DynaSaur 🦕: Large Language Agents Beyond Predefined Actions

Anonymous ACL submission

Abstract

Existing LLM agent pipelines typically operate with a fixed set of predefined actions, where an agent selects an appropriate action at each step. While these approaches have proven successful in simple and simulated environments, we argue that they face two major challenges in real-world scenarios: (1) selecting from a fixed set of actions significantly limits the flexibility of LLM agents, and (2) this approach requires substantial human effort to enumerate and implement all possible actions, which is impractical in complex environments with a vast number of potential actions. In this work, we propose a novel LLM agent framework that enables the dynamic creation and composition of actions online, where the agent interacts with the world by implementing and executing programs written in a general-purpose programming language at each step. Generated actions are accumulated over time for future reuse. Our extensive experiments on the GAIA benchmark demonstrate that this framework provides significantly greater flexibility and outperforms previous methods. Specifically, we observe that it allows an LLM agent to recover from scenarios where no relevant actions exist in the action set to solve a given task, or when existing actions fail due to unanticipated edge cases. At the time of writing, we hold the number one position on the GAIA public leaderboard.

1 Introduction

Developing autonomous agents has long been a central goal in AI research. While reinforcement learning has extensively studied this problem and achieved significant success in specific domains (Silver et al., 2016, 2017; Vinyals et al., 2019; Schrittwieser et al., 2020; Wurman et al., 2022), it often falls short in adaptability and generalization within dynamic and uncertain environments. Given the recent advancements of Large Language Models (LLMs) (Chen et al., 2021a; OpenAI, 2023; Bubeck et al., 2023; Anil et al., 2023; Reid et al., 2024) and the vast world knowledge they encapsulate during pretraining, LLMs are considered promising foundations for agent policies capable of solving complex, real-world problems (Schick et al., 2023a; Chen et al., 2023a; Yao et al., 2023b; Deng et al., 2023; Chen et al., 2024a; Zeng et al., 2024). For example, Yao et al. (2023b) synergizes reasoning and acting by interleaving thought and action at every step. Schick et al. (2023a) explores LLM agents that utilize external tools like calculators, search engines, or translation services to enhance their responses to complex question-answering tasks. Deng et al. (2023) proposes a two-stage agent pipeline for web browsing tasks, first filtering out a list of candidate elements and then selecting the appropriate action using an LLM.

Despite these efforts, most existing LLM agent systems operate in simplified environments that accept only a finite and small set of predefined actions. At every decision point, an LLM agent is constrained to select an action from this set, leading to several drawbacks. First, it restricts the agent's flexibility, preventing it from performing actions outside the predefined scope. Second, it requires significant human effort to carefully enumerate and implement all possible actions beforehand; while manageable for closed environments, this approach becomes prohibitively expensive and impractical for real-world settings. Third, in long-horizon tasks, the agent must compose sequences of primitive actions from scratch each time, limiting its ability to learn from past experiences and improve efficiency over time.

In this work, we propose DynaSaur, an LLM agent framework that allows for dynamic action creation to address these limitations. To achieve a universal action representation, we model each action as a Python function. At each step, the agent performs actions by generating Python code snippets that either define new functions when the ex-

isting set is insufficient or reuse existing functions from the current action set. The generated code is executed through a Python interpreter, and the resulting observations are returned to the agent. Furthermore, all actions generated by the agent are accumulated, building a library of reusable functions for future use. This approach enables the agent to extend its capabilities on-the-fly and compose complex actions from simpler ones, enhancing its flexibility and problem-solving abilities. Leveraging the extensive ecosystem of third-party Python packages, the agent can interact with a wide range of systems and tools.

Through extensive experiments on the GAIA benchmark (Mialon et al., 2024)—a comprehensive suite designed to evaluate the generality and adaptability of intelligent agents—we demonstrate that our framework enables extremely versatile LLM agents. The agent is capable of handling diverse tasks and file types without requiring human implementation of supporting functions. While the LLM agent is performant and capable on its own, extending this framework by incorporating tools developed by human experts is straightforward by simply including these tools in the agent's action set. We find that combining human-developed tools with agent-generated functions results in complementary capabilities, further enhancing the agent's performance and versatility.

2 Problem Formulation

We begin by formally stating our problem of interest. We model the behavior of an LLM agent as a Partially Observable Markov Decision Process defined by the tuple $(\mathcal{U}, \mathcal{A}, \mathcal{S}, \mathcal{O}, T, Z)$, where $\mathcal{U}$ is the task space; $\mathcal{A}$ is the action space, which most existing works define as a finite set of predefined actions: $\mathcal{A} = \{a_1, \ldots, a_n\}$; $\mathcal{S}$ is the state space; $\mathcal{O}$ is the observation space, $T : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the state transition function, mapping a state-action pair to a probability distribution over subsequent states; and $Z : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{O})$ is the observation function, mapping a state-action pair to a probability distribution over observations. Given a task $u \in \mathcal{U}$, the agent starts in an initial state $s_0 \in \mathcal{S}$. At each time step t, the agent selects an action $a_t \in \mathcal{A}$ which causes the environment to transition to a new state s_{t+1} according to the transition probability $T(s_t, a_t)$. The agent then receives an observation $o_{t+1} \in \mathcal{O}$ drawn from the distribution $Z(s_{t+1}, a_t)$. This process repeats until the agent reaches a terminal state s_T that satisfies the original task u.

