

DynaSaur : Large Language Agents Beyond Predefined Actions

Dang Nguyen^{1*}, Viet Dac Lai², Seunghyun Yoon², Ryan A. Rossi²,
Handong Zhao², Ruiyi Zhang², Puneet Mathur², Nedim Lipka²,
Yu Wang², Trung Bui², Franck Dernoncourt², Tianyi Zhou¹

¹University of Maryland, ²Adobe Research
{dangmn, tianyi}@umd.edu

Abstract

Existing LLM agent systems typically select actions from a fixed and predefined set at every step. While this approach is effective in closed, narrowly scoped environments, it presents two major challenges for real-world, open-ended scenarios: (1) it significantly restricts the planning and acting capabilities of LLM agents, and (2) it requires substantial human effort to enumerate and implement all possible actions, which is impractical in complex environments with a vast number of potential actions. To address these limitations, we propose an LLM agent framework that can dynamically create and compose actions as needed. In this framework, the agent interacts with its environment by generating and executing programs written in a general-purpose programming language. Moreover, generated actions are accumulated over time for future reuse. Our extensive experiments across multiple benchmarks show that this framework significantly improves flexibility and outperforms prior methods that rely on a fixed action set. Notably, it enables LLM agents to adapt and recover in scenarios where predefined actions are insufficient or fail due to unforeseen edge cases. Our code can be found in <https://github.com/adobe-research/dynasaur>.

1 Introduction

Developing autonomous agents has long been a central goal in AI research. While reinforcement learning has extensively studied this problem and has 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 in Large Language Models (LLMs) (Chen et al., 2021a; OpenAI, 2023; Bubeck et al., 2023; Anil et al., 2023; Reid et al., 2024) with strong reasoning ability and the vast amount of 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). Notable initial works include Toolformer (Schick et al., 2023a), which explores self-supervised training for LLM agents to utilize external tools, such as calculators, search engines, and translation services, thereby enhancing responses to complex question-answering tasks. ReAct (Yao et al., 2023b) proposes a synergistic approach by interleaving reasoning and action sequences at each step, which has become the de facto prompting framework in most LLM agent systems. Reflexion (Shinn et al., 2023), a follow-up work, investigates LLM agents that maintain a set of self-reflections on their past mistakes in failed trajectories; conditioning on self-reflection feedback significantly improves agent performance across various benchmarks, albeit with the trade-off of increased inference costs.

Despite these efforts, most existing LLM agent systems are studied in closed, simulated environments that accept only a finite and small set of predefined actions (Zhou et al.,

*Work done during internship at Adobe Research.

2024a; Yao et al., 2022; Deng et al., 2023; Shridhar et al., 2021; Liu et al., 2018). 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. To address these limitations, we propose DynaSaur, an LLM agent framework that enables the dynamic creation and composition of arbitrary actions by modeling 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 existing 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. All actions generated by the agent are accumulated over time, building a library of reusable functions for future use. This approach allows the agent to extend its capabilities on the fly and compose complex actions from simpler ones, thereby enhancing its flexibility and problem-solving abilities. By leveraging the extensive ecosystem of third-party Python packages, the agent can interact with a wide range of systems and tools.

Through experiments on a diverse set of benchmarks, including GAIA (Mialon et al., 2024), MATH (Hendrycks et al., 2021b), TabMWP (Lu et al., 2023), AIME (Li et al., 2024), and GPQA (Rein et al., 2023), 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 the framework by incorporating tools developed by human experts is straightforward, simply include these tools in the agent’s action set. We find that combining human-designed 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: $\mathcal{A} = \{a_1, \dots, a_n\}$; $\mathcal{S}$ is the state space; $\mathcal{O}$ is the observation space, $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$ is the state transition function, mapping a state-action pair to a probability distribution over subsequent states; and $Z : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{O})$ 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 paper, 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 $\mathcal{A}_t = \{a_1, a_2, \dots, a_t\}$. 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 $T : \mathcal{S} \times \mathcal{A}^* \rightarrow \mathcal{P}(\mathcal{S})$, and the observation function as $Z : \mathcal{S} \times \mathcal{A}^* \rightarrow \mathcal{P}(\mathcal{O})$.

