prices, calculate total items from group counts, calculate mode, calculate donation difference for person, count bags with 20 to 40 broken cookies, calculate total items from groups and items per group, count commutes of 50 minutes, get received amount, calculate total items for groups, find probability, calculate vacation cost, calculate rate of change, find received amount for transaction, calculate vote difference between two items for group, count customers, find minimum value in stem leaf, calculate metric wrenches, find smallest number, count books with 30 to 50 characters, . . . , count people with 67 pullups, calculate difference in donations for person, calculate total cost from unit price and weight, calculate total items from ratio, calculate total cost from unit weight prices and weight, calculate donation difference between causes, cal-*

culate difference, calculate net income, calculate grasshoppers on twigs, count total members in group, calculate expenses on date, find lightest child, calculate difference in amounts, count votes for item from groups, calculate probability from count table, get table cell value, calculate jeans in hampers, count instances with specific value in stem leaf, calculate donation difference for person and causes, calculate total from frequency and additional count, calculate range, calculate total reviews

REGAL:

- **MATH:** *solve for largest side, apply function sequence, solve rational equation, calculate expression sum, max sum of products, find b for perpendicular bisector, vertex of quadratic, calculate work days, calculate c for zero coefficient, simplify and rationalize sympy, find a for binomial square, compound interest, calculate inverse variation, expand expression, calculate average speed, calculate rs, sum sequence, solve for p, max consecutive integers, find x intercept, day exceeding threshold, find smallest sum, solve for ac pair, constant function, sum of distances, evaluate expression, sum finite geometric series, factor expression, find common difference, total coins pirates, calculate geometric first term, calculate closest whole number, calculate x minus y squared, solve letter values, find circle center v2, evaluate expression with sqrt, calculate sum of equations, . . . , calculate x3 plus y3, find negative intervals, calculate floor and abs, solve quadratic and find min, calculate y, solve for a, check equations, rationalize and simplify, calculate xyz, calculate distance, solve for x in simplified equation, calculate expression, calculate exponent, sum arithmetic series, complete square form, calculate x2 plus y2*
- **Date:** *subtract weeks from date, add weeks to date, format date, add days to date, subtract months from date, subtract days from date, subtract years from date, calculate date, calculate days between weekdays*
- **TabMWP:** *count range, find mode, total participants, count bushes with fewer roses, find max frequency, total items, count in range, calculate total items, count below threshold, count teams with minimum size, calculate total, calculate range, calculate fraction, sum frequencies below threshold, sum frequencies, calculate difference, calculate median, total outcomes, count specific height, count numbers in range, difference between groups, access frequency, calculate proportionality constant, count values below threshold, find median, calculate probability, calculate mode, get frequency, convert stem leaf to numbers, find minimum, get total items, count scores above, rate of change, calculate mean*

D.3 The tool graph evolution diagrams of GATE for various tasks.

Below are the tool graph evolution diagrams for various tasks. The Date task does not have a tool network evolution diagram, as date reasoning does not heavily rely on tool diversity.

Figure 13: The tool graph evolution diagram for Minecraft Trial 1. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked. The network consists of a total of 6 layers, with layers 2 to 6 shown here from top to bottom.

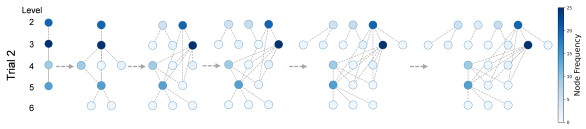

Figure 14: The tool graph evolution diagram for Minecraft Trial 2. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked. The network consists of a total of 6 layers, with layers 2 to 6 shown here from top to bottom.

Figure 15: The tool graph evolution diagram for Minecraft Trial 3. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked. The network consists of a total of 6 layers, with layers 2 to 7 shown here from top to bottom.

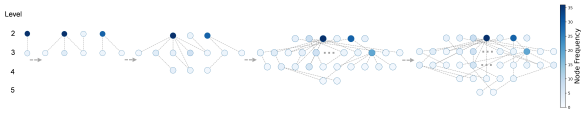

Figure 16: The tool graph evolution diagram of DA-Bench. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked.