In this work, we are interested in a more general setting where $\mathcal{A}$ is not fixed in advance. Specifically, we introduce a potentially infinite set $\mathcal{A}^*$ of all possible actions the agent can propose. At each time step t, the agent is allowed to propose any action $a_t \in \mathcal{A}^*$ to solve the task u. The cumulative action set at time t is defined as $\mathcal{A}_t = \{a_1, a_2, \ldots, a_t\}$. Each new action a_t may be an entirely novel action or a composition of previously generated actions from $\mathcal{A}_{t-1}$. Consequently, the overall action space $\mathcal{A}$ evolves dynamically as the agent encounters more tasks in $\mathcal{U}$. The state transition function is accordingly redefined as $T : \mathcal{S} \times \mathcal{A}^* \to \mathcal{P}(\mathcal{S})$, and the observation function as $Z : \mathcal{S} \times \mathcal{A}^* \to \mathcal{P}(\mathcal{O})$.

3 Methodology

Action Representation. In order to design such an LLM agent system, our first problem is choosing an appropriate representation for the action space. Specifically, the action representation must satisfy the following criteria: (1) **Generality**: It must be sufficiently expressive to represent actions capable of solving a wide range of tasks, and (2) **Composability**: It must naturally support the composition of actions. Considering the widespread success of programming languages, particularly Python, in solving diverse problems and the strong code generation capabilities of current LLMs acquired during pretraining, we select Python as the representation of actions in $\mathcal{A}^*$. Specifically, each action $a \in \mathcal{A}^*$ is represented as a Python function. This choice not only satisfies the aforementioned criteria but also facilitates seamless integration with existing tools and libraries.

Action Retrieval. We observe in preliminary experiments that including all generated actions as part of the prompt runs the risk of exceeding the context limit as the agent generates more actions. To address this issue, we decompose the action set $\mathcal{A}$ into two subsets: a human-designed action set $\mathcal{A}^u$ and a generated action set $\mathcal{A}^g$. Only the actions in $\mathcal{A}^u$ are included in the prompt by default. To provide the agent access to actions in $\mathcal{A}^g$, we introduce an action retrieval function $R : \mathcal{Q} \times \mathbb{N} \to 2^{\mathcal{A}^g}$, where $\mathcal{Q}$ denotes the space of queries and $\mathbb{N}$ is the set of positive integers. We then instruct our agent to provide a one-line docstring describing the purpose of each action function it generates. The doc-

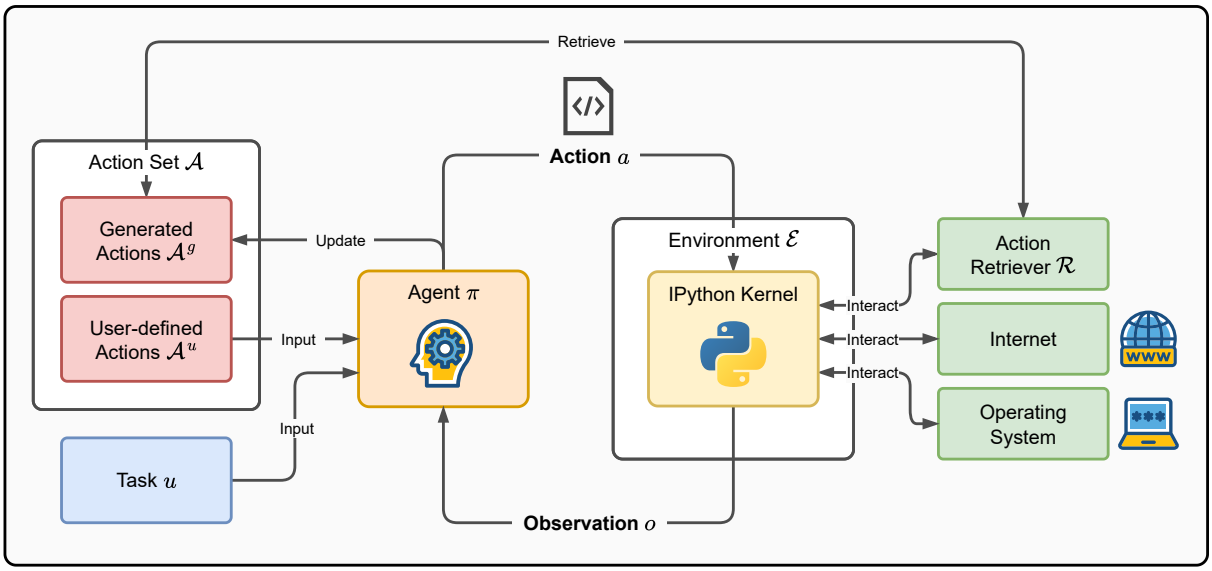

Figure 1: **Illustration of the DynaSaur agent framework**. In the first step, the agent receives a list of human-designed actions $\mathcal{A}^u$ and a task t as input. It then proposes an action a, implemented as a Python snippet. The function is executed by the environment, which internally contains an IPython kernel. Depending on the generated action a, the kernel may interact with either the action retriever, to retrieve relevant generated actions in $\mathcal{A}^g$; the internet, for information retrieval from the web; or the local operating system for any other tasks. We do not impose any constraints on which entities the agent can interact with, so the list shown in this figure is not exhaustive and is mainly for illustration purposes. After executing the action a, the environment returns an observation o to the agent. The observation can either be the result of executing a or an error message if the kernel fails to execute a.

strings are then embedded to form a set of indices of the generated actions. Given a query $q \in \mathcal{Q}$ and an integer $k \in \mathbb{N}$, the function $R(q, k)$ embeds the query using the same embedding, then computes the cosine similarity between the query's embedding and each action's docstring embedding. The top-k actions in $\mathcal{A}^g$ with the highest similarities are returned to the agent as part of its observations. To enable the agent to decide when to invoke action retrieval, we include the action retrieval function R itself as an action in the human-designed action set $\mathcal{A}^u$. Therefore, the agent can autonomously decide to perform action retrieval by selecting R during its decision-making process.