3 Methodology

Action Representation. To design such an LLM agent system, our first challenge is to select an appropriate representation for the action space. This representation must satisfy

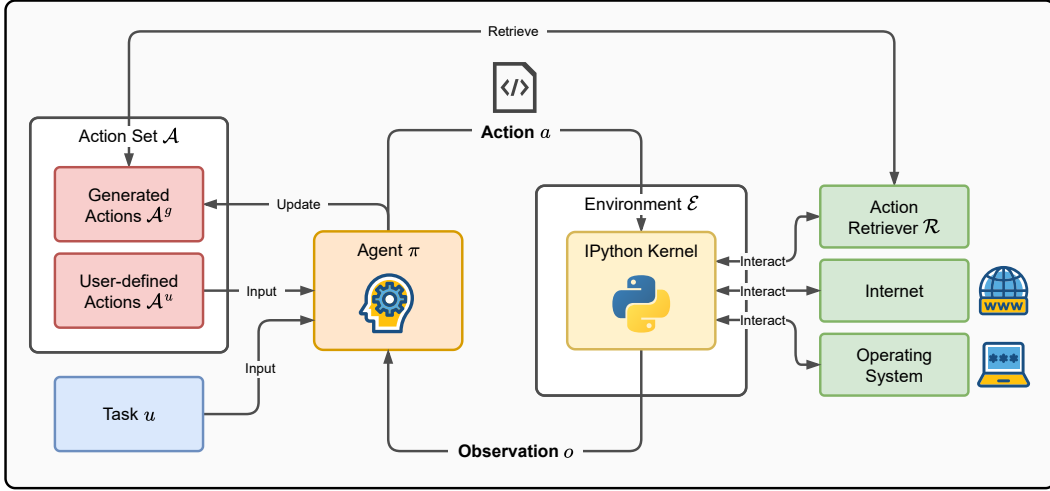


Figure 1: **Illustration of the DynaSaur agent framework.** The agent π receives a task t and optionally a set of human-designed actions $\mathcal{A}^u$. It then interacts with an environment $\mathcal{E}$ by proposing an action $a \in \mathcal{A}$, implemented as a Python function. The action is executed in an IPython kernel, which may interface with the operating system, the internet, or the action retriever as needed. The result, either the output of the function or an error message, is returned to the agent as an observation o . Generated actions that execute successfully are accumulated into $\mathcal{A}^g$.

two key criteria: (1) **Generality**: it should be expressive enough to represent actions capable of solving a wide range of tasks; and (2) **Composability**: it should naturally support the composition of actions. We argue that a general-purpose programming language meets these requirements well. We choose Python for its popularity and extensive ecosystem of libraries. This choice not only satisfies the aforementioned criteria but also facilitates seamless integration with existing tools and libraries. In our framework, each action $a \in \mathcal{A}^*$ is represented as a Python function.

Action Retrieval. 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: an optional human-designed action set $\mathcal{A}^u$ and a generated action set $\mathcal{A}^g$. Only actions in $\mathcal{A}^u$ are included in the prompt by default, allowing developers to inject domain-specific actions they consider important. To provide the agent access to actions in $\mathcal{A}^g$, we introduce an action retrieval function $R : \mathcal{Q} \times \mathbb{N} \rightarrow 2^{\mathcal{A}^g}$, where $\mathcal{Q}$ denotes the space of queries and $\mathbb{N}$ is the set of positive integers. When generating the actions, we also instruct our agent to provide a docstring describing the purpose of each action function it generates. The docstrings 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 $c_{t-1} = \{(h_1, a_1, o_1), \dots, (h_{t-1}, a_{t-1}, o_{t-1})\}$ 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 $\mathcal{A}_{t-1}^g$, we update the generated action set by setting $\mathcal{A}_t^g = \mathcal{A}_{t-1}^g \cup f(a_t)$, where $f(a_t)$ denotes

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.

	No Pipeline	Sibyl System	HF Agent	DynaSaur	#	AA	AI	IA	Level 1	Level 2	Level 3	Avg.
MATH	77.86	74.29	80.71	82.14	1	✗	✓	✗	33.96	18.60	7.69	21.82
TabMWP	95.71	95.00	96.43	97.14	2	✓	✓	✗	35.85	19.77	7.69	23.03
AIME	13.00	20.00	20.00	31.71	3	✗	✗	✓	43.40	37.21	11.54	35.15
GPQA	48.00	46.00	38.00	54.00	4	✗	✓	✓	47.17	40.70	15.38	38.79
					5	✓	✓	✓	49.06	41.86	26.92	41.82

Table 2: Performance comparison between various baseline methods on additional datasets. We utilize gpt-4o-2024-08-06 as the LLM backbone in this experiment.

Table 3: Ablation study on the impact of three major components: Action Accumulation (AA), Action Implementation (AI), and Initial Actions (IA).

the set of functions defined in action a_i . Our detailed prompt can be found in Figure 7. 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

Benchmarks. While numerous interactive environments exist for 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 evaluating our proposed agent framework, as they only support a limited set of predefined actions and do not allow arbitrary action execution. We instead evaluate DynaSaur on a set of static datasets. Specifically, we consider GAIA (Mialon et al., 2024), a general agent benchmark covering a broad range of tasks including web browsing, file parsing and processing, symbolic reasoning, video understanding, and audio understanding. Additionally, we evaluate our agent on MATH (Hendrycks et al., 2021b), TabMWP (Lu et al., 2023), AIME (Li et al., 2024), and GPQA (Rein et al., 2023) for a more comprehensive assessment.