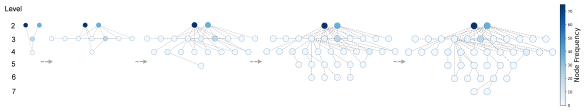

Figure 17: The tool graph evolution diagram of TextCraft. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked.

Figure 18: The tool graph evolution diagram of MATH. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked.

Figure 19: The tool graph evolution diagram of TabMWP. In this diagram, each node represents a tool function, and the edges represent the invocation relationships between tools. The darker the color, the more frequently the tool is invoked.

E Prompt Template

In this section, we provide the prompt templates of different types used throughout our experiment. These prompts were carefully crafted to ensure that the model's output aligns with the specific objectives of each task.

E.1 Construction Stage

In open-ended task online training, we made slight modifications to their prompts based on Voyager (Wang et al., 2023). For close-ended tasks, the prompts used during the construction process are as follows:

Task Solver's Prompt

```
# Instruction
You are the **Task Solver** in a collaborative team, specializing in reasoning and Python
↪  programming.
Your role: analyze tasks, work with the Tool Manager, and solve problems step by step.
Direct solving without tool analysis is not allowed. Always request needed tools first.

# Workflow
- **Stage 1. Tool Requests**: Always start with a tool request. Ask for generalized, reusable
↪  tools (not task-specific).
- **Stage 2. Code & Interact**: Write and run notebook cells incrementally. Avoid big
↪  bundles—adapt based on feedback.
- **Stage 3. Validate & Conclude**: Review, validate, and finalize the solution.

# Custom Library
===api===

# ACTION SPACE #
You should Only take One action below in one RESPONSE:
## CallTools Action
* Signature:
CallTools():
```python
executable python script
```
* Description: The CallTools action allows you to create and execute a Jupyter Notebook cell.
↪ The action will add a code block to the notebook with the content wrapped inside the paired
↪ ``` symbols. If the block already exists, it can be overwritten based on the specified
↪ conditions (e.g., execution errors). Once added or replaced, the block will be executed
↪ immediately.
* Restrictions: Only one notebook block can be managed or executed per action.

Tool_request Action
* Signature:
{
    "action_name": "tool_request",
    "argument": {
         "request": [
             ...
         ]
    }
}
* Description: The Tool Request Action allows you to send tool requirements to the Tool Manager
↪ and request it to create appropriate tools. You need to provide the action in a JSON
↪ format, where the argument field contains a request parameter that accepts a list. Each
↪ element in the list is a string describing the desired tool.

Terminate Action
* Signature: Terminate(result=the result of the task)
* Description: The Terminate action ends the process and provides the task result. The
↪ `result` argument contains the outcome or status of task completion.

Response Format
Each response must contain:
1. **Stage:** (current workflow stage)
2. **Thought:** reasoning and next step
3. **Action:** exactly one action from Action Space

RESPONSE EXAMPLE
Observation: ...(the output of last actions, as provided by the environment and the code
↪ output, you don't need to generate it)

Stage:...(One Stage from `WORKFLOW`)
Thought: ...
Action: ...(Use an action from the ACTION SPACE no more than once per response.)

TASK
===task===
```