Action Accumulation. Our complete pipeline is illustrated in Figure 1: Given a task $u \in \mathcal{U}$ and a human-designed action set $\mathcal{A}^u$ with $R \in \mathcal{A}^u$, at time step t, we sample a thought-action pair $(h_t, a_t) \sim \pi_\theta(a_t \mid \mathcal{A}^u, u, c_{t-1})$ following the ReAct framework (Yao et al., 2023b), where $c_{t-1} = \{(h_1, a_1, o_1), \ldots, (h_{t-1}, a_{t-1}, o_{t-1})\}$ represents the interaction history up to time $t-1$. The action a_t is executed, and an observation o_t is returned from the environment, updating the context to $c_t = c_{t-1} \cup \{(h_t, a_t, o_t)\}$. If a_t contains a new function not present in $\mathcal{A}^g_{t-1}$, we update the generated action set by setting $\mathcal{A}^g_t = \mathcal{A}^g_{t-1} \cup f(a_t)$, where $f(a_t)$ denotes the set of functions defined in action a_t. Our detailed prompt can be found in Figure 7. Note that the ordering in which tasks are presented forms a curriculum that influences the growth of $\mathcal{A}^g$. Consequently, the agent's performance on a task u_t may depend on previous tasks due to this accumulation. For evaluation, we employ action accumulation during training but disable it during testing. This approach ensures that performance on each test task is independent of other test tasks.

4 Experiments

4.1 Experimental Setup

Benchmark. Although numerous benchmarks exist for evaluating LLM agents such as WebArena (Zhou et al., 2024a), WebShop (Yao et al., 2022), Mind2Web (Deng et al., 2023), ALFWorld (Shridhar et al., 2021), and MiniWoB++ (Liu et al., 2018), they are not suitable for assessing our proposed agent framework. First, these environments accept only a limited set of actions that an agent can perform and do not support arbitrary action execution. Second, they are simplistic and focus solely on very narrow types of tasks. GAIA (Mialon et al., 2024),

Agent Pipeline	GPT-4o mini				GPT-4o			
	Level 1	Level 2	Level 3	Avg.	Level 1	Level 2	Level 3	Avg.
MMAC (rep.)	-	-	-	-	45.16	20.75	6.12	25.91
AutoGen Multi-Agent (rep.)	-	-	-	-	47.31	28.93	14.58	32.33
HF Agent (rep.)	-	-	-	-	49.46	28.30	18.75	33.33
Sibyl (rep.)	-	-	-	-	47.31	32.70	16.33	34.55
Trase Agent (rep.)	-	-	-	-	50.54	33.33	14.29	35.55
No Pipeline	7.53	4.40	0.00	4.65	13.98	8.81	2.04	9.30
Sibyl (repl.)	21.51	15.72	4.08	15.61	38.71	24.53	10.20	26.58
HF Agent (repl.)	32.26	21.38	**8.33**	22.67	39.78	27.04	14.58	29.00
`DynaSaur`	**45.16**	**22.01**	8.16	**26.91**	**51.61**	**36.48**	**18.37**	**38.21**

Table 1: Performance comparison between various baseline methods and our proposed approach on the GAIA benchmark, evaluated under two LLM backbones: `gpt-4o-2024-08-06` and `gpt-4o-mini-2024-07-18`. "No Pipeline" refers to the baseline where no agent pipeline is employed, and the raw LLM is used. Results marked with (rep.) are reported results, while (repl.) indicates replicated results. Each value represents the average exact match percentage between the predicted answers and the ground truth.

on the other hand, is a benchmark specifically designed to stress-test the capabilities of generalist agents across a wide range of tasks without imposing constraints on how an agent must interact with the environment. It covers a wide range of tasks such as reasoning, long-horizon tool use, and comprehension of diverse file types (e.g., `xlsx`, `png`, or `pdf`). Additionally, GAIA's tasks are designed to have short, single correct answers, which facilitates straightforward evaluation. For these reasons, we evaluate our framework on GAIA.

Baselines. We include the top 5 state-of-the-art agent systems from the GAIA leaderboard: MMAC v1.1 (MMAC), Multi-Agent Experiment v0.1 (AutoGen Multi-Agent) (Wu et al., 2023), Hugging Face Agents (HF Agent) (Roucher, 2024), Sibyl System v0.2 (Sibyl) (Wang et al., 2024), and Trase Agent. However, only HF Agent and Sibyl have published their code, so we only consider them for replication. Additionally, we assess the performance of raw GPT-4o models (without any agentic framework) to establish a lower bound for comparison.

Initial Actions. For our proposed method, following HF Agent, we provide an initial action set with tools from Microsoft's AutoGen (Wu et al., 2023), including a web browser, a file inspection tool that converts various file types into machine-readable Markdown format, and a visual question-answering tool. The detailed list of tools and their description can be found in Table 3.

Models. We utilize two LLM backbones for all agentic pipelines: GPT-4o (`gpt-4o-2024-08-06`) and GPT-4o mini (`gpt-4o-mini-2024-07-18`) through Azure OpenAI API. For further analyses, to save costs, we only evaluate using GPT-4o.

Implementation Details. We use OpenAI's `text-embedding-3-large` as the embedding model and set the number of retrieved actions to $k = 10$. We limit the maximum number of steps to 20 and set the temperature to 0.5 for all experiments. In the main experiment, we first run our agent on all examples in the validation set and accumulate the generated actions. We then freeze the action set for evaluation on the test set. Since only the GAIA validation set contains labels, for further analyses, we instead run action accumulation on 200 test examples, freeze the action set, and evaluate on the entire validation set. As our proposed pipeline does not require ground truth labels during the action learning phase, we are able to do so.