Baselines. We compare our method with agent systems that utilize a fixed set of predefined actions, including Hugging Face Agents (HF Agent) (Roucher, 2024) and Sibyl System v0.2 (Sibyl) (Wang et al., 2024b). For the GAIA benchmark, we also include MMAC v1.1 (MMAC) (Song et al., 2024c), Multi-Agent Experiment v0.1 (AutoGen Multi-Agent) (Wu et al., 2023), and Trase Agent (Systems, 2025) for reference. Additionally, we include vanilla GPT-4o models without any agentic framework to establish a lower bound for comparison.

Initial Actions. For a fair comparison with baselines, we initialize the action set with human-designed tools from Microsoft’s AutoGen (Wu et al., 2023), similar to HF Agent.

These tools include a web browser, a file inspection tool that converts various file types into machine-readable Markdown format, and a visual question-answering tool. A detailed list of the tools and their descriptions can be found in Table 5.

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.

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 demonstrates that the ability to perform arbitrary actions, combined with the capacity to accumulate actions over time, provides substantial advantages over traditional LLM agent pipelines with fixed action sets—particularly in highly complex, long-horizon tasks such as GAIA Level 2 and Level 3. In this experiment, because the exact version of GPT-4o used by HF Agent and Sibyl is unclear, we re-evaluated their pipelines under the same LLM backbones as ours to ensure a fair comparison. Their original results, as reported on the GAIA public leaderboard, are included as references. Results in Table 2 further show that our method consistently outperforms the baselines on the MATH, TabMWP, AIME, and GPQA benchmarks. We include additional experiments on open-source LLMs in Appendix A.

4.3 Further Analysis

In the following analysis, we use GAIA as the default dataset unless stated otherwise. Since only the GAIA validation set contains labels, we run action accumulation on 200 test examples and then evaluate on the entire validation set using the frozen action set.

4.3.1 The Impact of Action Accumulation, Action Implementation, and Initial Actions?

Our first analysis focuses on ablations of key components in the agent pipeline. We highlight three main components: (1) the initialization of the action set, (2) the capacity to implement arbitrary actions, and (3) the ability to accumulate actions across episodes. Notably, action accumulation depends on the agent’s ability to implement arbitrary actions, as previously generated actions must be executable.

As shown in Table 3, initializing the agent with a set of human-designed actions (row 3) improves performance on GAIA by 61% relative to the minimal baseline (row 1). This improvement is expected, as these tools are highly specialized for GAIA tasks, which often involve browsing the web or reading various file types. Adding support for arbitrary action implementation (row 4) further improves performance by 10% (relative to row 3), and enabling action accumulation (row 5) yields an additional 7% gain (relative to row 4). These results support the effectiveness of each component in our proposed framework.

4.3.2 How Does Implementing Arbitrary Actions Improve Agent Performance?

To dive deeper into understanding the specific advantages of action implementation, we filtered out GAIA 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 3 as agent A and row 1 as agent B. After filtering,

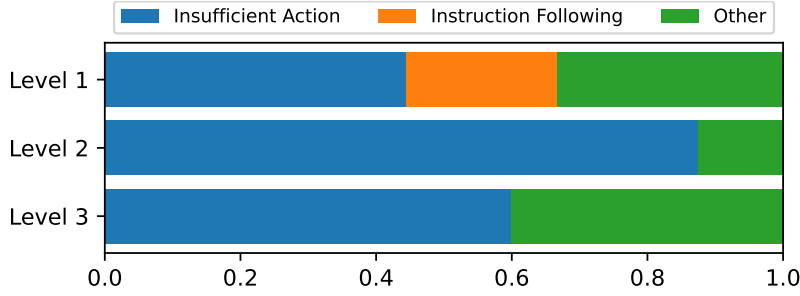


Figure 2: Distribution of error types in tasks where agent A (without action implementation) answers incorrectly, while agent B (with action implementation) answers correctly.

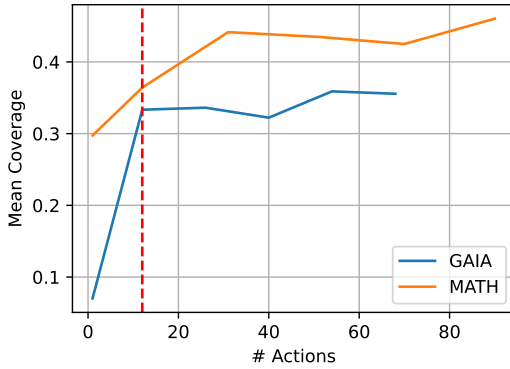


Figure 3: Mean coverage over the validation set as the number of actions increases. The red dashed line marks the point where human-designed actions are added.