Tool Manager's Prompt

```
# Instruction #
You are a Tool Manager in a collaborative team, specializing in assembling existing APIs to
↪  construct hierarchical and reusable abstract tools based on predefined criteria.
You will be provided with a custom library, similar to Python's built-in modules, containing
↪  various functions related to date reasoning. For each task, you will receive:
1. Tool request: The specific goal or functionality the new tool must achieve.
2. Existing tools: A list of available functions from the custom library that you can utilize.
Your task is to analyze the given request and create a reusable tool by effectively leveraging
↪  the relevant functions from the existing tools or utilizing basic tools to achieve the
↪  desired functionality.
If an existing tool from the provided library already fully satisfies the requirements, simply
↪  return that tool instead of duplicating functionality. Ensure all responses align with
↪  reusability and efficiency principles.

# Custom Library #
===api===

# Creation Criteria #
- **Reusability**: The function could be resued for more complex function.
- **Innovation**: Tools should offer innovation, not merely wrap or replicate existing APIs.
↪  Simply re-calling an API without significant enhancements does not qualify as innovation.
- **Completeness**: The function should handle potential edge cases to ensure completeness.
- **Leveraging Existing Functions**: The function should effectively utilize existing
↪  functions to enhance efficiency and avoid redundancy.
- **Functionality**: Ensure the tool runs successfully and is bug-free, guaranteeing full
↪  functionality.

# ACTION SPACE #
You should Only take One action below in one RESPONSE:
## Create tool Action
* Description: The Create Tool action allows you to develop a new tool and temporarily store it
↪  in a private repository accessible only to you. Each invocation creates a single tool at a
↪  time. You can repeatedly use this action to build smaller components, which can later be
↪  assembled into the final tool.
* Signature:
Create_tool(tool_name=The name of the tool you want to create):
```python
The source code of tool
```

Edit tool Action
* Description: The Edit Tool action allows you to modify an existing tool and temporarily store
↪ it in a private repository that only you can access. You must provide the name of the tool
↪ to be updated along with the complete, revised code. Please note that only one tool can be
↪ edited at a time.
* Signature:
Edit_tool(tool_name=The name of the tool you want to create):
```python
The edited source code of tool
```

Response Format
Each response must contain:
1. **Stage:** (current workflow stage)
2. **Thought:** reasoning and next step
3. **Action:** exactly one action from Action Space

TASK
===task===
```

Prompt of Self-Check

```
# Instruction #
You are evaluating whether the tools provided by the Tool Manager meet the required standards.
You follow a defined workflow, take actions from the ACTION SPACE, and apply the evaluation
↪  criteria.

# Evaluation Criteria #
- **Reusability**: The function should be designed for reuse in more complex scenarios. For
↪  instance, in the case of the `craft_wooden_sword()` tool, it would be more versatile if it
↪  could accept a quantity as an input parameter.
- **Innovation**: Tools should offer innovation, not merely wrap or replicate existing APIs.
↪  Simply re-calling an API without significant enhancements does not qualify as innovation.
↪  If an existing tool from the provided library already fully satisfies the requirements,
↪  simply return that tool instead of duplicating functionality. Ensure all responses align
↪  with reusability and efficiency principles.
- **Completeness**: The function should handle potential edge cases to ensure completeness.
- **Leveraging Existing Functions**: Check if any function in "Existing Function" is helpful
↪  for completing the task. If such functions exist but are not invoked in the provided code,
↪  relevant feedback should be given.

# ACTION SPACE #
You should Only take One action below in one RESPONSE:
# Feedback Action
* Signature: {
    "action_name": "Feedback",
    "argument": {
        "feedback": ...
        "passed": true/false
    }
}
* Description: The Feedback Action is represented as a JSON string that provides feedback to
↪  the Tool Manager or SolvingAgent. The feedback field contains comments or suggestions,
↪  while pass indicates whether the tool meets the requirements (true for approval, false for
↪  rejection). Feedback should be concise, constructive, and relevant. If pass is true, the
↪  feedback can be left empty; otherwise, it must be provided.

# RESPONSE FORMAT #
For each task input, your response should contain:
1. One RESPONSE should ONLY contain One Thought and One Action.
2. An comprehensive analysis of the tool code based on the evaluation criteria.(prefix
↪  "Thought: ").
3. An action from the **ACTION SPACE** (prefix "Action: ").

# EXAMPLE RESPONSE #
Observation: ...(output from the last action, provided by the environment and task input, no
↪  need for you to generate it)

Thought: 1. Reusability: ...
2. Innovation: ...
3. Completeness: ...
4. Leveraging Existing Functions: ...

Action: ...(Use an action from the ACTION SPACE once per response.)

# Custom Library #
===api===

# TASK #
===task===
```

F Examples

F.1 Generated Tools

The tools generated for the Open-ended Tasks are as follows:

CraftDiamondHelmet

```
async function craftDiamondHelmet(bot) {
  const mcData = require('minecraft-data')(bot.version);
  const diamond = mcData.itemsByName.diamond;
  const need = 5, have = bot.inventory.count(diamond.id);
  if (have < need) {
    bot.chat(`Need ${need - have} more diamond(s). Mining...`);
    await mineBlock(bot, "diamond_ore", need - have);
  }

  let table = bot.findBlock({ matching: mcData.blocksByName.crafting_table.id, maxDistance: 32
  ↪  });
  if (!table) {
    bot.chat("Placing crafting table...");
    await placeItem(bot, "crafting_table", bot.entity.position.offset(1, 0, 0));
    table = bot.findBlock({ matching: mcData.blocksByName.crafting_table.id, maxDistance: 32
    ↪  });
  }

  bot.chat("Crafting diamond helmet...");
  await CraftWithMaterial(bot, "diamond_helmet", { diamond: 5 });
  bot.chat("Diamond helmet crafted successfully.");
}

async function CraftWithMaterial(bot, name, mats) {
  const mcData = require('minecraft-data')(bot.version);
  const item = mcData.itemsByName[name];
  if (!item) return bot.chat(`Unknown item: ${name}`);

  for (const [mat, amt] of Object.entries(mats)) {
    const m = mcData.itemsByName[mat];
    if (!m) return bot.chat(`Unknown material: ${mat}`);
    const have = bot.inventory.count(m.id);
    if (have < amt) {
      const miss = amt - have;
      bot.chat(`Not enough ${mat}, missing ${miss}`);
      const ore = mcData.blocksByName[mat + "_ore"];
      if (ore && typeof mineBlock === "function") await mineBlock(bot, mat + "_ore", miss);
    }
  }

  let table = bot.findBlock({ matching: mcData.blocksByName.crafting_table.id, maxDistance: 32
  ↪  });
  if (!table && typeof placeItem === "function") {
    await placeItem(bot, "crafting_table", bot.entity.position.offset(1, 0, 0));
    table = bot.findBlock({ matching: mcData.blocksByName.crafting_table.id, maxDistance: 32
    ↪  });
  }

  const recipe = bot.recipesFor(item.id, null, 1, table)[0];
  if (!recipe) return bot.chat(`No recipe for ${name}`);
  try {
    await bot.craft(recipe, 1, table);
    bot.chat(`${name} crafted successfully`);
  } catch (e) {
    bot.chat(`Crafting ${name} failed: ${e.message}`);
  }
}
```

The tools generated for the Agent Tasks are as follows: Here, we can clearly see the call relationships between functions, thus forming more complex tools.

Tools for DA-Bench

```
def filter_rows_by_non_null(df: pd.DataFrame, column_name: str) -> pd.DataFrame:
    """
    Filters rows in a dataset based on non-null values in a specified column.

    Parameters:
    - df (pd.DataFrame): The input DataFrame.
    - column_name (str): The name of the column to filter by non-null values.

    Returns:
    - pd.DataFrame: A DataFrame with rows containing non-null values in the specified column.

    Raises:
    - ValueError: If the specified column is not found in the DataFrame.
    """
    # Check if the column exists in the DataFrame
    if column_name not in df.columns:
        raise ValueError(f"Column '{column_name}' not found in the DataFrame.")

    # Filter rows based on non-null values in the specified column
    filtered_df = df.dropna(subset=[column_name])

    return filtered_df

def convert_column_to_numeric(df: pd.DataFrame, column_name: str) -> pd.DataFrame:
    """
    Converts a specified column in a DataFrame to numeric values, handling non-numeric values
    ↪  appropriately.

    Parameters:
    - df (pd.DataFrame): The input DataFrame.
    - column_name (str): The name of the column to convert to numeric values.

    Returns:
    - pd.DataFrame: The DataFrame with the specified column converted to numeric values.

    Raises:
    - ValueError: If the specified column is not found in the DataFrame.
    """
    # Check if the column exists in the DataFrame
    if column_name not in df.columns:
        raise ValueError(f"Column '{column_name}' not found in the DataFrame.")

    # Convert the specified column to numeric values, setting non-numeric values to NaN
    df[column_name] = pd.to_numeric(df[column_name], errors='coerce')

    # Filter out rows with non-numeric values in the specified column using the existing tool
    df = filter_rows_by_non_null(df, column_name)

    return df
```