4.2 Main Results

We evaluate our proposed method and compare its performance with selected baselines in Table 1. As shown in the table, `DynaSaur` significantly outperforms previous baselines for both LLM backbones across all difficulty levels of the GAIA benchmark. This indicates that the ability to perform arbitrary actions, combined with the accumulation of more actions over time, offers significant advantages over traditional LLM agent pipelines with fixed, predefined action sets, especially in highly complex, long-horizon problems such as GAIA level 2 and

#	AA	AI	IA	Level 1	Level 2	Level 3	Avg.
1	✓	✓	✓	49.06	41.86	26.92	41.82
2	✗	✓	✓	47.17	40.70	15.38	38.79
3	✗	✗	✓	43.40	37.21	11.54	35.15
4	✓	✓	✗	35.85	19.77	7.69	23.03
5	✗	✓	✗	33.96	18.60	7.69	21.82

Table 2: Ablation of three major components in our framework: action accumulation (denoted as AA), action implementation (denoted as AI), and the initial set of actions (denoted at IA). Each number is the average exact match percentage between the predicted answers and the ground truth.

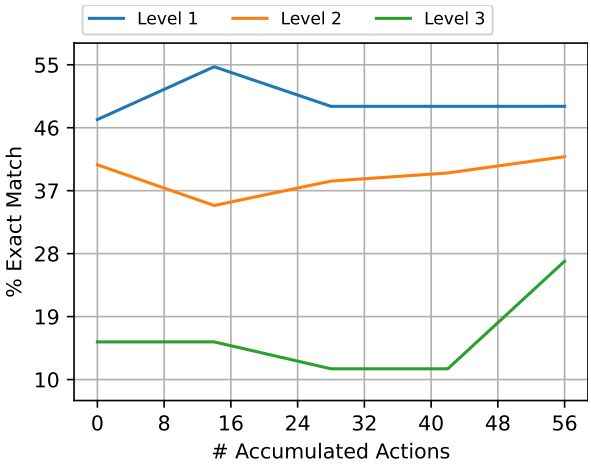

Figure 2: Impact of action accumulation on performance over time.

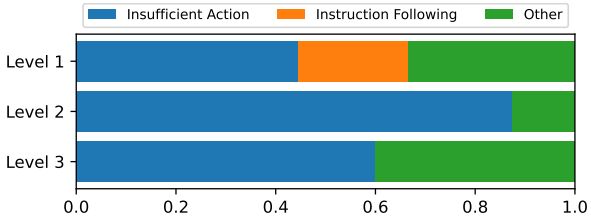

Figure 3: Distribution of error types in tasks where agent A (without action implementation) answers incorrectly, while agent B (with action implementation) answers correctly.

3 tasks. Note that in this experiment, because it is unclear which version of GPT-4o the HF Agent and Sibyl use, we reevaluated their pipelines under the same LLM backbones as ours for a fair comparison and included their reported results from the GAIA public leaderboard as references.

4.3 Ablation Study

Our first analysis focuses on the ablation of key components in our agent's pipeline. We highlight three main components: the initial set of actions (denoted as IA), the capacity to implement arbitrary actions (denoted as AI), and the ability to accumulate actions generated from previous episodes (denoted as AA). It's important to note that AA is dependent on AI, meaning action accumulation is only possible if the agent is capable of implementing arbitrary actions. Rows 1-2 and 4-5 presented in Table 2, demonstrate that action accumulation enhances overall performance in both scenarios, with and without initial actions, yielding average improvements of 7.81% and 5.55%, respectively. Additionally, rows 2 and 3 show that enabling arbitrary action implementation improves performance across all difficulty levels, with a 10.36% increase. However, the most substantial improvement comes from IA, with an impressive 81.59% gain. This is expected, as these initial actions are designed by human experts to provide the LLM agent with tools to perform a wide range of tasks.

4.3.1 Do Agents Benefit From Action Accumulation?

To better understand how action accumulation influences performance over time, we remove 25%, 50%, and 75% of the generated actions and re-evaluate the agent's performance on the validation set. As depicted in Figure 2, performance improves with the accumulation of more actions. Notably, the most substantial gains occur in level 3 tasks, while level 1 and level 2 tasks see more modest improvements. Interestingly, at the 25% removal point, level 1 performance peaks, which we attribute to small variances in trajectory sampling from GPT-4o. However, due to budget constraints, we were unable to further quantify this variance.

4.3.2 How Does Implementing Arbitrary Actions Improve Agent Performance?

To better understand the specific advantages of action implementation, we filtered out tasks that an agent without action implementation (referred to as agent A) answered incorrectly but an agent with action implementation (referred to as agent B) answered correctly. We then analyzed the reasons why agent A failed at these tasks and whether enabling action implementation in agent B helped resolve these limitations. We selected pipeline variants from row 3 in Table 2 as agent A and row 1 as agent B. After filtering, we obtained a set of 22 tasks. Because each trajectory is long and time-consuming to examine manually, we employed OpenAI's o1 model (`o1-preview-2024-09-12`) as