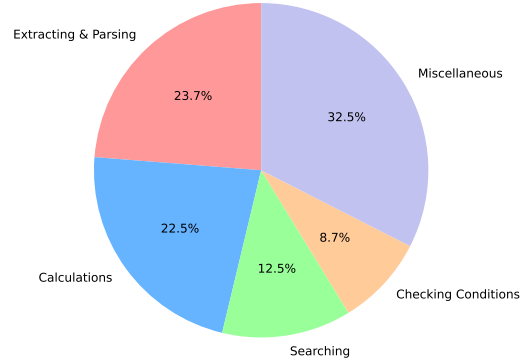


Figure 4: Categories of actions accumulated during evaluation on GAIA validation set.

we obtained a set of 22 tasks. We employed OpenAI’s o1 model (o1-preview-2024-09-12) as 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 Agent A’s 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 2. 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.

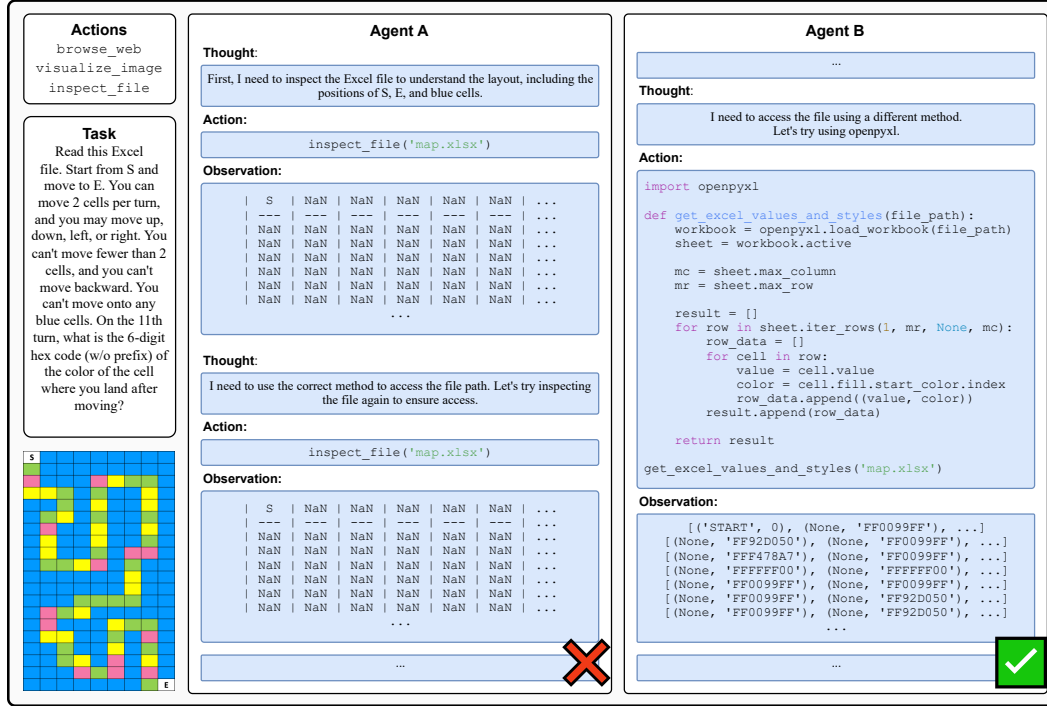


Figure 5: A case study demonstrates the difference in problem-solving flexibility between Agent A (a variant of DynaSaur without action implementation) and Agent B (the proposed agent framework). Both agents begin with the same initial step, but only Agent B, equipped with the ability to implement its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

4.3.3 How Useful Are Generated Actions on Unseen Tasks?

In this experiment, we evaluate how the utility of a generated action set improves as the agent accumulates more actions over time. To quantify this utility, we propose an *Action Coverage* metric, which measures how often an agent relies on the actions in a given set $\mathcal{A}$ to solve an unseen task u . Formally, given a task u , a ground-truth answer y , and an action set $\mathcal{A}$, we sample a trajectory $\tau = \{(a_1, o_1), \dots, (a_T, o_T)\}$ from the policy $\pi_\theta(\cdot | \mathcal{A}, u)$, where a_i and o_i denote the action and observation at step i (we omit the intermediate thought sequence h_i for brevity). We consider the task successfully solved if the final observation $o_T = y$. At each step, the agent either reuses an action from $\mathcal{A}$ or generates a new one not in $\mathcal{A}$. If the agent must generate a new action to complete the task, we consider the action set $\mathcal{A}$ insufficient for that task. We define the *coverage* of $\mathcal{A}$ over u as the proportion of steps where the agent uses actions from $\mathcal{A}$, conditioned on successful task completion:

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

However, computing this metric exactly is prohibitively expensive. In practice, we approximate it by sampling a single trajectory for each task. We report the average Action Coverage over the test sets of GAIA and MATH, as visualized in Figure 3.