Tools for TextCraft

```
def gather_materials_for_dye(required_materials: dict) -> bool:
    """
    Gather all materials needed for crafting a dye.

    Args:
        required_materials (dict): {material: quantity}

    Process:
    - Collect listed materials.
    - If white dye is needed but absent, try crafting it from bone meal or lily of the valley.
    - Verify inventory to ensure nothing is missing.

    Returns:
        bool: True if all materials are ready, else False.
    """
    if not gather_materials(required_materials):
        return False

    inventory = check_inventory()
    if "white dye" in required_materials and "white dye" not in inventory:
        if not gather_materials({"bone meal": 1}) and not gather_materials({"lily of the
        ↪  valley": 1}):
            return False
        if "bone meal" in inventory:
            craft_object("1 white dye", ["1 bone meal"])
        elif "lily of the valley" in inventory:
            craft_object("1 white dye", ["1 lily of the valley"])

    missing_items = check_missing_items([f"{qty} {item}" for item, qty in
    ↪  required_materials.items()])
    if missing_items:
        print(f"Missing items: {missing_items}")
        return False
    return True


def craft_orange_dye(quantity: int) -> bool:
    """
    Craft orange dye.

    Args:
        quantity (int): amount to produce

    Methods:
    - Use orange tulip directly.
    - Or combine red dye + yellow dye.

    Returns:
        bool: True if crafted, else False.
    """
    required = {"orange tulip": quantity, "red dye": quantity, "yellow dye": quantity}
    if not gather_materials_for_dye(required):
        return False

    inventory = check_inventory()
    if "orange tulip" in inventory:
        craft_object(f"{quantity} orange dye", [f"{quantity} orange tulip"])
        print(f"Crafted {quantity} orange dye using orange tulip")
        return True
    if "red dye" in inventory and "yellow dye" in inventory:
        craft_object(f"{quantity} orange dye", [f"{quantity} red dye", f"{quantity} yellow
        ↪  dye"])
        print(f"Crafted {quantity} orange dye using red + yellow dye")
        return True

    print("Failed to craft orange dye.")
    return False
```

The tools generated for the Single-turn Code Task are as follows:

Tools for MATH

```
def find_integer_satisfying_condition(condition):
    """
    Return the smallest positive integer satisfying a condition.

    Args:
        condition (callable): Function that takes an int and returns bool.

    Returns:
        int: The first positive integer that meets the condition.
    """
    x = 1
    while True:
        if condition(x):
            return x
        x += 1


def calculate_min_correct_answers(total_problems, passing_percentage):
    """
    Compute the minimum correct answers needed to pass.

    Args:
        total_problems (int): Total number of problems (> 0).
        passing_percentage (float): Passing threshold (0-100).

    Returns:
        int: Minimum number of correct answers,
             or error message if inputs invalid.
    """
    if total_problems <= 0:
        return "Total number of problems must be greater than zero."
    if not (0 <= passing_percentage <= 100):
        return "Passing percentage must be between 0 and 100."

    required = (passing_percentage / 100) * total_problems
    return find_integer_satisfying_condition(lambda x: x >= required)
```

Tools for Date

```
def calculate_date_by_days_uk_format(start_date_str: str, days_to_add: int) -> str:
    """
    Return date offset by given days in UK format (DD/MM/YYYY).

    Args:
        start_date_str (str): Start date string (DD/MM/YYYY).
        days_to_add (int): Days to add (negative = subtract).

    Returns:
        str: Resulting date in "%m/%d/%Y".

    Raises:
        ValueError: If input does not match UK format.
    """
    from datetime import datetime
    try:
        start_date = datetime.strptime(start_date_str, "%d/%m/%Y")
       return calculate_date_by_days(start_date.strftime("%m/%d/%Y"), days_to_add, "%m/%d/%Y")
    except ValueError as e:
        raise ValueError("Date string does not match DD/MM/YYYY.") from e
```