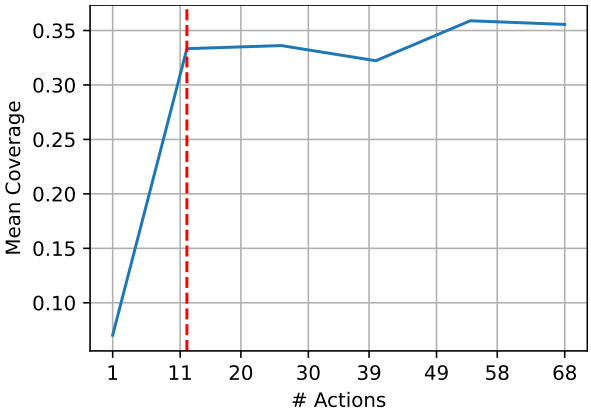

Figure 4: Mean coverage over the validation set as the number of actions increases. The red dashed line marks the point where human-designed actions are added to the action set. Subsequent data points reflect the accumulation of generated actions.

an evaluator. For each task, we provided o1 with the task, the correct answer, the reference trajectory from a human annotator, agent A's answer and trajectory, as well as agent B's answer and trajectory. We instructed o1 to summarize both agents' approaches with explanations for success or failure, then explain whether agent B succeeded or failed because of its ability to implement new actions. The detailed prompt is provided in Figure 6 in the Appendix. After o1's evaluation, we manually analyzed the reports from o1 to further categorize agent A's errors into three types: (1) failure due to insufficient tooling, (2) failure to correctly follow instructions, and (3) failure due to other reasons.

Our findings reveal that 61.91% of the failures were due to reason 1, with 12 cases where the agent lacked the necessary tools to solve the task, and 1 case where a human-designed tool failed to return relevant information. In 9.52% of the cases, agent A failed due to reason 2 (e.g., returning an answer with an incorrect unit). The remaining 28.57% of the failures were caused by other unrelated factors, such as the inability to find relevant information online or getting stuck without making progress. A more detailed breakdown of the error distribution for each level is shown in Figure 3. In all type-1 errors, agent B was able to complete the task by implementing custom actions. This result demonstrates that our framework significantly improves the agent's flexibility in problem solving.

4.4 Measuring Action Coverage

In this experiment, we aim to evaluate the quality of the generated action set $\mathcal{A}^g$, particularly focusing on the transferability of these actions to unseen tasks. To quantify this, we propose a metric that measures how effectively an action set $\mathcal{A}$ can cover a task u. Given a task u, a ground truth answer y, and an action set $\mathcal{A}$, we sample a trajectory $\tau = \{(h_1, a_1, o_1), \ldots, (h_T, a_T, o_T)\}$ from the policy $\pi_\theta(\cdot \mid \mathcal{A}, u)$, where h_i, a_i, and o_i denote the thought, action, and observation at step i, respectively. We consider the policy π_θ to have solved the task u if $o_T = y$. At each step in τ, the agent either reuses an existing action from $\mathcal{A}$ or generates a new action not in $\mathcal{A}$. We say that the action set $\mathcal{A}$ is insufficient for the agent π_θ to solve the task u if it must generate a new action $a \notin \mathcal{A}$. We define the coverage of $\mathcal{A}$ over u under π_θ as the rate at which the agent generates new actions not in $\mathcal{A}$ during the execution trajectory τ, conditioned on the task being successfully completed. Formally, we define:

$$C(\mathcal{A}, u) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\cdot \mid \mathcal{A}, u)} \left[1 - \frac{1}{|\tau|} \mathbf{1}[o_T = y] \cdot \left| \{(h_i, a_i, o_i) \in \tau : a_i \notin \mathcal{A}\} \right| \right] \quad (1)$$

However, computing this metric exactly is prohibitively expensive. In practice, we approximate it by sampling a single trajectory for each task. To compute the overall coverage across tasks, we average the coverage values obtained for each task, with the results shown in Figure 4. Note that in the actual implementation, we always include at least one action, `submit_final_answer`, so the action set is never truly empty. As expected, when $|\mathcal{A}| = 1$, the mean coverage is extremely low at 0.07. We observe that the mean coverage at this point is not completely zero because there are a few easy tasks where the agent submits its answer on the first step. The next data point in the plot, marked with a dashed red line, is when human-designed tools are introduced into the action set, after which we observe a significant increase in mean coverage. This aligns with our empirical observations: because human-designed tools were originally made for LLM agents to solve various tasks, our agent uses them frequently and generates new actions significantly less often than the variant without access to these tools. In subsequent data points, as more actions are generated, we observe slight fluctuations in mean coverage. However, the overall trend remains modestly upward. These fluctuations are due to the variance from sampling only