Initially, when $|\mathcal{A}| = 1$ (consisting only of the submit_final_answer action), coverage is low for both datasets, as expected. However, it is not zero, since a small number of tasks in both GAIA and MATH can be solved by directly submitting the final answer at the first step. When human-designed tools are added (marked by the red dashed line), Action Coverage increases substantially for GAIA, but less so for MATH. This is because the tools are well-aligned with the steps required in GAIA tasks, such as web browsing and file reading, but are less applicable to mathematical problem solving. In the subsequent steps, as the agent

accumulates generated actions, coverage increases for both datasets, with a stronger rise observed in MATH. Our analysis suggests this is due to the domain mismatch: on GAIA, the agent continues to rely heavily on the human-designed tools; whereas on MATH, where those tools are less useful, the agent increasingly relies on its own generated tools.

4.4 Statistics of the Generated Actions

We present summary statistics and analyses of the actions generated during evaluation on GAIA tasks. This section includes a breakdown of action types, categorization by functionality, and a complexity analysis of the generated code. Additionally, we provide concrete examples of both successful and failed actions in the Appendix.

4.4.1 Action Statistics and Categories

A total of 174 actions were generated, comprising 80 actions accumulated during training (from 165 examples) and 94 newly generated during testing (from 300 examples). We manually inspected each action and grouped them into categories based on their functionality. The distribution is shown in Figure 4. Specifically, 23.75% of the actions are dedicated to extracting and parsing information from data or files, 22.5% perform calculations, 12.5% involve searching operations, 8.75% check conditions (e.g., evaluating whether a statement is true or false), and the remaining 32.5% fall into a miscellaneous category, including tasks such as file conversion and counting.

4.4.2 Complexity Analysis

To assess the complexity of the generated actions, we use the cyclomatic complexity metric (McCabe, 1976), which quantifies the number of linearly independent paths in a program’s control flow. A cyclomatic complexity below 10 is generally considered a threshold for maintainable code, while higher values may indicate more intricate and error-prone logic. Based on this metric, the generated actions exhibit an average complexity of 3.06, slightly lower than the average of 3.72 observed in human-authored actions.

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 (we omit the full steps due to space constraints). We include additional case studies on the benefits of dynamic action creation in Appendix C.

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; Song et al., 2024c),

supervised fine-tuning (Schick et al., 2023b; Zeng et al., 2023; Chen et al., 2024b; Zhang et al., 2024a; Chen et al., 2023b; Wang et al., 2024a), or reinforcement learning (RL) algorithms for self-exploration (Zhou et al., 2024b; Song et al., 2024b; Yang et al., 2024; Aksitov et al., 2023; Christianos et al., 2023; Abdulhai et al., 2023; Gulcehre et al., 2023; Song et al., 2024a). However, these approaches mainly study agents under the assumption that the set of actions is fixed and provided by the environment. Furthermore, most existing work uses text (Schick et al., 2023b) or JSON (Qin et al., 2023) as the representation of actions, which significantly lacks the two criteria mentioned earlier: generality and composability. In contrast, DynaSaur can utilize available actions or create new ones if necessary, using code as a unified representation. In principle, acting with code enables agents to solve any Turing-complete problem.

5.2 LLM Agents for Code Generation

Although using LLMs to generate code is not new, these approaches have a long history dating back to the early stages of LLM development (Chen et al., 2021b; Austin et al., 2021; Hendrycks et al., 2021a). However, this line of research has 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 utilize programming languages as a tool to solve generalist AI agent tasks in the GAIA benchmark, which require multistep execution in partially observable and stochastic environments.

5.3 LLM Agents for Tool Creation

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 agent benchmark GAIA.

6 Conclusion

We propose a novel LLM agent framework that leverages Python as a universal representation for actions. By using a general-purpose programming language, our framework enables the implementation of arbitrary actions as well as compositions of existing ones—effectively addressing the limitations of prior agent systems that rely on a fixed, predefined action set. This design not only enhances expressiveness but also allows the agent to perform more complex, context-specific reasoning and decision-making. Moreover, our framework supports unsupervised accumulation of new actions over time, making it suitable for both online and offline deployment scenarios. This adaptability enables the agent to continually expand its capabilities without manual intervention or retraining. We believe that such flexibility is key to achieving strong generalization across diverse tasks and environments. To validate our approach, we conduct extensive experiments across a variety of challenging benchmarks, including GAIA, MATH, TabMWP, AIME, and GPQA. Results consistently demonstrate the effectiveness of our framework. In addition, our analysis reveals that the agent is capable of autonomously recovering from tool failures caused by unforeseen edge cases—highlighting its robustness in real-world settings.

Ethics Statement

As a proof of concept, our framework allows agents to generate and execute arbitrary Python code. While this approach is not advisable for real-world deployment due to potential security risks, we acknowledge that various safeguards can be implemented to mitigate these concerns. For example, one can apply a safety filter or a world-model-based formal verifier to each action during creation or prior to execution. Furthermore, the agent should be deployed in an isolated environment with restricted inbound and outbound traffic. Limiting file system permissions—such as enforcing read-only access or encouraging minimal edits instead of overwriting files—can further reduce the risk of unintended or harmful behavior. Restricting the agent’s permissions also helps prevent the execution of malicious scripts.