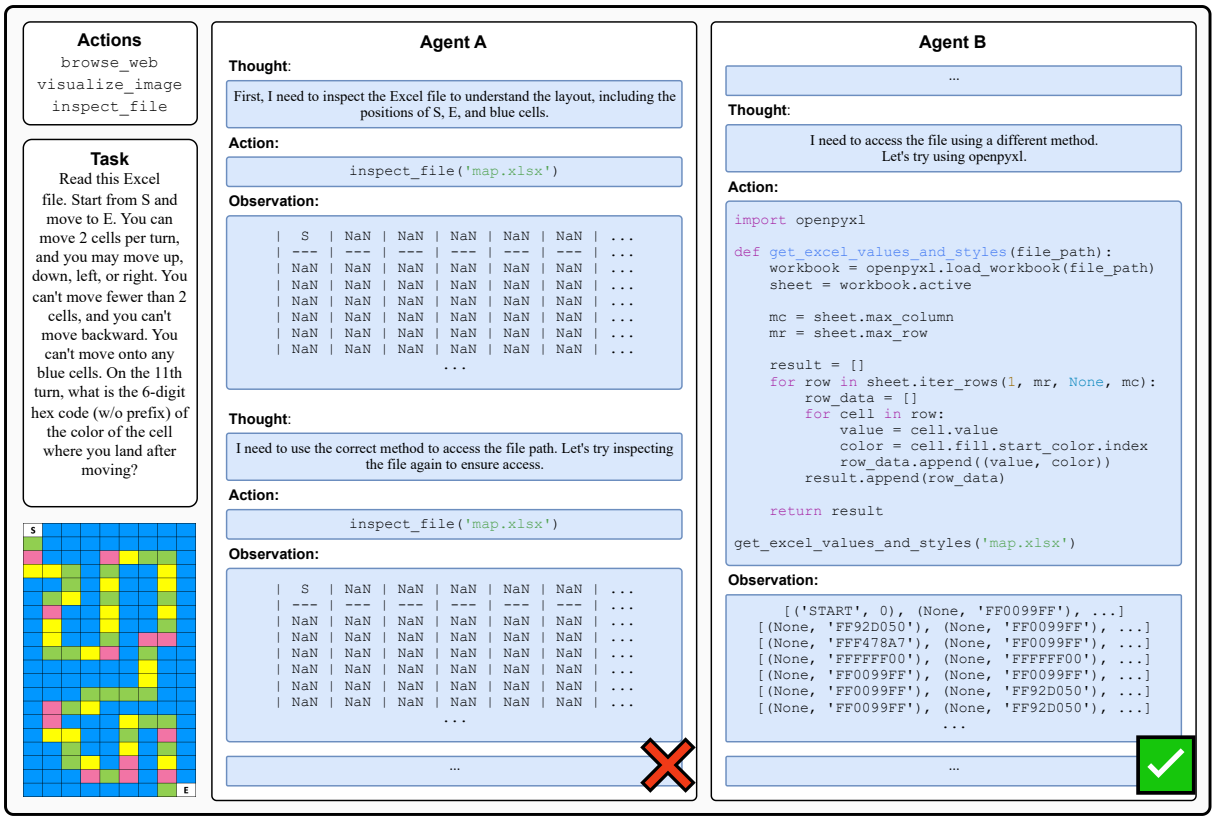

Figure 5: A case study demonstrates the difference in problem-solving flexibility between Agent A (a variant of `DynaSaur` without action implementation) and Agent B (the proposed agent framework). Both agents begin with the same initial step, but only Agent B, equipped with the ability to implement its own actions, successfully completes the task. Due to space constraints, the first step taken by Agent B is not shown.

a single trajectory per task and should diminish as the number of sampled trajectories increases.

4.5 Case Studies

We present a real case study comparing how an agent without action implementation (denoted as agent A) and an agent with action implementation (denoted as agent B) approach the same problem. In this example, the task requires the agents to load an Excel file containing a map, as shown in the lower left corner of Figure 5. The agent must then navigate through the map according to the task's movement rules and, after the 11th turn, return the color of the current cell. The provided action set is similar to previous experiments. In this scenario, the `inspect_file` tool, developed by Microsoft's AutoGen (Wu et al., 2023), assists an agent by reading diverse file types and returning the file content in Markdown format. However, when reading Excel files, the tool does not account for formatting properties such as cell color, leading to incomplete information being returned and preventing agent A from solving the task. Since agent A lacks other tools, it repeatedly attempts to invoke the `inspect_file` tool until the maximum iteration limit is reached. On the other hand, agent B also initially tries to invoke the same tool but recovers from the error by using a different approach to read the Excel file content through openpyxl. In subsequent steps, agent B implements the solution for map navigation as a function and successfully completes the task. However, due to space constraints, we are unable to show the entire trajectory.

5 Related Work

5.1 LLM Agents

Most current methods that utilize LLMs for agent tasks involve prompting techniques (Yao et al., 2023a; Liang et al., 2023; Gao et al., 2023; Kim et al., 2023), supervised fine-tuning (Schick et al., 2023b; Zeng et al., 2023; Chen et al., 2024b; Zhang et al., 2024a; Chen et al., 2023b), or reinforcement learning (RL) algorithms for self-exploration (Zhou et al., 2024b; Song et al., 2024; Yang et al., 2024; Aksitov et al., 2023; Christianos et al., 2023; Abdulhai et al., 2023; Gulcehre et al., 2023). However,

these approaches mainly rely on a fixed, predefined set of actions and use text (Schick et al., 2023b) or JSON (Qin et al., 2023) as the representation of actions, which significantly lack the two criteria we mentioned earlier: Generality and Composability. In contrast, using code as the action representation, in principle, enables agents to solve any Turing-complete problems.

5.2 Tool-using LLM Agents

Although the combination of LLMs and code is not new, having a long history that dates back to the early stages of LLM development (Chen et al., 2021b; Austin et al., 2021; Hendrycks et al., 2021), this research primarily focused on using LLMs as software engineering assistants for tasks like code completion or program synthesis (Austin et al., 2021; Zhang et al., 2024b). In our work, we use programming languages as a means to solve generalist AI agent tasks in the GAIA benchmark, which require multistep execution in partially observable and stochastic environments.

5.3 Tool-creation LLM Agents

There have been a few attempts to explore LLMs' ability to create their own tools, though these efforts have largely been limited to solving simple problems (Cai et al., 2023; Qian et al., 2023; Wang et al., 2023; Yuan et al., 2023). For example, (Cai et al., 2023) examines LLMs generating code snippets to tackle basic tasks such as word sorting or simple logical deduction. Their approach involves sampling three input-output pairs of a specific task type, using the LLM to generate a function to solve the problem, validating it with three additional pairs from the validation set, and then evaluating the solution on all test instances from the same task type. This setup simplifies the problem as the task type remains consistent during both training and testing. Similarly, (Qian et al., 2023) and (Yuan et al., 2023) explore tool creation, but restrict their focus to math problems, with (Yuan et al., 2023) also introducing VQA benchmarks. These tasks are typically solvable in a single step and do not require interaction with an external environment. We are the first to study generalist LLM agents that implement and accumulate actions within the real-world decision-making benchmark GAIA.