References

- Marwa Abdulhai, Isadora White, Charlie Snell, Charles Sun, Joey Hong, Yuexiang Zhai, Kelvin Xu, and Sergey Levine. Lmrl gym: Benchmarks for multi-turn reinforcement learning with language models, 2023.
- Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Koppa-rapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, Manzil Zaheer, Felix Yu, and Sanjiv Kumar. Rest meets react: Self-improvement for multi-step reasoning llm agent, 2023.
- 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. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023. doi: 10.48550/ARXIV.2312.11805. URL <https://doi.org/10.48550/arXiv.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. Program synthesis with large language models, 2021.
- 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. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712, 2023. doi: 10.48550/ARXIV.2303.12712. URL <https://doi.org/10.48550/arXiv.2303.12712>.
- Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *ArXiv*, abs/2305.17126, 2023. URL <https://api.semanticscholar.org/CorpusID:258947222>.
- Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *CoRR*, abs/2310.05915, 2023a. doi: 10.48550/ARXIV.2310.05915. URL <https://doi.org/10.48550/arXiv.2310.05915>.
- Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning, 2023b.
- 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. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021a. URL <https://arxiv.org/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. Evaluating large language models trained on code, 2021b.
- Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-flan: Designing data and methods of effective agent tuning for large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pp. 9354–9366. Association for Computational Linguistics, 2024a. doi: 10.18653/V1/2024.FINDINGS-ACL.557. URL <https://doi.org/10.18653/v1/2024.findings-acl.557>.
- Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-flan: Designing data and methods of effective agent tuning for large language models, 2024b.
- 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. Pangu-agent: A fine-tunable generalist agent with structured reasoning, 2023.
- Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *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*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/5950bf290a1570ea401bf98882128160-Abstract-Datasets_and_Benchmarks.html.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023.
- 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. Reinforced self-training (rest) for language modeling, 2023.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021a.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *CoRR*, abs/2103.03874, 2021b. URL <https://arxiv.org/abs/2103.03874>.

- Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks, 2023.
- Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13:9, 2024.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023.
- Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 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, 2018. URL <https://openreview.net/forum?id=ryTp3f-0->.
- Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL <https://openreview.net/forum?id=DHyHRBwJUTN>.
- T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4): 308–320, 1976. doi: 10.1109/TSE.1976.233837.
- Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 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, 2024. URL <https://openreview.net/forum?id=fibxvavhs3>.
- OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/ARXIV.2303.08774. URL <https://doi.org/10.48550/arXiv.2303.08774>.
- Cheng Qian, Chi Han, Yi Ren Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Conference on Empirical Methods in Natural Language Processing*, 2023. URL <https://api.semanticscholar.org/CorpusID:258841653>.
- 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. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.
- 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. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530, 2024. doi: 10.48550/ARXIV.2403.05530. URL <https://doi.org/10.48550/arXiv.2403.05530>.
- David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. *CoRR*, abs/2311.12022, 2023. doi: 10.48550/ARXIV.2311.12022. URL <https://doi.org/10.48550/arXiv.2311.12022>.

- Aymeric Roucher. Huggingface agent. 2024. URL <https://github.com/aymeric-roucher/GAIA>.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761, 2023a. doi: 10.48550/ARXIV.2302.04761. URL <https://doi.org/10.48550/arXiv.2302.04761>.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023b.
- 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. Mastering atari, go, chess and shogi by planning with a learned model. *Nat.*, 588(7839):604–609, 2020. doi: 10.1038/S41586-020-03051-4. URL <https://doi.org/10.1038/s41586-020-03051-4>.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *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*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. 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, 2021. URL <https://openreview.net/forum?id=0IX0YcCdTn>.
- David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panniershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi: 10.1038/NATURE16961. URL <https://doi.org/10.1038/nature16961>.
- 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. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017. doi: 10.1038/NATURE24270. URL <https://doi.org/10.1038/nature24270>.
- Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of LLM agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.409. URL <https://aclanthology.org/2024.acl-long.409>.
- Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization for llm agents, 2024b.
- Zirui Song, Yaohang Li, Meng Fang, Zhenhao Chen, Zecheng Shi, Yuan Huang, and Ling Chen. Mmac-copilot: Multi-modal agent collaboration operating system copilot. *CoRR*, abs/2404.18074, 2024c. doi: 10.48550/ARXIV.2404.18074. URL <https://doi.org/10.48550/arXiv.2404.18074>.
- Trase Systems. Trase — AI, uncomplicated., 4 2025. URL <https://www.trasesystems.com/>.