6 Conclusion

We have explored an LLM agent framework that implements its own actions as Python functions to interact with the world and accumulate its generated actions over time, thus growing a toolset of actions for problem-solving in future tasks. This framework aims to address the limitations of previous paradigms, where agents selected actions from a fixed, predefined set, greatly reducing their flexibility. Extensive experiments and analyses show that our agents are significantly more flexible and performant, supporting the potential of this new framework. Specifically, we achieved the top rank on the GAIA public leaderboard, one of the most challenging benchmarks for AI agents.

7 Limitations

One consistent limitation we observe when deploying our agent is its tendency to generate actions that are overly specific to a given task, despite being explicitly instructed to produce more generic and general functions. This issue is compounded by GAIA's diverse set of tasks, leading to a resulting set of generated actions that is often "sparse"—in the sense that the actions are largely irrelevant to one another. As a result, the agent seldom reuses past actions or creates new ones by composing lower-level actions. We hypothesize that to address this issue, we need to develop a task curriculum that provides a continuous stream of similar, relevant tasks. This would encourage more effective growth of the action set and the composition of higher-level actions. Another limitation of this work is that we only evaluate our method on OpenAI's models due to the high cost of running each GAIA task.

8 Ethical Considerations

Since we allow the agent to write and execute code for arbitrary actions, a natural concern arises regarding the safety implications. While we have empirically observed that GPT-4o does not produce harmful code, it's still advisable to evaluate the agent within a containerized environment like Docker. Additionally, exploring methods to constrain the LLM agent's code execution space to ensure that it remains safe without being overly restrictive would be an important future research direction.

References

Marwa Abdulhai, Isadora White, Charlie Snell, Charles Sun, Joey Hong, Yuexiang Zhai, Kelvin Xu, and Sergey Levine. 2023. Lmrl gym: Benchmarks

for multi-turn reinforcement learning with language models. *Preprint*, arXiv:2311.18232.

Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Kopparapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, Manzil Zaheer, Felix Yu, and Sanjiv Kumar. 2023. Rest meets react: Self-improvement for multi-step reasoning llm agent. *Preprint*, arXiv:2312.10003.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. 2023. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *ArXiv*, abs/2305.17126.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023a. Fireact: Toward language agent fine-tuning. *CoRR*, abs/2310.05915.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023b. Fireact: Toward language agent fine-tuning. *Preprint*, arXiv:2310.05915.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024a. Agent-flan: Designing data and methods of effective agent tuning for large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 9354–9366. Association for Computational Linguistics.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024b. Agent-flan: Designing data and methods of effective agent tuning for large language models. *Preprint*, arXiv:2403.12881.

Filippos Christianos, Georgios Papoudakis, Matthieu Zimmer, Thomas Coste, Zhihao Wu, Jingxuan Chen, Khyati Khandelwal, James Doran, Xidong Feng, Jiacheng Liu, Zheng Xiong, Yicheng Luo, Jianye Hao, Kun Shao, Haitham Bou-Ammar, and Jun Wang. 2023. Pangu-agent: A fine-tunable generalist agent with structured reasoning. *Preprint*, arXiv:2312.14878.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samual Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the

web. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. 2023. Reinforced self-training (rest) for language modeling. *Preprint*, arXiv:2308.08998.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *Preprint*, arXiv:2303.17491.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. *Preprint*, arXiv:2209.07753.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Cheng Qian, Chi Han, Yi Ren Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Conference on Empirical Methods in Natural Language Processing*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530.

Aymeric Roucher. 2024. Huggingface agent.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023a. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023b. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nat.*, 588(7839):604–609.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. 2021. Alfworld: Aligning text and embodied environments for interactive learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian

Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. 2024. Trial and error: Exploration-based trajectory optimization for llm agents. *Preprint*, arXiv:2403.02502.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354.

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. *Preprint*, arXiv:2305.16291.

Yulong Wang, Tianhao Shen, Lifeng Liu, and Jian Xie. 2024. Sibyl: Simple yet effective agent framework for complex real-world reasoning. *CoRR*, abs/2407.10718.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155.

Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Piyush Khandelwal, Varun Raj Kompella, HaoChih Lin, Patrick MacAlpine, Declan Oller, Takuma Seno, Craig Sherstan, Michael D. Thomure, Houmehr Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Dürr, Peter Stone, Michael Spranger, and Hiroaki Kitano. 2022. Outracing champion gran turismo drivers with deep reinforcement learning. *Nat.*, 602(7896):223–228.

Zonghan Yang, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. 2024. React meets actre: When language agents enjoy training data autonomy. *Preprint*, arXiv:2403.14589.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023a. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *Preprint*, arXiv:2310.12823.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2024. Agenttuning: Enabling generalized agent abilities for llms. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3053–3077. Association for Computational Linguistics.

Jianguo Zhang, Tian Lan, Rithesh Murthy, Zhiwei Liu, Weiran Yao, Juntao Tan, Thai Hoang, Liangwei Yang, Yihao Feng, Zuxin Liu, Tulika Awalgaonkar, Juan Carlos Niebles, Silvio Savarese, Shelby Heinecke, Huan Wang, and Caiming Xiong. 2024a. Agentohana: Design unified data and training pipeline for effective agent learning. *Preprint*, arXiv:2402.15506.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024b. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *Preprint*, arXiv:2401.07339.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024a. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. 2024b. Archer: Training language model agents via hierarchical multi-turn rl. *Preprint*, arXiv:2402.19446.