- 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, Çağlar 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. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019. doi: 10.1038/S41586-019-1724-Z. URL <https://doi.org/10.1038/s41586-019-1724-z>.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.
- Renxi Wang, Haonan Li, Xudong Han, Yixuan Zhang, and Timothy Baldwin. Learning from failure: Integrating negative examples when fine-tuning large language models as agents. *CoRR*, abs/2402.11651, 2024a. doi: 10.48550/ARXIV.2402.11651. URL <https://doi.org/10.48550/arXiv.2402.11651>.
- Yulong Wang, Tianhao Shen, Lifeng Liu, and Jian Xie. Sibyl: Simple yet effective agent framework for complex real-world reasoning. *CoRR*, abs/2407.10718, 2024b. doi: 10.48550/ARXIV.2407.10718. URL <https://doi.org/10.48550/arXiv.2407.10718>.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155, 2023. doi: 10.48550/ARXIV.2308.08155. URL <https://doi.org/10.48550/arXiv.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, Houmeir Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Dürr, Peter Stone, Michael Spranger, and Hiroaki Kitano. Outracing champion gran turismo drivers with deep reinforcement learning. *Nat.*, 602(7896):223–228, 2022. doi: 10.1038/S41586-021-04357-7. URL <https://doi.org/10.1038/s41586-021-04357-7>.
- Zonghan Yang, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. React meets actre: When language agents enjoy training data autonomy, 2024.
- Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *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*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023a.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 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, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.
- Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Ren Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. *ArXiv*, abs/2309.17428, 2023. URL <https://api.semanticscholar.org/CorpusID:263310662>.
- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms, 2023.

- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pp. 3053–3077. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.181. URL <https://doi.org/10.18653/v1/2024.findings-acl.181>.
- 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. Agentohana: Design unified data and training pipeline for effective agent learning, 2024a.
- Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges, 2024b.
- 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. 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, 2024a. URL <https://openreview.net/forum?id=oKn9c6ytLx>.
- Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. Archer: Training language model agents via hierarchical multi-turn rl, 2024b.

Agent Pipeline	Qwen2.5-32B-Instruct				Qwen2.5-Coder-32B-Instruct			
	Level 1	Level 2	Level 3	Avg.	Level 1	Level 2	Level 3	Avg.
No Pipeline	5.66	5.81	0.00	4.85	13.21	1.16	0.00	4.85
Sibyl	24.53	10.47	0.00	13.33	20.75	8.14	0.00	10.91
HF Agent	26.42	11.63	3.85	15.15	13.21	10.47	3.85	10.30
DynaSaur	35.85	30.23	3.85	27.88	35.85	20.93	11.54	24.24

Table 4: Performance comparison between various baseline methods and our proposed approach on the GAIA benchmark, evaluated under two LLM backbones: Qwen2.5-32B-Instruct and Qwen2.5-Coder-32B-Instruct. “No Pipeline” refers to the baseline where no agent pipeline is employed, and the raw LLM is used. Each value represents the average exact match percentage between the predicted answers and the ground truth.

#	Action Header	Description
1	submit_final_answer	Submits the final answer to the given problem.
2	get_relevant_actions	Retrieve k most relevant generated actions given a query.
3	informational_web_search	Perform an informational web search query then return the search results.
4	navigational_web_search	Perform a navigational web search query then immediately navigate to the top result.
5	visit_page	Visit a webpage at a given URL and return its text.
6	download_file	Download a file at a given URL.
7	page_up	Scroll the viewport up in the current webpage and return the new viewport content.
8	page_down	Scroll the viewport down in the current webpage and return the new viewport content.
9	find_on_page_ctrl_f	Scroll the viewport to the first occurrence of the search string.
10	find_next	Scroll the viewport to next occurrence of the search string.
11	find_archived_url	Given a url, searches the Wayback Machine and returns the archived version of the url that’s closest in time to the desired date.
12	visualizer	Answer question about a given image.
13	inspect_file_as_text	Read a file and return its content as Markdown text.

Table 5: List of initial actions used in this project.

A Additional Experiments on Open-source Large Language Models

To evaluate whether our method generalizes beyond proprietary models and remains effective when paired with models with weaker code generation abilities, we tested two open-source models: **Qwen2.5-Coder-32B-Instruct**, a code-specialized, instruction-tuned model, and **Qwen2.5-32B-Instruct**, a general-purpose, instruction-tuned model with weaker code generation capabilities. The results, shown in Table 4, highlight that DynaSaur works effectively with open-source models, not just proprietary LLMs. Even with a general-purpose model like Qwen2.5-32B-Instruct, which has weaker code generation abilities, DynaSaur significantly outperforms other pipelines. Surprisingly, Qwen2.5-32B-Instruct even slightly outperforms its code-specialized variant. We hypothesize that its stronger commonsense reasoning and decision-making capabilities may compensate for its weaker coding skills, ultimately resulting in better overall performance.

B Implementation Details

B.1 Initial Actions

We present the list of initial actions used in this project, along with their descriptions, in Table 5. Actions 3 to 13 are adopted from Microsoft’s AutoGen (Wu et al., 2023).

B.2 Prompt For Qualitative Analysis

The prompt for qualitative analysis with OpenAI’s o1-preview model is shown in Figure 6.

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.

B.3 DynaSaur’s System Prompt

The system prompt used for DynaSaur is shown in Figure 7.

C Additional Case Studies

We present another comparative case study of two agents: one without action implementation (referred to as agent A) and one with action implementation (referred to as agent B), illustrated in Figure 8. In this scenario, both agents are provided with a binary operator $*$ defined by a table and tasked with finding a counterexample to demonstrate that $*$ is not commutative. Successfully solving this task requires symbolic reasoning abilities. Agent A, lacking the necessary actions to address this task thoroughly, attempts reasoning within its Thought sequence but ultimately submits an incorrect answer. In contrast, agent B dynamically generates a specialized function to tackle the question. This action is general enough to solve other instances of the original problem as well. This example further highlights the advantage of equipping agents with the ability to dynamically generate and execute actions through code to tackle a range of problems.

D Examples of Generated Actions.

We include examples of both successful and failed generated actions in Figures 9 and 10. A generation is considered successful when the action is reasonably generalizable and applicable across various contexts. Conversely, an action is considered a failed generation if it contains hard-coded values or is too context-specific to be reused in different tasks.

```

# 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 trying until you reach the maximum iteration limit.

```

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

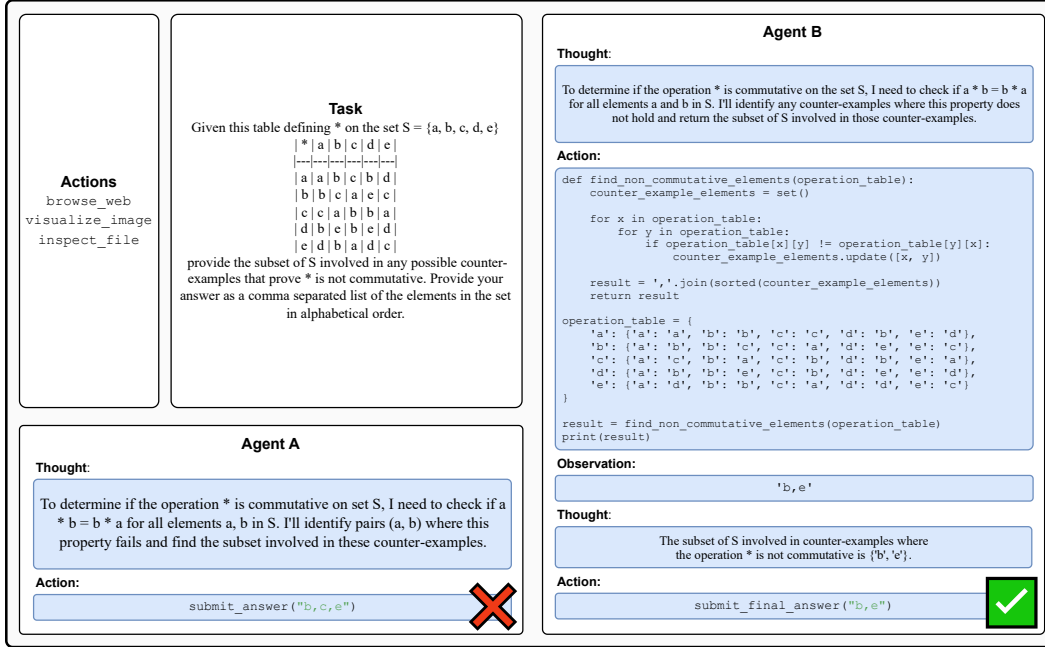


Figure 8: A case study demonstrates the difference in problem-solving flexibility between Agent A (a variant of DynaSaur without action implementation) and Agent B (the proposed agent framework).

```
import fitz

def extract_text_from_pdf(file_path: str) -> str:
    """Extract text from a PDF file."""
    text = ''
    with fitz.open(file_path) as pdf:
        for page in pdf:
            text += page.get_text()
    return text

from openpyxl import load_workbook

def inspect_excel_file(file_path: str):
    """Inspect data from an Excel file."""
    workbook = load_workbook(filename=file_path)
    sheet = workbook.active
    data = []
    for row in sheet.iter_rows(values_only=True):
        data.append(row)
    return data
```

Figure 9: Successful examples of generated actions in GAIA.

```
def calculate_food_sales(sheet) -> float:
    """Calculate the total sales from food items in the given Excel sheet
    """
    total_sales = 0.0
    for row in sheet.iter_rows(min_row=2, values_only=True):
        total_sales += sum(row[1:6])
    return total_sales

def count_crustacean_mentions(slide_text: str) -> int:
    """Count slides mentioning crustaceans in the provided slide text."""
    crustaceans = ['crayfish', 'isopods', 'Yeti crab', 'Spider crab']
    count = 0
    for crustacean in crustaceans:
        if crustacean in slide_text:
            count += 1
    return count
```

Figure 10: Failed examples of generated actions in GAIA.