#	Name
1	submit_final_answer
2	get_relevant_actions
3	informational_web_search
4	navigational_web_search
5	visit_page
6	download_file
7	page_up
8	page_down
9	find_on_page_ctrl_f
10	find_next
11	find_archived_url
12	visualizer
13	inspect_file_as_text

Table 3: List of initial actions used in this project.

A Implementation Details

A.1 Initial Actions

A.2 Prompt For Qualitative Analysis

A.3 DynaSaur ’s System Prompt

```
There are two types of LLM agents: agent A and agent
B. Both types of agents work as follows: Given a
task and the same set of actions T, both agents
proceed in a series of steps to solve the task.
However, agent A only uses actions from T at each
step, while agent B either uses actions from T or
implements new actions as Python functions if T is
not sufficient (e.g., when the task requires
processing an .xlsx file but T only contains actions
for web browsing and visual question answering).

You will be given a task, the correct answer, the
gold trajectory from a human, agent A's predicted
answer, agent A's trajectory, agent B's predicted
answer, and agent B's trajectory. Your task is to
write a report evaluating which agent performs
better and why. Focus on how agent B's ability to
implement its own actions affects its performance
(either positively or negatively). Your report
should follow this JSON format:

```json
{
    "task_summary": "Brief summary of the task",
    "A_summary": "Brief summary of agent A's
trajectory",
    "B_summary": "Brief summary of agent B's
trajectory",
    "better_agent": "Output `A` or `B` depending on
which one is better",
    "why_worse": "Explain why the worse agent
answered incorrectly or performed worse.",
    "why_better": "Explain why the better agent
answered correctly or performed better.",
    "impact_of_action_implementation": "If agent B
performs better or worse, is it due to its ability
to implement new functions? Answer Yes or No and
provide a brief explanation."
}
```

Here are the necessary information:
Task
{question}

Gold answer
{gold_ans}

Gold trajectory
{gold_traj}

AI agent A's answer
{A_pred_ans}

AI agent A's trajectory
{A_pred_traj}

AI agent B's answer
{B_pred_ans}

AI agent B's trajectory
{B_pred_traj}
```

Figure 6: Prompt for OpenAI’s o1 to perform qualitative evaluation.

Instructions
You are an AI assistant that helps users solve problems. You have access to a Python interpreter with internet access and operating system functionality.

When given a task, proceed step by step to solve it. At each step:
1. **Thought**: Briefly explain your reasoning and what you plan to do next.
2. **Code**: Provide Python code that implements your plan. For example, to interact with or gather information from web pages, use `requests`, `bs4`, `lxml`, or `selenium`. To handle or read Excel files, use `openpyxl` or `xlrd`. To handle or read PDF files, use `PyMuPDF`. If the relevant packages are not installed, write code to install them using `pip`. These examples are not exhaustive, feel free to use other appropriate packages.

The interpreter will execute your code and return the results to you. Review the results from current and previous steps to decide your next action.

Continue this process until you find the solution or reach a maximum of <<max_iterations>> iterations. Once you have the final answer, use the `submit_final_answer` function to return it to the user.

Output Format
At each step, output a JSON object in the following format:

```json
{
    "thought": "Your thought here.",
    "code": "Your Python code here."
}
```

Example:

```json
{
    "thought": "I need to retrieve the HTML content of the target webpage.",
    "code": "import requests\n\ndef get_html_content(url):\n    response = requests.get(url)\n    return response.text\n\nhtml_content = get_html_content('http://example.com')"
}
```

Available Functions
You are provided with several available functions. If you need to discover more relevant functions, use the `get_relevant_tools` function.
```
<<tool_descriptions>>
```

Guidelines for Writing Code
1. First, decide whether to reuse an existing function or define a new one.
2. Look at the list of available functions. If no existing function is relevant, run `get_relevant_tools` to find more functions and proceed to the next step.
3. If the retrieved functions are still not relevant, define a new function.
4. When implementing a new function, you must ensure the following:
   - The function is abstract, modular, and reusable. Specifically, the function name must be generic (e.g., `count_objects` instead of `count_apples`). The function must use parameters instead of hard-coded values. The function body must be self-contained.
   - Explicitly declare input and output data types using type hints.
   Example: `def function_name(param: int) -> str:`
   - Include a one-line docstring describing the function's purpose, following PEP 257 standards.
   - When your function calls multiple other functions that are not from a third-party library, ensure you print the output after each call. This will help identify any function that produces incorrect or unexpected results.

Guidelines for Analyzing the Output
After execution, analyze the output as follows:
1. If the code fails to execute successfully and an error is returned, read the error message and traceback carefully, then revise your code in the next step.
2. If the code executes successfully and an output is returned, proceed as follows:
   - If the output contains relevant information, you can move on to the next step.
   - If the output does not contain any relevant information, consider alternative approaches. For example, try different data sources or websites, use different functions or libraries, implement new functions if necessary.

Important Notes
1. When reading a file or a web page, make sure you have read all the content in it so you don't miss any details and arrive at the wrong conclusion.
2. Pay close attention to the task specifics, such as the required unit of the answer or how many digits to round to.
3. Base your decisions on real-world data. All tasks are backed by real-world data, which is either available on the internet or in the file provided to you. Rely solely on real-world data to generate your answers; do not rely on your own knowledge, and do not imagine data out of nowhere, as it will mislead you to an incorrect answer. In your code, write comments that cite your data sources (e.g., which website it came from, which line in the file, etc.) so that a human can verify them.
4. DO NOT GIVE UP. Keep tr ying until you reach the maximum iteration limit.

Figure 7: The system prompt of our DynaSaur agent framework.